COP 2805C (Advanced Java Programming) Lecture Notes

Lecture 1

Welcome! Introduce myself. *Make sure students update Zoom client to latest version! Some features are new, some won't work using the web browser Zoom client.*

Review: COP-2800 emphasized learning fundamentals of programming and most of the Java language. As we discussed then, you only learn that by writing programs. A lot of programs. You have not written nearly enough programs!

However, there are other skills needed that were not covered in the first course. Employers today want programmers who understand modern programming principles, including object-oriented design, distributed programming techniques, client/server software design, and who understand the process of producing software: design, implementation, testing, debugging, and maintenance. Along with all kinds of development methodologies, employers will expect some familiarity with related environments or toolsets (that they've spend money on already). Such activities (in addition to actually implementing software, or *programming*) are known as ***software development***. In turn, ***Software Engineering*** includes software development and the management of software projects, development teams, and the process of software development.

> Writing the code is often the easy part. The hard part is understanding the existing code base, understanding project requirements, designing application structure, etc. You need to be able to read other people's code and understand what you read, in order to debug, enhance, perform a code review or compliance audit, and other things.

In this course, we emphasize learning software development skills: use of tools, workflow, group communications, and learning by reading code. We will also learn more about Java. Naturally, you should still be writing lots of code too!

Basic Java you should already know (covered in COP-2800):

- Data types and their variables; when each type is appropriate
- Computer math issues (over- and under- flow, floating point round off errors)
- Control Structures
- Primitive vs. Object reference variables, and wrapper classes
- Methods (return value, parameter passing)
- Scope and modifiers
- Object based and Object-Oriented programming
- Garbage Collection
- Nested and Inner classes, Lambdas

- Packages (`CLASSPATH`)
- Java Doc (API) comments
- Inheritance (`super`, `this`, overload vs. override, method signatures)
- Abstract classes and methods
- Interfaces (including functional interfaces used with lambdas)
- Collections basics
- Basic graphics (delegation event model, AWT, swing, and JavaFx; multi-media)

In-class program: Without using a computer at all, write a Java program that display the integers from 1 to 100, replacing those divisible by 3 with the word "`Fizz`", those divisible by 5 with the word "`Buzz`", and those divisible by both 3 and 5 with the word "`FizzBuzz`". Time limit: 5 minutes.

> **This is in fact a common sort of interview question**, designed to weed out those who can't code at all. (A depressingly large percent of applicants fail this simple test.) Nearly all employers will require interviewees to write some simple code on a blackboard, without any computer, as part of the early interview process. (Later in the interview process, you may need to design more complex code for some positions, but you won't have to implement it on a blackboard in 5 minutes.)

Let's start off with a number of short topics, to provide some basic concepts and tools you should know. First up, what's in a `.class` file?

Bytecode demo: Show `javap -c` *`NameOfClass`*. Show `.class` hex with Cygnus or Neo Hex Editor. That "machine code" can be converted back into Java, with the right tools. Hex editors allow one to edit machine code (that's what hackers do), edit or inspect images or any non-text file.

Slow code? Use **`visualvm`** tool to get profiling information to see the CPU and memory hogs in your code. To use this tool, compile your code using the extra options "`-g -parameters`" to include all debugging info possible inside of the generated `.class` files. (Demo: Start `jvisualvm` and some Java app, e.g., davmail or Eclipse. Show the information available: monitoring, heap snapshot, profiling data. Add plugins.) Full documentation is available at visualvm.github.io.

A similar tool that comes with the JDK (for now) is called **`jconsole`**. You can start a Java application, then you need to find its PID (process ID). On Windows, start Task Manager and click on the processes tab. On Mac and Linux, use the "`ps -ef`" command. Then launch `jconsole` as "`jconsole` *`PID`*". (*Show.*)

Bytecode is specific to a particular version of the Java virtual machine. The JVM has been updated over the years with new features and changes to acceptable

bytecodes. The current JVM always supports several older versions (but not all). For compatibility reasons, when compiling code for an older JVM version you must use the `--release` *`version`* option. To run older code you likewise use the `java --release` *`version`* option to specify *`version`* of the JVM to use. (And for both commands, you can also specify `--enable-preview` to allow experimental Java features to be used.

Java Flight Recorder is a set of monitoring and profiling tools that, prior to JDK 11, were part of Oracle's commercial version only. It is now open sourced but requires a separate download of one of the components, JDK Mission Control (JMC). You can find the current version at jdk.java.net or at bell-sw.com (Liberica Mission Control) To begin data collection, you run your program with the option -XX:StartFlightRecording=*collectionOptions*. You can see the documentation for these options in the java tool documentation (*extra options for Java* section). You can also use `jcmd` *`PID`* to begin recording/monitoring of an already started Java program; you can do that from within JMC too.

The monitoring done by JMC is similar to that of jconsole or VisualVM (that is, a JMX console). The extra power of JFR shows when you begin to collect and save data in a file. You use the GUI tool JMC on that file to review the data. Unlike other profiling tools, JFR is built in to the JVM and has extremely low overhead.

See this dzone.com article for a tutorial.

JRE Redux

A JRE is more than just a JVM and library. (I ~~lied~~ over-simplified last semester.) **It also provides services** to your Java programs, through the library "API". After all, the JVM is completely isolated from your system. Without the services provided by the JRE, your programs couldn't access files, the Internet, a printer, or any other resources. If you use (for example) `java.awt.print`, you code will compile just fine. But if the JRE didn't provide the print service, your program wouldn't print. (You can think of the JRE as a mini operating system for the virtual computer, the JVM.)

There are several different JREs defined! Applets (now deprecated) use a JRE that provides security, audio, and other services. (If you create a stand-alone Java program that creates, `init`s, and `start`s an Applet, that Applet still won't be able to play audio, since the Java SE JRE doesn't provide that service.)

JREs officially defined include ones for Java WebStart (deprecated), Java SE, Java Applets (deprecated and removed), Java ME, and Jakarta EE (EE4J as of 10/2017, Jakarta EE as of 6/2019). In addition, each of these has officially designated subsets of APIs and services defined, called *profiles*. (There are three of these for Java SE ≥ 8 alone.) (Since Java 11, Applets and WebStart are gone.)

In 2017, Oracle gave control of Jakarta EE to the Eclipse Foundation. They decided to rename Java EE to **Eclipse Enterprise for Java**, or ***EE4J*** since Oracle continues to hole the trademark "Java". EE4J is based on Jakarta EE 8. The final name as of 2019 is Jakarta EE. However, as of 5/2020 Oracle and Eclipse have not come to any agreement about the use of `jakarta.*` packages and Maven artifact names. (See this 2/2019 blog post.) The new names will be `jakarta.*` for Jakarta EE 9, which was released in late 2020. This renaming was a breaking change!

Oracle also contributed Glassfish, the Jakarta EE reference implementation, and the Jakarta EE Technology Compatible Kit (TCK), used to validate implementations; the TCK is now open source.

In this course, EE4J, Java EE,, J2EE, JEE, and Jakarta EE are used interchangeably.

The JREs defined for Jakarta EE are called *containers*, such as a servlet (web app) container and an EJB container. (The JREs used for Java ME (Micro Edition, but really *Mobile Edition*) have other names.) Jakarta EE containers are also sometimes called *application servers* and *middleware*. (I think they are called containers because they contain applications that use their services.)

So, to create a Java application that uses the services offered by the Jakarta EE standard, you need an enhanced JRE that provides those services. Examples of these include the popular Tomcat (which provides some of the services defined for Jakarta EE, enough for most simple web applications), Glassfish and JBoss (both of which provides all of the services defined in the Jakarta EE standard, including email, messaging, timer services that can start programs automatically, etc.)

Many of these *containers* (Java application servers) are modular in design, allowing you to "plug-in" different ***service providers*** to provide some of the services. For example, the code in the container that implements database persistence services to applications, is called the *persistence provider*; EclipseLink is one such provider, and the one included with the Glassfish container. But if you don't like EclipseLink, you could plug-in a different persistence provider (such as Hibernate or OpenJPA) into such containers.

To compile your programs that use services such as `jakarta.persistence.*` classes, you will need to add your service provider's Jar file(s) to your `CLASSPATH`. These are usually found in a "`lib`" sub-directory where you installed the container.

You don't need a different JDK! To create Jakarta EE apps, you only need to have the required Jakarta jar files found on `CLASSPATH`. `javac` will compile them just fine, but you cannot run them using the Java SE JRE.

(See the Jakarta EE Tutorial for more information.)

The Oracle (and thus, standard) JREs are not the only ones out there. The most popular (2015) mobile platform in the world is Android, which takes a Linux kernel and runs a JRE on top of it, called *Dalvik*.

> The Dalvik JRE is not directly compatible with Oracle's JRE SE. The virtual machine architecture is different (stack based for Oracle, register based for Dalvik) and the byte codes are thus different. So to create a Java app for Android, you need the appropriate compiler. (Dalvik uses bytecode in `.dex` files, or *Dalvik executable*.)
>
> One other difference worth mentioning: When an app is installed on Android, it is "optimized", meaning the bytecode is changed to directly access the device. Such modified byte code is stored in ".odex" files.

Static and Dynamic Linking

There are two types of libraries in the software world: static link libraries and dynamic link libraries (or "DLLs"). (The Linux term for DLLs is *shared object* files, which have an extension of "`.so.<version>`".) Which type you use depends on the type of application you are compiling, a static or a dynamic one.

With *static* linking of executables, all the code used by the application is copied from the various libraries (and other object files) into the executable file. **Such a statically linked application only needs the OS to run.** (Note that if some function is needed from a library, the whole object file from that library containing that one function must be copied, not just the one function needed. For this reason, many libraries contain many small object files rather than just a few large ones.)

As the functions are appended to the executable file being built, their memory addresses are determined. The app code that invokes such library functions is modified to use the correct addresses for the functions used. This process is called ***link-editing***, or simply *linking*.

In contrast, **dynamically linked executables don't actually copy the object code from libraries into executables**. At *runtime*, the DLL libraries used by the application must be loaded into memory separately; the runtime locates the DLL files on disk and opens them if they aren't already loaded into memory. The executable must then be linked to the loaded library, so its functions can be invoked. (This requires linking to occur at runtime.) This has the benefit of reducing the executable's size, allows sharing of libraries in memory, and allows the libraries to be updated independently of the applications that use them. It does become the end-user's responsibility to ensure the correct libraries are installed and up to date, and if some updated library breaks your app, too bad. Testing after any update is a must. Statically linked executables are unaffected by library updates; you must recompile them to use the new versions (and then test them).

Java applications don't have a choice; they are dynamically linked. This is one reason why the end user must have a proper JRE installed. Sun and later Oracle have gone to great pains to ensure updates won't break your applications.

Integrated Development Environments (IDEs) and Tool-chains

Command line tools are very useful and popular. But, **large programs have many steps to perform: edit, compile, test, document, package, and deploy**. This makes basic command line tools such as those found in the JDK tedious to type. Fortunately, command line tools can be scripted. The various tools used, such as `notepad`, `javac`, and `jar`, are called the ***tool-chain*** (or *toolset*).

Using DOS cmd.exe, Windows PowerShell, or Linux shell scripts works fine (MS integrated Linux into Windows mostly to support developers, so only one script language was needed), but everyone needs to create them (if the application is complex, or if the deployment steps are tedious), and that is not an easy skill to master. There are a number of tools such as Ant (rarely used anymore), Maven, and Gradle that are much simpler to use than creating scripts from scratch. These tools, known as ***build tools*** (or sometimes *dependency-management tools*), have predefined tasks you can easily use and customize. Maven seems to be the most popular (2019). (Gradle is interesting as it can be customized using Groovy, a Java-like scripting language that runs on the JVM.) Also popular is the so-called *continuous integration* ("CI") tools, such as Jenkins (and its ancestor, Hudson) or GitHub Actions. There are also tools such as Gerrit, which can help automate much of the code review process.

A common non-GUI tool-chain is JDK, Git, Maven, Gerrit (web-based code review), and Jenkins, which can all work together nicely. (These tools have plug-ins for Eclipse and other IDEs, if you prefer a GUI.) Today, GitHub has code review and CI tools built-in, so naturally that is very popular.

> You must learn something about Ant, Maven, and Gradle as use of these tools is common. (Ant is not used as much anymore.) Maven is discussed in detail below.

Not a fan of the command line? Another option is to use a GUI toolkit, called an IDE. This provides access to a large collection of tools that all use the same GUI interface. The IDE can make a programmer very proficient but IDEs can be difficult to master: you still need to understand all the step and concepts, then master the user interface.

There are many IDEs to choose from, and each one works and looks different. Some are designed for student use and come with fewer tools but provide student-friendly features. For Java development, jGRASP, DrJava, and BlueJ are well known IDEs designed for students. (Due to historical bad feelings by Microsoft toward Java, Visual Studio does not support Java (2022), although VS code does.)

> There are many popular text editors designed for programmers these days. They have some of the features of IDEs as well, often available as plug-ins. Microsoft's VS code is very popular now. It includes an extension "live share" that allows students to collaborate while editing. (A similar tool is repl.it, a browser-based editor.) For quick editing I personally use Notepad++, otherwise Eclipse or VS Code.

There are also many IDEs designed for professionals. These can take a lot of time and effort to master (just like the command line) but are generally considered worth the trouble. By far the most popular professional grade IDEs for Java are Eclipse, NetBeans, and IntelliJ IDEA (which has a free but limited edition and a full commercial edition that is free for students. The limited version does not support Jakarta EE!) NetBeans seems more polished than Eclipse, but Eclipse is backed and used by IBM and Google, and some plugins are only available for Eclipse. (Of course, the reverse is true too, but I know of some cool Eclipse plugins.) Note Eclipse is flash drive friendly. It doesn't use the Windows registry and doesn't install stuff elsewhere, so you can carry it around with you on a flash drive. (It's written in Java!) Currently, I prefer IntelliJ IDEA and Spring Tool Suite 4 (a customized version of Eclipse).

> Even when using a GUI IDE, you should know something about the command line tools. Most IDEs can use those tools via plugins; such projects don't depend on which IDE you used (so other developers on the same project can use a different IDE).
>
> In addition, the command line tools are used by continuous integration tools, and you will need to know them to set those up and configure them.

Discuss Java Install and developer setup, show Eclipse (can use any, e.g. NetBeans). *Do HelloWorld in IDEs, if time.* Later we will learn to use build tools such as Maven, as well as versioning and collaboration software such as Git.

> **Note:** If you have a 64-bit computer, you should install the 64-bit version of the JDK and the 64-bit version of IDEs and other tools.

Note: the JDK no longer installs the Derby database, a.k.a. "JavaDB", in `JAVA_HOME\db`. You can instead install the latest version directly from db.apache.org/derby. (That site has good documentation, regardless.) To use this DBMS, you will need to put some Jar files on your `CLASSPATH`. I recommend adding `...\db\lib\*` to CLASSPATH. Or you can copy the files `derby.jar`, `derbyclient.jar`, `derbynet.jar`, and `derbytools.jar` to your own extensions directory, say `C:\Java\MyExt`, then list that directory on the classpath: `CLASSPATH=.;C:\Java\MyExt\*`.

> A more popular database for use with Java is the H2 database; it supports all the features of Derby. Today (2024), it is also possible to run a standard DBMS such as PostgreSQL in an in-memory container, so that is also a common option.

You will also need to set some ***environment variables*** and modify others; the installer won't always do this for you: `JAVA_HOME` (where you installed the JDK), `DERBY_HOME` (usually `JAVA_HOME/db`), and prepend to `PATH` both `JAVA_HOME/bin` and `DERBY_HOME/bin`. You may need additional environment variables, and additional locations listed on `PATH`, once you install other command line software such as Maven ("`mvn`") or Gradle.

> If you don't know what an environment variable is, you need to learn about that and other basic concepts including using the command line. Please ask your instructor!

Setting environment variables varies by system (and version!), and Windows was one of the most difficult. I recommend using Rapid EE (environment editor) for Windows 8 or older. (The built-in Windows 10 and 11 editors are good enough.)

(Note, the use of environment variables in the values for `CLASSPATH` and `PATH`, such as `JAVA_HOME`, won't always work since Windows XP; avoid using them.)

> A professional-grade IDE can help an experienced developer do a better job, faster. It often impedes novices though, who must learn a complex IDE which can hide some steps taken, hindering comprehension. This is not obvious when creating a *Hello, world!* project!

IDEs set various environment variables on a per project basis, using the system set values by default (usually). You can change these settings for any project, or for default project templates (used when creating new projects). Setting `CLASSPATH` is common, although it may not be called that; in the project's properties, look for *classpath*, *buildpath*, *external* Jars, or some such setting. For example, in Eclipse, when adding the first JUnit test case to your project, Eclipse will prompt you to add the Jar file for that to your project. Say yes. (JUnit 5 consists of several Jars.) It may be easier in the long run to edit the Java project template, to include `C:\Java\MyLibs\*` (where I put all such Jars) automatically.

Other items include *run configurations*. You can create several new ones, such as running a Java stand-alone app with certain command line arguments (to enable flight recorder for instance), or via JUnit (for testing). In your IDE project, you are asked to implement a trivial program, but you should take the opportunity to explore at least one IDE in some detail.

Test your setup by opening a fresh command line window (after setting the environment variables), and trying some commands you've installed:

```
java --version
javac --version
ij NUL
sysinfo
```

(The last two are from Derby database, so they won't work if you install some other DB.)

> Instead of opening a fresh window, you can use the command `refreshenv` on Windows to reset the environment variables.

Once you have basic Java setup, you can add many other components. Complete directions are found in the class resources, including how to test your setup. You will use most or all of these tools in our class. Run similar commands to the above to test all newly installed software is working.

Show Java Setup web resource.

[Discuss using Maven or Gradle, then demo. Add to IDE project.]

One last tool to mention is Graal VM. Graal is built from OpenJDK but designed for speed. The interesting bit is that Graal includes a tool to build native executables! See www.graalvm.org for more information. For a good overview, see this Guide to Graal VM. (Maven can create executable jar files but that's not the same.) (Show Graal Demo.)

The JDK includes many standard tools such as jar, java, javac, and about two dozen others. See JDK Tools for a list.

Apache Maven — [*Parts from: maven.apache.org*]

Maven, a Yiddish word meaning *accumulator of knowledge*, was originally started as an attempt to simplify the build processes in the Apache Jakarta Turbine project. There were several projects, each with their own Ant build files that were all slightly different, and Jars were checked into CVS (an older versioning system). They wanted a standard way to build the projects, a clear definition of what the project consisted of, an easy way to publish project information, and a way to share Jars across several projects. The result is a tool that can now be used for building and managing any Java-based project.

Maven is a collection of technologies that simplify many aspects of working with software projects, especially Java ones. Maven uses repositories of code (binaries and source) and Maven plugins, and can manage their dependencies and verify their integrity. (Dependency management is a key technology!) Maven defines a standard project folder layout (which you can change, but shouldn't). Most other build tools use this same Maven layout:

```
C:.
|   pom.xml
|
└───src
    ├───main
    |   ├───java
    |   |   └───mypackage
    |   |           App.java
    |   |
    |   └───resources
    |           some.gif
    |
    └───test
        ├───java
        |   └───mypackage
        |           AppTest.java
        |
        └───resources
```

(Additional files such as `ReadMe.md`, `.gitignore`, and `License.txt` go into the top-level of the project. Other projects such as for Spring (a framework for Jakarta EE) may have additional directories as well.)

Maven defines *project lifecycle phases* such as compile, test, package, install, and others; a typical use of Maven is to edit your files, then run "`mvn install`" to do all phases up through `install`. Unlike other tools, Maven already knows how to do all these tasks by having defaults (which vary depending on the project type). So unless you wish to change the defaults, you need to specify very little. This also means you can tell someone how to build your project with very simple directions; do "`mvn clean install`" should be all you need to say!

Maven has many standard plugins that support (for example) JUnit tests, deploying web applications, and generating skeleton projects (usually, running these just produces "`Hello, world`" on the output) of various types.

Maven can provide change log documents created directly from source control, cross referenced sources, mailing lists, dependency lists, unit test reports including coverage, and other information about a project, in a variety of formats (including HTML).

To use all that, you will need to know a few Maven concepts and jargon. There are excellent on-line (and free) books for Maven. I recommend *Maven by Example* to get started (see resources for link).

Maven is currently at version 3.9. Version 4.0 is nearly released (as of 5/2025) and will include many changes and improvements. See what's new in Maven 4 for details.

Maven Repositories ("Repos")

To begin with, Maven stores every built project, plugin, and library in one or more Maven *repositories*. Maven calls the items in its repos *artifacts*. Every developer has a repository (also called a *repo*) created in their home directory called "`.m2`". All your built projects go in there, making it easy for you to reuse your stuff.

There is also a central Maven repository Maven Central (central.sonatype.com), containing over 719,779 published artifacts (as of 8/2025). When your project needs one, Maven will automatically download it, validate it, and store a copy in your local repo for all your projects to use. What's really cool is that if that artifact depends on others, Maven will automatically fetch those as well! Use the Maven search feature to find projects (and their versions), or you can just browse Maven Central. (Search for `org.junit.jupiter` and `org.junit.jupiter.api` (for the javadoc.jar file).)

Qu: How secure and trustworthy are 3rd party artifacts from Maven Central? Ans: Apache foundation makes the rules for Maven. The company Sonatype hosts Maven Central and enforces these rules. To protect everyone using Maven Central, an author must create a publishers account with Sonatype, provide a namespace (for example, com.wpollock.*anything* or github.com/wpollock/*anything*), a public GPG key, and an email address, all of which Sonatype will validate. All artifacts published with a that namespace must be signed with that key. All artifacts published must include developer info, license info, and more (in the POM). As an extra safety measure, Sonatype will scan the uploaded source for known security issues and report them and (some) code quality metrics, since 2022. (View slf4j-api as an example.) **However, the code is not tested for fitness, compatibility with older versions, or even unknown malware.** No large repo can do that, it is up to users when they use other peoples' code. (More about security below.)

> Maven central is so popular that many Java projects only published to there and do not maintain a download page or website of their own. So even if you don't use Maven, you may still need to use the Maven Central repo. This repo is a good place to get things such as the older javaee-api.jar file or the new jakarta.jakartaee-api, and their javadoc (API doc) which comes bundled as a Jar instead of a zip and has a name like `jakarta.jakartaee-api-10.0.0-javadoc.jar`. (Maven cannot create zip files, but you can open any .jar file with any zip file reader.)

The first time you use Maven locally, a ton of plugins will be downloaded. This is because the Maven tool is very small and uses plugins for most of its functionality.

In addition to your personal (local) repo and Maven Central, you or your organization can create additional Maven repos, for either public or private use.

Maven Coordinates

Every artifact in the Maven world has a globally unique name, or *Maven coordinate*, consisting of a *groupId* (e.g., "`com.wpollock`"), an *artifactId* (e.g., "`my-thingie`"), and a *version number* (e.g., "`1.0.0`"). (Also a package type, but that isn't used to uniquely identify an artifact.) Maven coordinates are used in a project's configuration file to specify dependencies and which Maven plug-ins to use. They look like this:

```
groupId:artifactId:packaging:version
```

For example:

```
com.wpollock.web:some-app:jar:1.0.0-SNAPSHOT
```

> By convention, projects undergoing active development have a version string that ends in "`-SNAPSHOT`". (When first developing new software, the major version should be zero, not one. This indicates an unstable project: you are at the exploratory stage, or what I call *noodling around* stage). Once the project is ready to be deployed ("released") the `-SNAPSHOT` string is removed from the version string.)

Note that if you specify some artifact (or plug-in) will be used in your project, you get that exact version even if a newer version has been published. This makes for ***repeatable builds***. (Maven does have a plugin available to check for newer versions for you.) Repeatable builds are a good thing, but not using the latest version of your dependencies means it is on you to check for bugs and security issues with the version you use. (The flip-side is that updating versions can and often does break your software, causing lots of unexpected developer effort to find compatible versions of everything.)

Version numbering in Maven

This is a surprisingly complex topic. Maven 4 and earlier require Semantic versioning 1.0.0 also known as semver (and not the current 2.0.0 standard!) style version numbers, or Maven may not compare versions correctly. There are some differences from that in Maven, such as case-insensitive comparisons. You can read the gritty details at Version Order Specification. Here, I'm only showing a simplified version of the rules (you may not think this is simple but that's not my fault).

Such numbers have the format of X.Y.Z (or *Major.Minor.Patch*). Bug fixes not affecting the API increment the patch version, backwards compatible API additions/changes increment the minor version, and backwards incompatible API changes increment the major version. Pre-release (unstable API) versions have a major number of zero. When incrementing X, Y and Z are reset to zero. When incrementing Y, Z is reset.

The version number can be followed by a dash and a string (letters, digits, and a dash). These are pre-release versions and are compared lexically (ASCII case-insensitive order). Examples are "`-beta1`" and "`-rc2`". All pre-release versions are older (lower precedence) than the same version without the "`-string`".

Violating these rules turns off Maven's normal version comparison, and an alternate comparison is used which can cause unexpected results. For example, 1.1 < 1.10 < 1.9 (Not semver, so alt comparison used), but 1.1.2 < 1.9.2 < 1.10.2 (normal comparison used). The alt comparison is usually not what you would expect!

When working on a new X.Y.Z version, Maven uses X.Y.Z-SNAPSHOT. This version is under development and thus can change anytime for a newer version. When ready for release, the "-SNAPSHOT" suffix is removed. Versions without the "-SNAPSHOT" should be immutable; any change requires a new version number.

Version numbers can be inherited from parent POMs, so you don't always need to specify a version in your POM. The flip-side of that is, when you need a newer version of a Maven plugin, `mvn.exe` will by default use the older version and you will need to add a dependency to your POM to override that. Sooner or later, a newer version of Maven is released which uses more recent versions by default.

When using Git or other VCS (*version control system*) that allows "tags" on commits, you should use tag names of "vX.Y.Z" on any release versions.

When to bump the major, minor, or patch numbers can be confusing and is somewhat subjective. Here is some guidance:

- Adding new stuff is generally a non-breaking change, so increment the minor number. Adding new stuff with common names may be a breaking change, if some user is already using the new name.
- A bug fix normally only requires a patch number increment.
- Using new language features in your library's implementation that requires your users to upgrade their compilers, is a breaking change. Increment the major number.
- A change to the user license is a breaking change; increment the major number.

- Beware of Hyrum's law: Somebody is depending on the old, buggy behavior if your code is used by enough people! If possible, release a newer version that accepts the old (buggy) behavior but issues deprecation warnings instead of crashing. This may not be possible depending how critical the fix is. If your support team hears of the change causing issues, when you do make the change, increment the minor, not just the patch number.
- When making (known) breaking changes, have a path forward for users that doesn't require immediately updating their code. First release a minor version update that includes the new version of the API and that marks the older API version as deprecated. Issue warnings (e.g., in log messages). Include migration advice with the release. At that time (or some time after that change), release a major version update that removes the deprecated parts of the API.

Project Object Model ("POM") Configuration File (`pom.xml`)

In the top folder of each project is its one and only configuration file, **`pom.xml`**. It is often simply referred to as a project's POM. This must be a valid XML file, and is case-sensitive.

This XML file starts with general information about a project, such as its coordinates, project website (if any), names and email addresses of developers, and a description. Only the coordinates are required.

Next comes any settings that you want to override Maven from defaults; these are rarely used but can be used to specify additional repositories, for example.

In Maven's `pom.xml`, you can define *properties*. These are value placeholders (like variable names). Their values are accessible anywhere within a POM by using the notation `${X}`, where `X` is the property name. Or they can be used by plugins as default values (in some cases, automatically as long as you get the name correct). Often, you use properties to define version numbers; that makes it easy to change them.

Next comes a section describing the various plugins your project uses. You can also define various settings for each plugin here.

Finally, there is a section describing all the dependencies of your project. For example, specifying JUnit is done here. This is the section most developers will need to maintain as version numbers change and you want to use the latest version (unless you use a property).

A minimal `pom.xml` would look like this:

```
<project>
  <modelVersion>4.0.0</modelVersion>
```

```
    <groupId>com.example.app</groupId>
    <artifactId>my-app</artifactId>
    <version>1.0.0-SNAPSHOT</version>
  </project>
```

The required items are: modelVersion (should be set to 4.0.0), the groupId (the id of the project's group), artifactId (the id of the artifact you are building), and the version (the version of this artifact, often "1.0.0-SNAPSHOT" to start with). The "packaging" is assumed to be "`jar`" if omitted.

The minimal POM doesn't include many features. A real `pom.xml`, one with some optional sections included, might look like this:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.wpollock.myapp</groupId>
  <artifactId>myapp</artifactId>
  <packaging>jar</packaging>
  <version>1.0.0-SNAPSHOT</version>
  <name>myapp</name>
  <!-- Optional descriptive sections go here -->

  <properties>
    <project.build.sourceEncoding>UTF-8
    </project.build.sourceEncoding>
    <!-- See the java.compiler.release tag, below -->
    <maven.compiler.source>15</maven.compiler.source>
    <maven.compiler.target>15</maven.compiler.target>
    <main-class>com.wpollock.myapp.Main</main-class>
  </properties>

  <dependencies>
    ...
    <dependency>
    <!-- Testing dependencies -->
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter</artifactId>
      <version>5.11.3</version>
      <scope>test</scope>
    </dependency>
```

```
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>3.4.2</version>
        <configuration>
          <archive>
           <manifest>
      <mainClass>${main-class}</mainClass>
           </manifest>
          </archive>
        </configuration>
      </plugin>
    </plugins>
  </build>

  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-report-plugin</artifactId>
        <version>3.5.2</version>
      </plugin>
    </plugins>
  </reporting>
</project>
```

As you can see, you can add plug-ins for use in different life-cycle phases. (This saves time and memory as only the needed plug-ins for the currently running lifecycle is loaded into memory.) For each, you can add properties to override the defaults and to use later (such as "`main-class`").

> Here we used JUnit 5 for testing. The popular older version was completely rewritten and is known as JUnit 5 but technically is called `org.junit.jupiter` (maybe because Jupiter is the 5th planet?). You will learn about JUnit and testing later in the course. The JUnit 4 dependency is:
>
> ```
>   <dependency>
>     <groupId>junit</groupId>
> ```

```
    <artifactId>junit</artifactId>
    <version>4.13.1</version>
    <scope>test</scope>
  </dependency>
```

In this example I did not use a *property* for the version, although I do in my own POMs. You must define that in the `properties` section, and I find it easier to have version numbers that may change all in one place near the top of the POM, so I don't have to hunt through the whole POM for them.

All Maven projects and artifacts have a POM associated with them. Each POM can add to or override the built-in global Maven POM. Your project's `pom.xml` file can override those settings. Finally, you can override any settings from the command line.

You can also create a `~/.m2/settings.xml` file. This file can include items you want in all projects, without needed to add them to each project's POM, such as developer (name, email, URL) info, additional repositories, and so on. See the reference for Maven settings for details.

One nice use of this file is to reduce typing on the command line. With a `settings.xml` file of:

```
   <settings

xmlns="http://maven.apache.org/SETTINGS/1.0.0"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"

xsi:schemaLocation="http://maven.apache.org/SETTING
S/1.0.0

https://maven.apache.org/xsd/settings-1.0.0.xsd">
       <pluginGroups>

<pluginGroup>com.github.spotbugs</pluginGroup>
       </pluginGroups>
   </settings>
```

You can use "`mvn spotbugs:gui`" instead of "`mvn com.github.spotbugs:spotbugs-maven-plugin:gui`"!

Maven supports the concepts of sub-projects. A large project may be composed of several semi-independent sub-projects. Each sub-project is in its own subfolder

under the parent project's folder. And of course, there is a `pom.xml` for the parent project, specifying the sub-projects.

> When specifying dependencies in a POM, you don't always need to specify a version number; one can be "inherited" from a parent POM. See the introduction to Maven's dependency mechanism for details.

Some editors (including Notepad++ with the "XML tools" plugin) can validate XML files if you point them to a DTD or XML Schema. Most IDEs can too. You can find one for the current (4.0.0) version of POM at /maven.apache.org/xsd/maven-4.0.0.xsd. So any typos are flagged as you go.

Project Phases

As mentioned earlier, Maven defines a series of project lifecycle phases. On the command line, you usually just list one of these; all preceding phases will be done in order first. (Unless something fails of course.) The default lifecycle phases are:

- `validate` — validate the project is correct and all necessary information is available
- `compile` — compile the source code of the project
- `test` — test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
- `package` — take the compiled code and package it in its distributable format, such as a Jar.
- `integration-test` — process and deploy the package if necessary into an environment where integration tests can be run
- `verify` — run any checks to verify the package is valid and meets quality criteria
- `install` — install the package into the local repository, for use as a dependency in other projects locally
- `deploy` — done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.

You can list more than one phase on the command line. For example:

```
mvn clean install site
```

First runs the `clean` phase (removing any prior artifacts from the target directories), then `install` (build and test everything), and finally the `site` phase, which generates a website often used to show project data, statistics, JUnit output, etc.

Goals and Plugins

The steps taken for a given lifecycle phase are called ***goals***. The specific goals run depend on which plugins you use.

Each Maven plugin contains the code to execute some goal. (A plugin may define several goals). For example, the standard plugin to compile Java source code has a `compile` goal that will call `javac` for you, automatically setting CLASSPATH appropriately. Naturally, you can customize the `javac` arguments in your POM or on the command line.

The goals of some plugin are ***bound*** to (associated with) one or more (or none) lifecycle phases. So by specifying a plugin in your POM, some goals are automatically bound to some phases. It is entirely possible to specify plugin goals on the command line to execute them (in the order listed). You can mix goals and lifecycle phases on the same command line.

The default Maven installation includes most of the plugins you need for Java development already. Thus you rarely need to specify plugins in your POM, but you may do so in order to override their default settings.

One useful standard plugin is help, which can tell you about your system or the details of any plugin (Try "`mvn help:help`" and "`mvn surefire:help -Ddetail=true`". (Note plugins usually have a `help` goal.)

> Any modern IDE can be configured to bundle and deploy Jar and War files. So why not use that? Here's a good answer from joergmmeier@arcor.de, posted on comp.lang.java.programmer on 7/25/14:
>
> Professionally run projects don't have people build from whatever IDE they happen to have installed at the moment. Instead, they **build and deploy in a reproducible and controlled environment**. Maven has a very clearly defined behavior: you can tell users/developers to do "`mvn package deploy`", instead of "Download Eclipse version 3.2 (but not the 64-bit version), go to Settings, Preference, tick setting A, uncheck setting B, then add a new server in the server view, select Tomcat (but only version 6!!!), download the patch for the web plugin, ...". Typically, with Maven the developer simply runs a command such as "`mvn clean compile package deploy:deploy`".

Using Maven from the Command Line

Maven is a command line tool. To create a new project is very simple: create a folder to contain your project and make that the current directory. Then you can either create the `pom.xml` file yourself or ask Maven to create a default one for you. Then you create your source code and JUnit tests, and add your PNG and other resources.

When ready, you run Maven with one or more *lifecycle phase* arguments to build and test your code. (Maven is also integrated into popular IDEs, but that won't be covered here.)

First up is to use the Maven plugin "`archetype`" to create a "Hello, World" project for us. You must run this in an empty directory, or Maven might fail (the lines ending with "^" are continued on the next). This is the interactive setup:

```
C:\Temp>mkdir mvn-projects
C:\Temp>cd mvn-projects
C:\Temp\mvn-projects>mvn archetype:generate ^
   -DgroupId=com.wpollock.hello -DartifactId=hello
...
```

(Feel free to use a different groupId value, e.g. "`cop2805C`" or "`example.com`". The groupId becomes the package name.)

At the prompt, hit enter or type "`maven-archetype-quickstart`" (without the quotes), then hit enter to accept all remaining defaults. You can type a filter word such as "`quickstart`" to reduce the number of items shown. If you know in advance which archetype you want, you can specify it on the command line and even turn off interactive mode (so all defaults are chosen):

```
C:\Temp\mvn-projects>mvn archetype:generate ^
-DgroupId=demo -DartifactId=demo ^
-Dversion=1.0.0-SNAPSHOT ^
-DarchetypeVersion=1.4 ^
-DarchetypeArtifactId=maven-archetype-quickstart ^
-DinteractiveMode=false
```

The resulting POM will almost certainly need a few tweaks: It uses JUnit 4, but you should change the version of that from 4.11 to the latest version, 4.13.1 or change to use Junit Jupiter (Junit 5). In addition, it specifies Java 7 in the `properties` section. You should replace the two lines there with "`<java.compiler.release>11</java.compiler.release>`".

> The *release* option replaces the *source* and *target* options, but sadly the current (2021) version of Maven (3.6.3) still defaults to an older version of the compiler plugin, one that doesn't recognize the new *release* option. Using the archetype version of 1.4 (or newer if available) fixes that issue, by specifying working versions of all the standard Maven plugins in the `pom.xml` file, thus overriding the broken, default versions. (IntelliJ IDEA does that for you.)

Aside from fixing your `pom.xml`, you may need to add some folders such as for "`resources`". The reason is that Maven cannot (currently) create empty

directories using an archetype. The `main/resources` directory is used to hold the resources used for your project such as images and icons, sound files, text files (say for configuration data or alternate language versions of strings), and so on. The `test/resources` directory is used for the same but when testing, so you can use special versions of resources if you want.

> Maven arranges for the contents of the `resources` directory to be accessed by the path "`/`". So "`.../main/resources/some.gif`" is referred to in your Java code as "`/some.gif`".

To build your project:

```
C:\Temp\mvn-projects>cd hello
C:\Temp\mvn-projects\hello>mvn install
...
```

In this example, we used the `generate` goal of the `archetype` plugin to create a project (including the `pom.xml` file). The `groupId` and the `artifactID` are the only required arguments to this goal, but as you see it is a good idea to specify some others.

(Note! If you do not override the version number, these archetypes may include a poor one such as version 1.0-SNAPSHOT. In general, always use semantic versioning so there are no surprises!)

> The first time your run this, Maven will need to download a zillion plugins and artifacts into your local repository, which takes time. The commands are much faster after that.

After creating the new project, we changed directory into the project's root (top-level) folder. This is where the `pom.xml` file is found. If you don't do this, Maven can't find the `pom.xml` and just won't work.

> Archetypes are useful, but a zip of a prototype project is just as useful.

In the last command, we ran Maven using a lifecycle phase argument of `install`. (While not required, it is common to run Maven with two (or more) phases in one command, such as "`mvn clean package site`".)

The required stuff is downloaded, new directories are created, the code is tested with JUnit, then built, and finally bundled into a (non-executable) Jar file. (Which is also copied to your local repository.)

> **Each downloaded artifact should be digitally signed by the artifact's author. By default, Maven doesn't check that these are valid.** Limit yourself to trusted repositories, and only those that support TLS (HTTPS).

To run the newly built app, type (all one line):

```
java -cp target/hello-1.0-SNAPSHOT.jar
com.wpollock.hello.App
```

(A transcript of this, with nearly all the download lines removed, can be found in the class resources. While the transcript was made on Linux, it is exactly the same on Windows. ***Show***.)

By using other available plugins, Maven can (if you add the required dependencies to your `pom.xml`) generate Java API doc comments, upload to a website, and even deploy your Jakarta EE apps to appropriate servers (there's one available as a plugin, for testing).

> The major headache with Maven is finding the plugins and their spotty documentation. To help address the problem, you can use their website to find plugins, and there's a `help:describe` plugin and goal, to help you with each standard plugin.

A more complete ("fancy") `pom.xml` for building an executable Jar file can be found in the class resources. The generated (default) `pom.xml` looks like this:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>

   <groupId>com.wpollock.hello</groupId>
   <artifactId>hello</artifactId>
   <version>1.0.0-SNAPSHOT</version>
   <packaging>jar</packaging>

   <name>hello</name>
   <url>https://wpollock.com/</url>

   <properties>
      <java.compiler.source>11</java.compiler.source>
      <java.compiler.target>11</java.compiler.target>
      <project.build.sourceEncoding>UTF-8
      </project.build.sourceEncoding>
   </properties>

   <dependencies>
      <dependency>
         <groupId>junit</groupId>
         <artifactId>junit</artifactId>
         <version>4.13.1</version>
```

```
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

You can (and should!) edit the POM to suit your needs. In the Maven resources section of our class web page is a modified POM file I created, along with a demo use. For example, it includes support for the exec plugin, so you can easily run your Maven project with "`mvn exec:java`".

With the POM modified to create runnable Jars, after "`mvn install`", you can run the Jar by double-clicking (if the program is GUI), or by:

```
java -jar target/hello-1.0-SNAPSHOT.jar
```

Note the modified POM in the demo changed the JUnit 3 tests to JUnit 4 tests.

> You can actually create a new archetype that will generate a JUnit 4 (or 5) stub class instead. In fact, you can generate your own prototype project, then a single Maven command with generate the new archetype from it that you (and your team) can use.

Here's the replacement `AppTest.java` file, for JUnit 4:

`src\test\java\com\wpollock\hello\AppTest.java`:

```
package com.wpollock.hello;

import static org.junit.Assert.*;
import org.junit.Test;

/**
 * Unit test for simple App.
 * @author Wayne Pollock, Tampa Florida USA
 */

public class AppTest {
    /** A through test! */
    @Test
    public void testApp() {
        assertTrue( true );
    }
}
```

Maven and Jenkins (the continuous integration server) are often used together. Once set up, you merely use your favorite IDE to work on your code, and when you want it built, do a commit. Jenkins will notice and issue the appropriate Maven (or other) commands for you, collecting the results, and alerting you and your team as desired.

You can also use Maven directly with IDEs. (*Homework: explore the Maven plugin(s) for your favorite IDE.*) This is probably the most popular way to use Maven. Maven provides uniform build system regardless of IDE and offers a repository of stuff you can easily reuse. For example, if you created a Maven project from the command line and would like to make it into an Eclipse project, run "`mvn eclipse:eclipse`" to generate the requires Eclipse project files. Then go to Eclipse and import an existing Maven project. (**Do NOT** use the standard "import existing project" option!)

> Eclipse and other IDEs may have older project settings saved even if you've updated the IDE. One of these is to use Java 5 for (at least) Maven projects; you should try to change those settings for existing projects. And change the defaults for new projects too.

Most IDEs provide easy ways to edit POM files and can add dependencies automatically for you. Even so, fiddling with dependencies can cause initial project setup to take hours. For Spring projects, you should look at using Spring Boot and its web tool Spring Initializr (yes that's how it's spelled). This cool tool is a web page where you click on checkboxes to say what you want in your project, and it will create a zip of a Maven (or Gradle) project for you with the `pom.xml` correctly filled out! Project configuration that used to take hours or even days now takes a few minutes only.

Maven and Java 9+

The default compiler version used by Maven's default compile plugin was 1.5, as of 2019. That means no later Java features will compile correctly. Secondly, if you installed Java 9 or newer, it no longer supports the option to compile for 1.5! Thus the compile phase will fail.

To fix, you need to add these lines in your `pom.xml` for Java 11:

```
<properties>
  <maven.compiler.source>11</maven.compiler.source>
  <maven.compiler.target>11</maven.compiler.target>
</properties>
```

Since Java 8, you need to drop the "1." part of the version, although it seems to work through Java 10 to use "1.10". Those four lines can go anywhere between the initial `<project>` tag and the final `</project>` tag, but not inside of any other tags. Just above the final `</project>` tag is fine. (Hopefully, a new version of that plugin will address that problem!)

Since Java 9, the old `-source` and `-target` Java options were replaced with a single option `-release`, so for newer JDK versions, use this:
`<maven.compiler.release>13</maven.compiler.release>`

instead of the two properties above. However, this won't work with the default version of the Maven compiler plugin (as of Maven 3.6.3).

Updating Maven Plugins

It seems weird but using "9" or higher in the above may not work even though we are running Java 9 or newer. This is because Maven uses various plugins that don't get updated automatically just because you've updated *your* JDK. The older versions of the Maven plugins (such as the Java compiler plugin) simply don't know about Java 9+ features and may not run or compile correctly with newer Java versions. Indeed, the Maven command itself does not get updated just because some plugins were updated.

Because of this, you sometimes must override the default plugin versions used by the `mvn` command and/or override the default configuration of some plugins. (At least until a newer version of Maven gets released.) IntelliJ IDEA does that for you when creating pom.xml, but otherwise you need to manage versions yourself.

You may be able to update your standard Maven plugins by deleting them from your local repo (the "`.m2`" folder in your HOME directory). The next time you build with Maven, all the plugins are downloaded afresh and usually you get the latest versions (unless you specified a version in your POM). But if Maven still specifies the old versions as the defaults, this won't do anything. The safe way is to specify working versions of plugins in your pom.xml file, or for all projects in the `~/.m2/settings.xml` file. (The 1.4 or newer version of the QuickStart archetype does this for you.)

> In Maven 2 you could use "`LATEST`" for the version in your POMs, but that "feature" was deprecated in Maven 3. (Some other features such as version ranges are deprecated but still work in Maven 3.)

I suggest you download and add to your POM the `org.codehaus.mojo:`**`versions-maven-plugin`** from Maven Central. It has many goals to help you manage and even update versions of plugins and dependencies (artifacts) in your projects; you can read about it from its home website or view a good tutorial from www.baeldung.com. Examples to view and then update versions:

```
mvn versions:display-dependency-updates
mvn versions:use-latest-releases
```

> Using such Maven plugins requires a valid `pom.xml`. This means the currently referenced versions need to be present. Thus, if you delete your `.m2` repo from your HOME directory, build the project before using versions plugin.

The versions plugin can also update properties containing version numbers. (Then use "`versions:update-properties`" instead.) This plugin finds property names by working back from the `<version>${some-name}</version>` part of the dependency. I suggest using some naming scheme for such properties, such as "*artifact-name*.version".

> There are many useful plugins like Versions. For example, to release project artifacts often requires a lot of repetitive steps. You can use the Maven "Release" Plugin, or the popular alternative "Unleash" Maven Plugin.

Using 3rd Party Software with Maven

It may not be obvious, but whenever you use a dependency tool such as Maven, you are automatically downloading software written by unknown parties. Just because some package was uploaded to Maven Central is not a guarantee of quality! (The same holds true for other repositories and languages.) Worse, such software often will depend on other packages (so 4th party software and beyond); these are called ***transitive dependencies***. It is unlikely any developer can do quality checks on all that. What can be done?

You can check the home website of each explicit dependency you have, to look for *red flags* (indicators of poor quality) and *green flags* (indicators of high quality).

Red flags include major but old bugs not fixed (look at the project's issue tracker for these) or no recent commits. If the software doesn't include a test suite, that's a red flag. If the project seems to have a single developer/maintainer, that's a red flag (such projects are likely done in some person's spare time). Download and compile their code with all important warnings and checks enabled. Run their test suite. Any issues are red flags. Inspect some of the code for style and design issues. Does it seem as if the code is high quality? Is the code well documented? Does it have a clear license (and does that license allow your intended use)?

Green flags include a large number of users (check the download/dependency statistics), through test suite, their issue tracker shows important issues are fixed quickly, commit log shows recent activity (so the software is actively maintained), and (if it is open source software) has the project earned a Core Infrastructure Initiative (CII) Best Practices badge (or at least are they well on their way to that)? An OSS project that has earned a CII Best Practices badge implements a number of best practices for sustainably developing secure software. (CII is now OpenSSF, the Open Source Security Foundation.) View the CII Badge Criteria for more information.

Check the software and its transitive dependencies for security issues from the U.S. National vulnerability Database (NVD). An easy way to do this is with the OWASP dependency checker. It can be run several ways but using it from Maven may be easiest for Java programmers:

```
<dependency>
  <groupId>org.owasp</groupId>
  <artifactId>dependency-check-maven</artifactId>
  <version>5.2.2</version>
  <type>maven-plugin</type>
</dependency>
```

Use it like this from the command line:

```
mvn org.owasp:dependency-check-maven:check
```

The first time can take over 10 minutes, since the whole NVD must be downloaded. Later runs only download any updates and should take a few seconds if run regularly. (See also the examples at jeremylong.github.io/DependencyCheck to include the check automatically as part of the build.)

Any serious security issues in the current versions are serious red flags.

> All projects are bound to have some red flags and some green flags. Some red flags may be simply an unfamiliar development process. For example, SQLite ships as a single 200,000 line C source code file. That's a red flag, but if you look deeper you will see they develop using many files that are automatically combined for deployment purposes.
>
> You need to decide whether or not to use the software in your project after evaluating all the flags. Considering your intended use, some flags may be more important than others for you.

A typical large-ish project can have dozens or even hundreds of dependencies. It may be impossible to validate them all in a reasonable time frame, but you should do some checking regardless.

Any dependency you select, consider using a wrapper class to isolate it from your code. The wrapper class should only expose the minimal functionality required by your software, not every feature provided by the dependency. If someday you need to update the dependency or replace it, you only need to make the change in a single place. Include the wrapper's functionality in your own unit test suite.

Finally, keep all software up to date (I use Maven's versions:display-dependency-updates plugin for this).

In-class program: Write a program that reads in a list of integers (until EOF), and then prints all values greater than the average.

Professional Ethics

This is an important topic. **Professional ethics matter**, and you need to understand them, before you find yourself in an uncomfortable position. (Show and discuss ethics resources.)

Personal ethics refers to the ethics that a person identifies with in respect to people and situations that they deal with in everyday life. Professional ethics refers to the ethics that a person must adhere to in respect of their interactions and business dealings in their professional life. In some cases, personal and professional ethics may clash and cause a moral conflict. For example [from iaa.govt.nz on 1/28/2020]

- A police officer may personally believe that a law that he is required to enforce is wrong. However, [generally a police officer] is required to obey all lawful and reasonable instructions to enforce that law unless there is good and sufficient cause to do otherwise.
- A doctor may not personally believe that the course of medical treatment chosen by a patient is the right one. However, [generally a medical professional] must respect the rights, autonomy and freedom of choice of the patient.

This topic is more important than ever, for many reasons. For one, modern software development cycles are very short, resulting in the near elimination of management, financial, and legal reviews. Consider the software for Facebook, Twitter, Amazon, eBay, etc. Small teams, and sometimes individual programmers, develop and deploy software directly. Such software isn't just used locally. This is even truer for cell phone apps. **What is the developer's ethical responsibility for understanding and respecting the cultural, regulatory, and legal differences around the globe?** In the past, the developer could rely on management review of code to protect them from such considerations, but no longer.

Software impacts the public in many ways. It is not sufficient to rely on personal ethics to decide what to do, and what not to do; this is the subject of professional ethics. On-line ad systems are the source of much of the software industry's revenue (and thus your salary). They can have many negative effects, including isolation (by only suggesting things in which you've already shown interest, sometimes called a *filter bubble*) and racial bias. The same algorithms can affect political campaigns.

Worse, the machine learning techniques are not designed to show how they arrive at a decision, say to refuse cheap health care, or lower your credit rating. This can rob people adversely affected of any hope of due process. (This happened to me when I tried to purchase a car without a loan. The "computer" decided the amount of the check was outside my normal spending habits, and although I have sufficient funds to cover the check, it was rejected.)

Please make sure you spend some time browsing and reading some of the on-line professional ethics resources found on our class web page, and feel free to discuss any of the stories you find there in class.

Look at what happened when Josh Bloch copied nine lines of code from Sun/Oracle's Java implementation into Android. The code in question was originally written by Bloch himself while he worked at Sun, and then reused by him when he moved to Google. But it was still considered a breach of copyright during a multi-billion dollar law suit.

The code in question:

```
private static void rangeCheck(int arrayLen, int fromIndex, int toIndex) {
    if (fromIndex > toIndex)
        throw new IllegalArgumentException("fromIndex(" + fromIndex +
            ") > toIndex(" + toIndex+")");
    if (fromIndex < 0)
        throw new ArrayIndexOutOfBoundsException(fromIndex);
    if (toIndex > arrayLen)
        throw new ArrayIndexOutOfBoundsException(toIndex);
}
```

Software Licensing and Intellectual Property Rights

Anytime you use software you didn't write yourself you are subject to the terms of the license for that software. This includes applications, libraries, plug-ins, graphics and other media, database components, documentation. Some applications are bundled with libraries and thus there could be several licenses that apply.

A license provides you with the legal right to use some product, within certain specified constraints. This is known as the ***License Grant***. The grant clause of a license states the permissible uses of the software or service, including when and how you can distribute the software.

Permissible uses clauses involve limits on the site or location, equipment, network, third party data, outsourcing entities such as service bureaus, and parties such as successors and corporate subsidiaries or affiliates.

Software licensing is a complex topic that involves intellectual property and copyright, trademark, trade secrets, and contract law, including (in the U.S.) the *Uniform Commercial Code* (UCC), international law (e.g., if the Internet is used), *Uniform Computer Information Transaction Act* (UCITA), and the *Digital Millennium Copyright Act* (DMCA). Even patents!

Licenses that accompany commercial products are written by lawyers to provide maximum control and minimum liability to the company that makes the product. Often it takes your own lawyer to interpret the license terms for you. In many cases, these are badly written or not legally enforceable (this may vary from state to state). Also in many cases, you can negotiate the terms with the company, although they usually don't make this easy. Software is generally

licensed rather than sold, in order to better protect the vendor's intellectual property right. The license often prohibits reverse engineering, modifying, disclosing, and transferring the software.

Some licenses grant intrusive rights to the licensor, such as the right to examine your hard drive, install extra software (spyware), the right to monitor your use of the software or service, or the right to sell your personal information.

If you plan to depend on this product, it is essential to understand the license. Many companies don't allow you to make any backups of their software, forcing you to exclude that from your normal backup procedures. In some cases, the software is allowed on a single system, and if you upgrade your hardware you may not be allowed to use that software without purchasing another license.

If the company has been sold or is no longer operational, or has simply abandoned some product, it may be impossible to renew or purchase a license, forcing you to abandon that product. In some cases, this risk can be mitigated by the use of ***software escrow agents***, who (for a price) will keep a copy of the source code for some product, and will release it under certain conditions (such as the company that produced the software is out of business). For example, NCC Group provides software escrow.

Disaster recovery issues must be addressed in the context of the need to use the software in a remote location while protecting the vendor's intellectual property. Computers may be physically changed with software compatible upgrades or exchanges, due to upgrade needs or disasters.

Besides understanding licensing from the point of view of the purchaser, you may need to **license your own (or your organization's) work**. While the license for any commercial product should be developed only with the advice of a competent attorney, there are some **standard licenses** you can use or adapt for your products. Most of these were developed for *open source* products; some of these are discussed below.

Types of Licenses

With a **Dual-use License,** the user can choose the least restrictive or otherwise most appropriate license. This is popular for software released in a commercial version and a freely available version (available "as-is").

You should decide whether or not your users must agree to the terms before being able to use the services offered or downloaded software. Such licenses are known as ***Click-through*** or ***click-wrap agreements***, but technically are called ***end user license agreements*** or ***EULAs***.

Often employees of some company must agree that various inventions and ideas developed by them and related to the business of the company are deemed owned by the company (***Invention Assignment Agreements***), so you can't just develop

some wonder product and then sell or even give it away; it belongs to the company.

A ***Site License*** is sometimes called a *seat license*, and may limit software use to particular computers. The identification of these computers can be a tricky issue. It is a widely misused term; it is often misapplied to a licensing agreement that allows software to be run on a large number of CPUs. It actually refers to an unlimited number of software licenses purchased and sometimes distributed at no charge to affiliates. There are very few true site licenses these days, since they tend to be prohibitively expensive.

What does a "seat" or site actually mean? Simultaneous users? Is it better for the vendor to have a sliding scale fee structure based on the size of a licensee? Concurrent use must also be considered regarding work shifts (several people using one computer), use on laptops, and home use (one person using several computers).

Volume Purchase Agreements or ***Software Volume License Agreements*** are formal agreements between you and a software manufacturer or vendor. A purchasing source, order process, and pricing limits are negotiated and put in a contract with specific terms and renewal conditions. Volume purchase agreements are rarely mandatory (i.e., end users can buy elsewhere), but generally offer very advantageous pricing. (Microsoft and some other vendors offer these to HCC students.)

Exclusive versus ***Nonexclusive*** licenses: Mass-marketed (***shrink-wrapped***) and ***click-wrapped*** (for downloaded software) software is generally marketed on nonexclusive basis. This means the vendor can sell the product or service to others besides you. However, with software developed on a custom or *exclusive* basis, the user/licensee wants to gain an exclusive license (to maintain an edge over competitors). Because the vendor may want to market the software more widely, these licenses can cost significantly more. (e.g., custom web portal, search engine, database). Either way, the license granted limits the permissible uses.

> "Nothing seems to prevent [companies] from attempting to prohibit certain conduct in their documentation, even if the prohibition is unenforceable. By way of example, DVDs marked 'for home use' or 'non-commercial private use only' are not legally restricted as such. Unless there is a contract, those representations are not true.
>
> "A given use, e.g., classroom showing, is permitted or infringing depending on copyright law, not what is printed on the packaging—unless there is a contract including that restriction. Rights holders still print such claims on the packaging, however. They might as well print that infringers are obligated to forfeit their first-born child." — Matt Schruers

Network Uses: A site license must consider network issues since software/data on a network at a given site may be processed elsewhere. The implications of how using particular software over the internet must be considered. Software that runs on servers may well be utilized in an internet context since so-called "n-tier" (enterprise) computing simply runs software components on several servers (often including web and database servers) as part of a single application.

Transferability Restrictions: As well as use by subsidiaries and affiliates, the issue of what happens if the licensee, or even the licensor, is sold in some manner must be addressed.

Understanding and Evaluating Free/Libre Open Source Software (FOSS)

"*Free as in beer*" means you don't need to pay to use the software. "*Free as in freedom*" means there are few restrictions on your use of the software, e.g. include it with your commercial, proprietary product. To indicate something free in both senses, the term FLOSS was used; today, FOSS is preferred by most.

> Market research firms reportedly estimate that for 2014-5, over 67% of all active services worldwide are Linux (or Unix) based, with some estimates over 95%. Netcraft (Internet research firm) reports the FOSS Apache web server at 48% of the market (with MS server at less than 30%, Nginx at 14%, and all others under 3% each). Other popular open source produces include Firefox (and other Mozilla.org projects), FreeBSD Unix, PostgreSQL RDBMS, SugerCMS, and thousands of others. Most cell phones, set-top boxes, gaming consoles, etc., all use open source for part of the system. Even proprietary software is often developed using OSS compilers, tools, and IDEs.

Before committing to a FOSS project, you need to understand that not all FOSS projects are successful. If the software you decide to depend upon is not supported you will not be happy with the additional expense of hiring programmers to re-write it, or to replace it.

Your organization is thinking of using (and depending on) some FOSS. If that FOSS fails your organization's products will fail. So how do you tell if a FOSS project is healthy or a bad risk?

Projects that have an element of intellectual engagement (i.e., are a hot topic) attract more and better talent than a project that offers a useful project or fulfills a community obligation.

Most healthy FOSS projects have one or two people that founded the project (say by creating a preliminary version then taking the project public), two to ten additional "core" programmers (those with access to commit changes to the repository). A few exceptional projects have many more core programmers

(Debian, Linux, Apache), but in general too many core programmers means that a lot of time is wasted synchronizing their efforts.

Part of this core is the current project leader, who maintains a "vision" and thus can resolve disputes. The leader often also coordinates official releases (or versions) of the project. Initially the founder is the leader, but it is a healthy sign if the leadership has successfully transitioned to a new project leader once the project matures.

Surrounding the core programmers are a group of ***active users*****. These are those who participate in early testing, who communicate ideas and report bugs directly with core programmers, but who don't have commit access to the repository**. Over time, some active users become core programmers, but FOSS projects rarely if ever have formal management that determines who gets commit rights. If you gain the respect of the project leader, founder, and other core programmers, you may be granted such access if you request it. Active users contribute to IRC, mailing list, and newsgroup (forum) discussions, write how-to documents and user guides, maintain a wiki site, etc.

> You can also check the mailing list for core developers, if available, and see how they all work together...or not.

Active users serve another vital role: they provide support to the vast hordes of ***passive users***, those who just download the software and need questions answered. It is not a good sign if the core users do this much, they quickly get burned-out and can leave a project stranded.

www.OpenHub.net (formerly ohloh.net) is a site that provides reviews and metrics for many/most open source projects (Demo Apache: select *compare projects*, show activity, contributors, and reviews). This is a good start to evaluation of some project! Note that GitHub and Sorceforge.net provide some of that data as well. Another interesting site is chaoss.community, which works on developing metrics and analytics for FOSS projects.

Note that if your organization chooses to make some software open source, you should examine several successful (and similar sized) FOSS projects, to see how they are organized and run.

It may not be obvious, but **whenever you use a dependency management tool such as Maven or Gradle, you are automatically downloading software written by unknown parties.** Just because some package was uploaded to a popular repo is not a guarantee of quality! Worse, such 3rd party software often will depend on other packages (so 4th party software and beyond); these are called ***transitive dependencies***. It is unlikely you'll have sufficient time to do quality checks on all that. What can be done?

You can check the home website of each explicit dependency you have, to look for red flags (indicators of poor quality) and green flags (indicators of high quality). (See the previous discussion of red and green flags, above.)

Check the software and its transitive dependencies for security issues from the U.S. National vulnerability Database (NVD). An easy way to do this is with the OWASP dependency checker, discussed previously.

All projects are bound to have some red flags and some green flags. You need to decide whether or not to use the software in your projects after evaluating all the flags. Considering your intended use, some flags may be more important than others for you.

> All software, not just FOSS, should be evaluated prior to adoption. One way to do this is simply using Google with the right keywords, to see what issues people have had in the past. Some good choices include (in addition to the name of the software): *crash, won't compile, won't build, won't start, slow*, etc. (Keep in mind there will be some hits for these searches, even for excellent quality software.)

Open Source Licenses — FOSS

Open Source licenses (FOSS means *free, open source software*; this used to be called FLOSS) vary by how restrictive they are. Some are incompatible with others. Such licenses are evaluated by the ***Open Source Initiative*** (OSI) at OpenSource.org, to determine if they are really open source or not.

> There is something like 80 licenses approved by the open source institute and there is an ongoing effort trying to reduce the number of these licenses. But you are not required to pick any of these for your FOSS. You can take any of these and make variations, giving rise to more than 1,000 known variations on licenses.

The European Union has created their own open source license, the European Union Public License (EUPL). It is compatible with most OSI licenses (See the EUPL compatibility matrix).

Some of the most significant and common of these open source licenses are the first four are ranked from least restrictive to most restrictive):

- **The MIT Consortium (X11) License**
  This just says you can use the software any way you wish, as long as you don't sue the authors over it. **This is the least restrictive license** (of those listed here).
- **The BSD License**
  Almost as liberal as the MIT license, this license says you can do what you want with the code, provided you don't remove the copyright notes or

disclaimer, both with source code and with binaries. The copyright and disclaimer simply ask for credit of the work and disclaiming any responsibility for it. You can't use the BSD name to endorse your product.

- **The Apache Software Foundation License**
  Similar to the BSD license, except extra clauses are added to ensure you don't use *Apache* to endorse or promote your product without prior permission.
- **The Gnu Public License (GPL)**
  This one is more complicated and restrictive than the others mentioned above and is currently the focus of on-going legal and political battles. This says you can use GPL'd coded in your products, only if you release your products under the GPL as well. This is sometimes referred to (by distracters) as the *viral* property of the license. Note there are two incompatible versions of this license, GPLv2 and GPLv3.
- **The Lesser GPL (LGPL)**
  This license allows your products to link to GPL'd libraries, without forcing your code to be released under the GPL. The LGLP used to cover libraries or other modules used with other software, can also have a "viral" property depending on how it is incorporated into your (proprietary) software. If statically linked, the whole software is covered by the GPL license. If dynamically linked instead, it is not covered.
- **The Eclipse Public License (EPL)**
  This is a revised version of the CPL, an IBM license (a section about patent litigation was removed). To reduce the number of open source licenses, IBM and Eclipse Foundation agreed upon using solely the Eclipse Public License in the future. The OSI now shows the CPL as deprecated and superseded by EPL.
- **The Microsoft Public License**
  Similar to the BSD license, this was approved by the OSI in 2007. Many of the projects hosted at the MS site codeplex.com use this license.
- **The Microsoft Community Promise -** www.microsoft.com/.../community-promise (formerly at www.microsoft.com/interop/)
  This is a legally binding commitment, through which Microsoft pledges not to assert its patents against others who implement certain Microsoft standards and technologies. It is generally compatible with open source licenses and philosophies. In 2009, it was applied to ECMA 334 and 335, which define C#. This means Mono and other non-MS implementations of C# need never fear a patent infringement lawsuit.

There are so many licenses out there, many incompatible, that is has become a real problem to track all the licenses that apply to a given software product (that uses many FOSS parts). Black Duck Software sells a product that can identify all the licenses that apply and determine any incompatibilities. (They were recently awarded a software patent on their method, but it is being contested.)

ChooseALicense.com helps users select an appropriate FOSS license through a web interface. A similar site from the EU is license wizard. tldrlegal.com explains various licenses in plain English.

Documentation is covered under licenses. But there is a problem with these. You might be able — technically — to remix FOSS content from Wikipedia, Flickr, YouTube, etc. But can you remix it legally? Will the licenses for that "free" content permit that free content to be remixed?

The surprising answer is probably not. Even if all the creative work you want to remix is licensed under some FOSS ("copyleft") license, because those are different licenses, you can't take creative work from one and remix it in another. Such a remix is defined as a *derivative work*, and few FOSS licenses allow derivative work to be under any other license (for fear of having FOSS works remade into proprietary works.)

Artistic works are also covered by licenses. As with software, these can vary. There are a number that have embraced FOSS principles, such as from creative commons or the "free art license" from artlibre.org/license/lal/en. Artistic works include fonts. These are often covered under the Open Font License (OFL) from scripts.sil.org/ofl.

License Incompatibility

Just because two licenses are both approved as open source licenses, that does not mean you can freely combine software licensed under them. The licenses in question must be *compatible*. The FSF defines this as:

"In order to combine two programs (or substantial parts of them) into a larger work, you need to have permission to use both programs in this way. If the two programs' licenses permit this, they are compatible. If there is no way to satisfy both licenses at once, they are incompatible."

As a concrete example, the ZFS filesystem is licensed under the CDDLv1 license, and Linux under the GPLv2 license. These licenses are *not* compatible, making it a violation to ship a compiled Linux kernel containing ZFS. Canonical thought they could work around that by keeping ZFS in a separate file from the Ubuntu main kernel, as a loadable kernel module ("zfs.ko"). But the software conservatory and others disagree and I don't think Canonical will risk it. It *is* permissible to ship

source code, allowing end users to compile the code, but that is not a feasible option to most companies.

Violations of open source licenses are often detected by competitors or any user. Typically, the violator is convinced to rectify the violation and obey the license terms. Organizations such as the Free Software Foundation, the Software Freedom Conservancy, and the Software Freedom Law Center will do that for you if you report the violation to them, turning to legal measures as a last resort.

Ethical Licenses

The JSHint software had this license since 2011: "The Software shall be used for Good, not Evil." It took seven years for the project lead to change this restrictive and vague license to a standard one, but other those years, they lost their place as the top JavaScript linter to arguably lesser programs, but which had better licenses.

There are several similar, so-called ethical licenses, such as the Hippocratic License. These licenses prohibit people from using the licensed software if they "actively and knowingly endanger, harm, or otherwise threaten the physical, mental, economic, or general well-being of underprivileged individuals or groups."

The problem with such licenses is ambiguity. The licensor might not know exactly what is meant by "evil". The term is not defined by most legal systems in the world, including in the U.S. Legally-conscious objectors to software with such a license are simply refusing to enter into an ambiguous contract.

Avoid these licenses. Whatever your political opinions or your ethics, the best possible chance for your open source project to become widely used is to use a standard open source license.

Copyrights and Patents

A ***copyright*** is a government granted protection to the owner/creator of some content (written works, art, music, software, etc.) A copyright applies to the expression of an idea, not the idea itself. See the Copyright Clearance Center's educational resources for more information.

> A *copyleft* is a play on the word copyright, copyleft was first described by Richard Stallman (Free Software Foundation), in 1985. It describes the right of people to use, modify, and redistribute content. (Copyleft uses copyright law to enforce this.) These are the sorts of terms included in FOSS licenses.

A ***patent*** is the exclusive rights granted by some government to a holder on any use of the ideas covered in the patent. A patent typically lasts for 20 years (since 1995, when the U.S. agreed to the WTO's TRIPS agreements), and may be extended under some circumstances. Companies can license their patents for *royalties* (money); IBM typically earns $2 billion a year from their portfolio of patents. (Some companies sell no products, and just hold a portfolio of patents.) Patents

need to be filed for in every country; a U.S. patent won't stop someone in Europe from using the ideas covered by the patent.

Open Source versus Open Standards

FOSS (or just OSS) may or may not be the best choice for key technology for an organization. But **without a doubt, *open standards* are always the best choice.** For example, you might choose some proprietary email client, but most likely, you would be upset if it didn't support SMTP or POP. Good sources of open standards are the RFCs for the Internet, ISO standards, ANSI standards (ANSI is part of ISO), W3C (web tech) standards, khronos.org standards (for media hardware), Oasis-approved standards, and others.

There are rare exceptions, of course. Sometimes a proprietary standard is just better than the open alternatives. For example, some people argue that H.264 is superior to the VP8 media standard, even though it is not free. Another example is Microsoft's ActiveSync protocol.)

FOSS and License Compliance

Most FOSS software does require the corporate user, who often tweaks or patches the software to include in their own products and services, to obey certain restrictions on the distribution and sale of such products and services. A company must ensure compliance with the terms of all such software (and non-FOSS software too), usually by periodic audits and a set of management processes and policies known as ***IT governance***.

Interest in IT governance is partly due to compliance initiatives, for instance Sarbanes-Oxley in the USA and Basel II in Europe, as well as the acknowledgment that IT projects can easily get out of control and profoundly affect the performance of an organization.

FOSS compliance means that users of FOSS must observe all the copyright notices and satisfy all the license obligations for the FOSS they use in commercial products. This is more complex than expected, since your organization may also want to protect their intellectual property from unintended disclosure. Note that every change to your source code for products and services may run afoul of changed licensing terms, or new components with different licenses. It is suggested that the trouble-ticking system include an item for license compliance checks. Note that any changes will require another (sub-)ticket, to update the end-user documentation as well.

An essential member of an **open source review board (OSRB)** is the organization's legal counsel. They provide numerous services to ensure a company's products comply with open source copyright and licenses. They provide approval of the use of FOSS in products, advice on licensing conflicts, advice on IP issues associated with the use of FOSS, and review and approve updates to end-user documentation. As product and service updates affect

licensing compliance, the legal counsel should establish and help maintain a FOSS policy and process across products and product teams. This should include directions for how to handle compliance inquiries sent to the company in relation to FOSS compliance. To make software development simpler/cheaper, the OSRB should have some resources kept up to date, including a pre-approved license list, and a license compatibility chart.

The SPDX standard is an effort to help with license compliance. SPDX is an open standard for communicating software bill of material (SBoM) information, including components, licenses, copyrights, and security references. SPDX reduces redundant work by providing a common format for this information and various tools to work with it. They make a standard format for license information to be included in any package and support a variety of tools that can read that info and produce compliance reports. The list of licenses includes standard (short) identifiers and URLs for each; use in your `pom.xml` file or in a file such as "`license.spdx`" your Git repo. A short file describing your package's license is common, but you can include a lot more information such as all your dependencies' licenses too. SPDK is now (2023) an ISO standard.

The `cargo` build tool (for Rust, not Java) can include a check (with a plugin) of all the licenses in all dependencies against a list you provide in a configuration file, using the SPDX license identifiers. Violations are reported as build errors. I love that tool! While there are some Maven plugins for this, they don't seem as easy to use.

Software Bill of Materials (SBoM) [From bell-sw.com]

A software bill of materials (SBoM, SBOM, or sometimes just BOM) is an inventory of all dependencies used to build a software product. It is similar to a traditional bill of materials listing all raw materials, assemblies, and components utilized for product manufacture. An SBOM includes open-source and proprietary software used.

Such an inventory of used software in your project can be very useful to check for reported security vulnerabilities or check for changed license terms. New versions can be easily found too. For users of software, an included SBoM can help them make informed purchasing decisions.

There are legal reasons as well. In 2021, the U.S. federal government requires an SBoM for any software used in many federal agencies. Over time, there will most likely be others who require this.

You can create an SBOM at any time of the software development cycle, but at build time is recommended by the NTIA. Your SBOM must be in an approved standard format, one of Software Package Data eXchange,

> CycloneDX (my choice), or Software Identification (SWID) tags defined by the ISO/IEC 19770-2:2015 standard.
>
> There are many free and non-free tools available to generate SBOMs in one or another approved format. I currently like CycloneDX, which can be run through a Maven or Gradle plugin too. A command line tool can be used in a CI/CD pipeline (so generation is automatic when you build your software).

Created in 2007 by HP, FOSSology is an open source FOSS license compliance tool. You can use it to run license, copyright and export control scans from the command line. FOSSology includes a database and GUI to make compliance checks simple. The results can be saved as an SPDX or text file, with the copyrights notices from your software. Another popular tool is FOSSA.

Some available resources for those responsible for FOSS compliance include a training course and license compliance checking tools sponsored by the Linux Foundation. They are also putting together a set of best practices (see Openchain).

> All this stuff makes me long for the days of programming in the 1970s and 1980s: you just wrote code, and management would take care of license and any legal or regulatory issues, quality assurance would take care of all compliance issues, marketing would review all documentation, security was not an issue, and use of 3rd party dependencies was unheard of. The job of a software developer was much simpler then!

In-class program: Write a program using console input (`Scanner`) that inputs the length, width, and height of a package (in inches). The program should display "`oversize`" if the length plus girth (diameter) is more than 100 inches, "`standard`" otherwise. (Note: as this calculation is too complex for the average traveler, or airline employee for that matter, most airlines just calculate length+width+height and compare to 80 inches, give or take.) (Time-limit: 10 minutes.)

Lecture 2 — Source Code Control Systems (Versioning Control Systems)

Two factors (among others) can affect implementation. One is the ability to pull up previous versions of classes/packages (in general, software modules) and to show the differences between two versions. Another is the ability to work in a group on the same code. Both factors can be met using ***Source Code Control Systems***, also called ***Revision Control Systems, Version Control Systems, Content Management Systems, Source Code Management Systems, Software Configuration Management Systems***, and no doubt many other marketing buzz-terms. I will use VCS (for version control system) here.

VCS systems also provide change tracking and logging features, often used with bug tracking (aka issue tracking, or simply *ticketing*) systems.

VCS systems are useful because they can provide an accurate audit log of a project's development and evolution, often needed for compliance auditing and management reports. Modern VCS systems allow many developers to work on the same code at the same time, without corrupting the code. A third use is to fix *regression* bugs in large codebases. Suppose you discover that some change made in the past caused a serious bug. How do you locate the lines of code containing the bug, when you might have thousands or millions of lines to look through? Also, when was the bug introduced, do you have to patch old versions or just the current one? With a good VCS, a developer can easily find the exact change that introduced the bug, and thus only have a small amount of code to examine (also, you know exactly when the bug was introduced).

The first of these was SCCS, developed for Unix in the 1970s. Even today, SCCS is mandated by POSIX (the standard for Unix like systems), and still used. Development of both SCCS and RCS stopped long ago. However, recently Jörg Schilling ("Schily") revived SCCS as a FOSS project at sccs.sourceforge.net, adding additional features and performance improvements.

An incremental improvement to the original SCCS that proved popular is RCS. Even today, when all you need is per-file history, RCS or the new SCCS work great and are simple to understand and use: any developer can *check out* a read-only copy of the code to look at. To make changes, you check out the code for editing; this locks the file so no one else can check it out for editing. When done, you check in the modified version. These systems work by storing the original version, then a *diff* listing (aka a *patch*, described in detail below) for each subsequent version. (*Demo diff on two versions of some .java file.*)

> A limited version of `diff` is available from the DOS command line prompt, named `fc` for *file compare*. Don't use that for patches. Use the `diff` that comes with Git for Windows, or the Windows' version of `diff`: `wsl diff`.

Eventually people wanted to work on shared projects across the Internet. SCCS and RCS only work on one machine, and don't help people collaborate (due to locking). Newer tools were needed that were network-capable.

One of the most popular was CVS, which allows one to set up a single code ***repository*** that all developers can access across the Internet. Each developer ***checks out*** a complete copy of the code, works on it, and periodically ***checks in*** any changes they have made. Unlike the earlier systems, the files aren't locked; multiple users can work simultaneously. Usually that's not a problem, as each developer typically works on different files. If two developers make changes to the same file, the first one to check in their work has no problems, but the second one usually will find they can't simply check in their changes because of conflicting changes. That check in process is actually a ***merge*** operation, and often must be done by a human.

> The term *repository* (or *repo*) is overused is programming. Dependency management build systems (such as Maven) have repositories of artifacts (generally jar files for Java). Version control systems (such as Git) have source code repositories. There are other types too!

Despite its popularity, CVS has some problems. Newer systems with many of those issues addressed include the popular ***Subversion*** ("SVN"). Unlike CVS and RCS, SVN tracks changes to whole projects, not just per file. That means if you rename or move a file, or change its permissions, SVN can track that. IDEs have plug-ins to support a number of VCS systems. Today, Subversion is the most popular centralized VCS (CVCS). However, ***distributed versioning control systems* (DVCSs) have displaced CVCSs**. Today it is expected for developers to be familiar with Git (although there are other DVCSs such as Mercurial).

> Subversion is an improvement over CVS in that it supports *atomic commits* (all changed files are checked in at once). It also versions the entire project, not each file individually.

Distributed Versus Centralized VCS

For working privately on projects, any VCS is good enough. I use the simple RCS when all I need is per-file history (such as configuration files on a server). But most software development projects have more than one developer, who must all work together as a team. Originally this meant using a "centralized" VCS such as CVS or Subversion.

In a *centralized VCS,* there's only one repository. Only those with "submit" (or "commit") access can make changes, saving their work frequently as "checkpoints" or "commits". Other developers must submit patches (diff listings showing changes to one or more files) to someone with such rights.

It's awkward to work privately on a shared project with centralized VCS, because it means building up a huge patch (set of changes) without any saving any checkpoints (in the central repo), then submitting the whole thing at once as a surprise on the other developers. This behavior is referred to as "dropping a bomb". But if everyone submits frequently, buggy code gets used by everyone. You can use multiple "branches", say one for untested code and one for tested code. But with CVCSs, that's a lot of extra work to maintain (since early VCS didn't support branches directly) and to merge in everyone's changes. On the other hand, if you don't do that then developers will work for days or weeks without checking in anything, losing some of the benefits of version control.

In a **distributed version control system** (DVCS), every user starts with a fully legitimate "fork" (or copy, or *clone*) of the *repository* of the project. The one "official" tree (a repo is a tree of code) is purely a matter of social or legal convention. With a DVCS, all users have their own local repository, complete with project history, branches, etc. Switching to an alternate branch, examining file history, and even committing changes, are all local operations. So, while working on your code, you can check it in locally as often as you wish. You can even make "experimental" branches to try out ideas. None of your local change will affect other developers.

Once your code is tested, you then ***push*** the changes you made to the central repository. (Or you can push your changes to a co-worker's repository, so they can test your code.) If you don't have permission to make changes, you can create a ***pull request***, which is usually an email containing the changes as a set of patches, sent to some developer who does have permission to make changes. (See below for details.)

A DVCS is a great fit when using *agile* development methods or several semi-independent coding teams. Coders can implement some code, change it, toss it out, and implement it again, all with using local commits. Team members can push (or pull) changes to each other's repository; this doesn't affect the main repository or other developers. And if the modules that each team is working on are loosely coupled, the risk of conflicts is low. (That means automatic merging will usually work.)

Individual DVCS repositories exchange information via ***push*** and ***pull*** operations. A *push* transfers some local information to a remote repository, and a *pull* copies information from a remote repo to the local repo. (Since both repositories may have some local history that the other does not have yet, you can do both operations to *synchronize* the two repos.) A DVCS system makes it easy for repositories to describe unambiguously the history they have (or the history they are requesting).

A DVCS can be used in a similar manner to a traditional CVCS if desired, with a central "authoritative" repository with which each developer synchronizes. For large projects, it is possible to have a hierarchy of repositories with maintainers for each repository accepting changes from downstream developers and then forwarding them upstream. Basically, DVCS allows traditional workflows as well as newer ones. An organization only needs to make a policy as to what workflow should be used.

DVCSs also allow developers to share work with each other directly. For example, two developers working on a new feature could work on a common branch and share work with each other independent of any "authoritative" server. Once the work is stable, it can be pushed to a public ("more authoritative") repository for a larger audience.

Because there is no single authoritative repository, the terms client and server don't necessarily apply. When talking about two repositories, they are typically referred to as the *local* and *remote* repositories, rather than client and server.

CVCS are still used. It can be hard to make a business case for DVCS to managers used to the older workflow model:

- Many business scenarios require a single authoritative repository, and an official "keeper of the flame" (person who approves commits). Branches have to be strictly decreed and controlled, as a matter of corporate policy.
- If a DVCS were used in place of a CVCS, the manner in which it would have to be used would offer few benefits over CVCS. Changing the VCS procedures needs a large benefit, because there is a cost of changing the current system.
- Some managers don't see local commits as being a good thing; to them, it sounds like developers wasting time or doing unsupervised work. To many managers, the DVCS features are interpreted as leading to lack of control and of visibility.
- The longer you keep your changes local (even if "locally committed"), the higher is the risk of a conflict with other developers' *changesets*. Conflicts waste time and money to resolve, so the benefit of local commits is mostly for remote developers, say for a FOSS project.
- History is super-important in these organizations — it is how they locate scapegoats. (Only partially kidding; the "scapegoat" is typically the one who is in the best position to fix a problem, so identifying him or her without having to call a meeting to ask every developer is worth money.)

Mostly, these concerns are groundless.

DVCSs include **Git** (most popular but hardest to learn), **Bazaar** (the one I learned first, and used to maintain my website, before I switched to Git) and **Mercurial** (also supported on `google.com`).

> Microsoft used their own *SourceDepot* DVCS for Windows code (MS uses many different VCSes for different projects including their own Team Foundation Version Control, but they are migrating (5/2017) to Git. Their Windows code base is so huge they needed to make changes to Git to support it, which should eventually find their way into the Git mainstream. (Read the interesting story at Ars Technica.)

There are a huge number of FOSS projects in VCS repos publicly available on the Internet. However, it can be hard to discover them unless you already know about them. There are some standard repository servers anyone can use for free on the Internet. Repositories on such servers are usually "public"; to create private repositories on these sites, you often need to pay.

For Git DVCS projects, you can publish on **`github.com`** (the world's largest code repository), `Launchpad.net`, `BitBucket.org`, ~~`code.google.com`~~, `freecode.com`, SourceForge.net, FreeCode, and others. The maintainer of the "official" repository will frequently *merge* the changes from the other developers' repos (using a *pull* operation).

> VCS is not the only way developers can collaborate on projects today. Sharing and communicating using Facebook, Twitter, Skype, blogs, wikis, and even SMS are becoming common.

Google Code was one of the best of the free options. Not only could you create projects, control who could commit to them, and use various VCSs, it supported web pages where you could make stuff available for download (such as installers for your project's releases). Most of the others have no web interface, making you host your projects' files on some other site. GitHub has recently added support for hosting releases, and Google dropped their Google Code site. (Note, Microsoft acquired GitHub in 2018.)

> Recent versions of OpenJDK's Java SE and the JDK are equivalent to Oracle's version, and the development now takes place on GitHub at github.com/openjdk.

Using Git

First step is to install Git. It may be installed already if you have Linux or a Mac computer. If you have Windows, you can install Git (for windows) that provides a command line interface (Linux-like) and a simple GUI interface. Or if you have Windows 10 or newer, you can enable Linux (WSL, or Windows Subsystem for Linux) and use `git` from there. Git is well-integrated into all popular IDEs (and

probably pre-installed). If you want to use Git without an IDE and prefer Windows Explorer right-click integration, I suggest TortoiseGit. The default install options are fine but some can be changed, such as using Notepad++ instead of notepad.exe.

Git is harder to explain than to use. You should read or watch some of the resources available on our class web page for details, such as *Git for ages 4 and up*. A few basic concepts and commands will get you started:

- A **Git repository (or "repo")** consists of a directory where you keep your projects' files. This is called the *working directory*. This directory can have as many sub-directories as you wish.
- Also in the working directory is a sub-directory "`.git`". (In Unix and Linux, names starting with a period are hidden, making them easy to ignore. In Windows 7 or earlier, you can't create such files from GUI.) This sub-directory contains your repo except the working directory's files.

  A Git repo contains many things: a set of ***commit*** objects, which are snap-shots of the working directory's files at a given point in time (and other information); lists of references (easy-to-use names) to the various commit objects, to make working with them easier; and configuration settings. (This is over-simplified, but it makes a good starting point to understanding Git.)

- Git has computer-wide ("`system`") settings (with Windows 64-bit Git, at "`C:\Program Files\Git\mingw64\etc\gitconfig`"), per-user settings (oddly named `global`", in `.gitconfig` in your home directory), and per-repo settings (in `.git/config`).

  Before using Git for the first time, you usually should set your name and email address in the per-user settings, as every commit object should have that info included:

```
git config --global user.name 'Wayne Pollock'
git config --global user.email 'pollock@acm.org'
```

  The "`--global`" option says to view/edit the per-user settings; "`--system`" means the system-wide settings, and neither means use the default of "`--local`", the per-repo settings (current repo only).

Git (by default) can store usernames and passwords (or tokens) using your system's secure credential storage (such options are stored in the system-wide Git config file). For Windows, Git uses the Windows "Credential Manager". Using this you won't be prompted when using "git push".

If you have an authentication issue using (say) GitHub with HTTPS, you can remove the saved credential from there. The next time you connect, Git will

prompt you for your username and either password or token, then save them. (SSH credentials are managed differently.)

Note! Some IDEs include a bundled version of Git and may or may not use the standard configuration files! You should check the Git settings in your IDE to see if the changes made above are used. If not, you will need to make the changes from your IDE.

If using Notepad++ as your editor, you need to tell Git where it is and pass some options. Use this (all one line, including the quotes and apostrophes):

```
git config --global core.editor "'C:\Program
Files\Notepad++\notepad++.exe' -multiInst
-nosession -notabbar -noPlugin"
```

If you installed the 32-bit version of Notepad++, it's probably installed elsewhere and you should use the correct path.

(*Show .gitconfig; a sample is in class resources.*)

As mentioned above, dot files (files whose name starts with a period) are hidden on Unix, Linux, and Mac. But on Windows, only files set with the *hidden* attribute are hidden. To have a Windows user experience similar to Linux users, `git` creates a repo with the `.git` directory marked as hidden. To change that behavior, change this setting:

```
git config --global core.hideDotFiles false
```

Set to `true` to hide all created dot files; `false` means to hide nothing. The default is "`dotGitOnly`". To recursively remove the hidden attribute from a directory named `mydir` and everything in it, from the DOS command line run:

```
attrib /S /D /L -H mydir\*.*
```

Another setting to watch is `core.autocrlf`. This helps with the line endings problem (Windows uses <CR><LF>, others, including repo servers, use <LF> only to mark the ends of lines.) Usually setting to `false` works well, as most IDEs (even on Windows) are happy to edit files either way, and there is no need to convert back and forth. But if you use Cygwin's `git`, it will insist on LF only, and the changed line endings result in a conflict when trying to commit. Should that happen, remember to change this to `true`, which causes Git to change the line endings when checking files in or out. If not using Cygwin, a Mac, or a Linux host, setting this to `false` will usually work better.

Some fun things to try after setting up your `.gitconfig` (and a repo): Try "`git webui`" and "`gitk --all`".

- **To create a new Git repo** is easy. Make some directory the working directory (a new directory, or one with a project's files already present). Then run the command "**`git init`**". That's it! It's also easy to *clone* an existing repo; you'll do that for your Git project. (*Demo **git status** command before & after.*)
- Next you work on your files for a while, until you come to a good stopping point. That is usually after a successful compile, after a chunk of code is done, or simply the end of your working day. When you reach a stopping point, you should make a *commit*, recording the current state of your code. Each commit records all the changes you made: new files added, old files deleted, and modified files. (*Demo creating README.txt, show git status again.*)

  The more frequently you make commits, the more flexibility you have; this is called *granularity*. If you make a commit once a week, you can only see the current files or the ones from last week. But if you make a commit after finishing a method or adding a test or two, you can examine and restore the code from any of those points.

  > Unlike most older VCS systems, Git doesn't store the difference between versions of files. It stores complete copies of files in a "tree" that mimics the directory structure of your project. Each commit stores a copy of all the files (logically anyway; in practice, Git is more efficient).

- The biggest difference with Git is that none of the working directory files are included in a commit! (I lied earlier, sorry.) **You must add all the files you want to commit**, to what is called the ***staging area***. Initially, this was called the ***index*** and you will still see that name used in places (Eclipse Git has the command *add to index*). You need to add each new or modified file, and also need to "officially" tell Git about any files you deleted, renamed, or moved. That information is also included in the staging area.

  This feature is powerful in that you can add just the files you want in that commit's *file tree*. Once you set the staging area to the exact files (and deletions, etc.) you want, you make a commit. What gets saved in the commit object is all the files in the staging area, not the files in the working directory. This gives developers a lot of control.

  > If you have modified many files since your last commit, the resulting changeset (a patch) will be large and hard to do code review well. Instead, you can include only a few of the files (or just one) in a commit, and put the other changes in other commits. This keeps the commits small and easy to review.

  But the staging area is also a pain, since you must remember to add a fresh copy of modified files even if they were previously added. This is because after a commit is done, the staging area is left unchanged. So, if you created new files,

modified files, or deleted files from the working directory, none of that is automatically included in the staging area and thus not included in your next commit.

Usually, you just need to add all new and modified files to the staging area; just an extra step for developers. Fortunately, there is a handy short-cut. Adding a directory recursively adds everything under it as well. So, to commit all your files you just need to add the top directory.

The command to add files to the staging area is "**`git add`**". To add everything (except the `.git` directory), from the top directory of your project, use the command "**`git add .`**". Remember, if you add a file now and modify it later, you need to re-add it to the staging area or the old version is used in the commit, not the new version. (The `git add` command will ignore some files; see `.gitignore` below.) The `git status` command will report if you have files in your working directory that are different ("modified") or not present at all ("untracked") in the staging area.

(*Demo git add, then git status, then edit a file and run git status again.*)

- As you work on your project, you may want to rename, move, or delete files. The old renamed or deleted files cause problems with Git; when you commit, Git thinks you merely didn't add those files to the staging area. The result is the old files are marked as modified but not added. If you moved or renamed a file, the new name is seen as a new file by Git (when adding). This breaks the history of that file.

  If a file is under version control, you really need to tell your VCS about the file rename/move/delete. You could use your OS or IDE to move the file, then run a command to remove the old name from Git, then add the new name to Git. (If the contents haven't changed much before your next commit, Git will autodetect the file was just renamed, so you don't lose any history. If Git can't tell, you will lose history this way.)

  To move or rename a file with Git, use the command "**`git mv`**". To delete a file, use "**`git rm`**". These commands will move, rename, or delete the file from the working directory **and** update the staging area with the info. If you just delete a file from your IDE or directly from your operating system, the change is not made to the staging area and your next commit will likely surprise you. If you did this, you still need to tell Git about the change.

  Here's a recovery command you can use from the Git command line that finds the names of files you deleted only from the working directory, and has git delete them from the staging area too:

```
git rm $(git ls-files --deleted)
```

(This strange-looking command is actually two commands in one. The `git ls-files --deleted` command lists the names of all deleted files, and passes them as arguments to the `git rm` command.)

> Some IDEs will do all this for you automatically on a Git project, provided you use your IDE's commands to rename/move/delete files. Other IDEs will have the commands for this in one place or another.

Finally, Git can track if a file is executable or not. Implementing that feature was tricky since different operating systems deal with that differently. You can set the executable status when you add a file with "`git add --chmod=+x` *`file`*" (and clear it using "`-x`"). (In Linux terms, Git file permissions are either 644 or 755.) Git cannot (currently) track other permissions.

If you have added files to the staging area that you now decide you don't want to include in the commit, use the command "`git rm --cached`".

The related command "`git reset` *`file...`*" resets the copy in the staging area to the version in HEAD (the previous commit) but doesn't affect the copy in your working directory.

The command "`git reset --hard` *`file...`*" will revert both the working directory and staging area versions of the files to the version from the HEAD commit.

- Once you have added all your changed files to the staging area, it is time to make a snapshot, the commit object. The command to make a commit is "**`git commit`**". This will prompt you for a commit message. This message should summarize the changes made in this commit. (You can include the message on the command line instead, with "`-m "`*`message`*`"`" if it is short enough.)

> **It is a good idea to begin a commit message with a single short (less than 50 character) line summarizing the change, followed by a blank line and then a more thorough description.** The tools that turn commits into emails will use the first line as the email subject and the rest of the commit message as the email body. Also, the first line only is shown in the history (git log).
>
> You can use the Git commit linter to check your commit messages before the commit. Read Why Good Commit Messages Matter for good advice.

Git has commands (`git status`) to show you any files in the working area that are different from (or not added to) those in the staging area. IDEs can show you that and the differences from your last commit as well. The GUI makes it easy to add (or remove) files from the staging area, before committing.

- **Not all files in the working directory should be placed in the VCS**. Aside from the `.git` directory itself, you rarely need to or want to share generated

files (.class or .jar files), temporary files, or IDE-related files (not always though). And while you will want media files in your repository, you really don't want to version them if they are large AND reproducible (backed up) outside of your VCS. (VCS systems such as Git don't do well with large, non-text files). On the other hand, icons and other media elements of your application may change from version to version, so you do want to add them to your repos. Usually they don't change often, and thus are not a problem.

In the working directory (or any sub-directory), you can create a file named "**`.gitignore`**" which lists file and directory names not to include when you run the `git add` command. The names can include wildcards such as "`*.class`". Listing a directory here ignores all children of that directory as well. See gitignore pattern format for details.

For our class Java projects, we need to ignore IDE-specific files and directories. (You will likely want to do the same with any repositories you create; otherwise, as you work in teams, you may find your IDE settings changing every time you do a pull!) You can create a `.gitignore` file with these entries to ignore such files, and generated output too:

```
.classpath
.project
bin
build
target
nbproject
build.xml
manifest.mf
.idea
*.iml
.mvn
mvnw*
.vscode
```

(Notice how many of these files are *dot files*, hidden on Linux and Mac.)

When you run the `git add` command, files matching those in this .gitignore list will not be added to the staging area. You can override that with a force option.

- **References:** A *reference* is a handy short name to some commit object. An initial repo has no commits in it, and thus no references. (You'll see the message "NO HEAD" when you run some Git commands.) After your first commit however, Git makes some references to it automatically.

The reason references are so handy is that commits are named by a unique 40-digit hex number. You can abbreviate that number to the first six characters or more, as long as that number is unique among all the commits in your local repo. **These commit IDs are globally unique in the world; they are actually SHA1 checksums (or message digests or hash codes) of the commit object.**

> Hard-coding SHA1 as the naming algorithm was a mistake. Today SHA1 is not considered a good choice and other algorithms (such as SHA2 or SHA3) would be safer, but it is hard to change this in Git. This is cost of coupling! The algorithm should have been wrapped in a functions such as "generateName", then the algorithm is in just one place and easy to change without affecting any other code.

By default, Git creates two references from your first commit. "**`master`**" is the name of the *branch* (see below). (The term "master" in Git may be replaced with "**main**" (or something else) someday, but as of 1/2021 this hasn't happened. However, I will use "main" in this course.)

The other reference created, "**`HEAD`**", is the last checked-out commit on the current branch, also known as the *tip of the branch*. (`HEAD` is really just a reference to some branch name, which in turn is a reference to the most recent commit on that branch, the *tip*.)

Non-branch references are called ***tags***. You can use the command "**`git tag`**" to list, create, or delete tags. (Tags are discussed in more detail, below.)

> Another type of reference is to a remote repo, typically a URL. For example, when you clone a remote repo (say from GitHub) locally, a reference named **`origin`** is created that refers back to the original repo.

You can see all your references with the command "**`git show-ref`**".

- All work in Git is done in ***branches***. By default, your first commit is done in the branch "**`main`**" (see the comment above about names). Each time you make a new commit, the current branch reference is changed to refer to the new commit. So what about the previous commit?

  The ID of the previous commit is stored in the current one, so the commits in a branch forms a chain (a linked-list) from the tip of that branch back to the initial commit object in the repo. Each commit except the first contains the name of the previous commit, called the *parent*.

  > Git branches allow for several possible workflows for your group. You can have each developer commit and push their own branch, then merge only some of those commits into main. You can have developers use a branch

per feature or issue. You can have developers push directly to main, but then you get lots of manual merges unless you take care to avoid that.

If you decide to use pull requests (patches) in your workflow and you use GitHub, it works best if each developer can "fork" the main repo to their GitHub account, and clone that on their PC to work. This is because GitHub has great pull-request features, but you can only use them if you create your pull requests from one GitHub repo to another GitHub repo.

A modern developer workflow called Git flow is to use the `main` branch only for tagged releases of code, and a branch named `develop` for all else. You start work on something (a "feature") by creating a new branch off of `develop`, never `main`. You push your branch up to the shared repo (say GitHub) so others can view and collaborate. When ready, you can create a pull request or otherwise make your changes available for code review. Finally, when approved, you merge the feature into `develop`.

(Simpler workflows are described in project directions.)

You can refer to previous commits in a number of convenient ways. For example, to toss out all work since you last commit, you can check out `HEAD`. That replaces the working directory files with the ones from the most recent commit. (Be careful with that, there's no undo.!)

You can refer to older commits in the current branch easily. To compare the last two commits for example, you can use `HEAD` and `HEAD^` (or `HEAD~1`).

You can use this notation to refer to other commits, relative to `HEAD`. For example, `HEAD^^^` or `HEAD~3`.) You can use this notation with branch names too, such as `main~3`. (You can compare commits with either the working directory or the staging area as well. The command for this is "`git diff`". For example: `git diff HEAD~ HEAD`

(You can also use `git show HEAD` for similar output. *Demo.*) You can use any commit reference instead of `HEAD` here.

To create a new branch is easy:

```
$ git branch branchName
```

This creates a new branch without switching to it. The new branch refers to the same commit object as `HEAD`. The command:

`git branch` `[-a]`

(The "`-a`" says to list all branches, including *remote-tracking* branches discussed later.) This lists branches (the one with the "`*`" is the currently checked-out branch) and can be used to get other information about branches.

Use "**`git branch -d` *`branchName`***" to delete a branch.

To create a new branch reference and make it the current branch at the same time, use:

```
git checkout -b branchName
```

Branches are usually created for each feature, each issue (often from a trouble-ticket or issue tracker), or just because you want to experiment a bit without mucking up the main branch.

Once you have several branches, you can switch working on one to another easily, using "**`git checkout` *`branchName`***". This makes all tracked files in your working directory identical to the versions in the commit object you just checked out, including deleted, renamed, and moved files. Untracked, modified, and staged files are left unaffected unless you add the "`-f`" (force) option).

> You can re-checkout the current branch: "**`git checkout -f HEAD`**" will revert all tracked files and staged files. You can also checkout one or more files and not the whole branch.

If the branch didn't work out, you can delete the branch. That deletes the reference, not the commits. Git servers including GitHub periodically run a *garbage collector* to find *orphaned* commits and delete commit objects not on any branch and without tags (so there's no simple way to refer to them anyway). You can run the command "**`git gc [--aggressive]`**" to manually clean local repos.

> **You can checkout any commit object.** If you don't checkout one on a branch tip, if you make changes and commit, the new commit will have no references to it; none of the branch references change. (If you do this, you're in what is known as a *detached HEAD* state.) Should you decide to keep those commits, you should make a reference to them. The easiest way is to create a new branch.

- **Merging:** If the branch did work out, you then can ***merge*** it into another branch such as main. This operation usually results in the one branch's commits appended to the end of the other branch. Here's an example of using a branch and merging it later:

```
$ git checkout -b myfeature main
Switched to a new branch "myfeature"
```

Let's say you work on your new branch for a while, making several commits. When done, you want to incorporate all your changes into the main branch. To merge the changes you've made in the `myfeature` branch into the main

branch, first make sure you've committed all changes on the `myfeature` branch:

```
$ git add .
$ git commit -m "...my changes..."
```

Then switch to the branch you wish to merge into, and run the merge:

```
$ git checkout main
Switched to branch 'main'
$ git merge myfeature
Updating ea1b82a..05e9557
(Summary of changes shows here)
```

You always merge some branch into the one you are currently on (and have checked out), or you can list both branches on the command line. When done, you can delete the branch reference (all its commits are now part of the main branch):

```
$ git branch -d myfeature
Deleted branch myfeature (was 05e9557).
```

This deletes the `myfeature` branch reference, but not any of the commits. (If you delete a branch with commits that weren't merged into other branches, those commits become *orphaned*, since there is no reference to them anywhere. (As mentioned above, Git servers run a garbage collector periodically to delete these, or you can manually run `git gc`.)

This automatic merging (called ***fast-forward***) usually works fine. But it can fail if other commits were added to the main branch before attempting the merge of your branch. If the merge fails because the same files were modified on both branches with different changes to the same line(s), the merge operation will be interrupted at that point. Git will check out a special version of the project's files, one with conflicts marked in the file(s) to show the different content from both branches. Git also updates the index (staging area) to indicate we are in the middle of a merge. (The new versions of non-conflicted files are placed in the working directory too.) It is up to the developer to resolve the conflicts in each conflicting file somehow. The conflict markers in a file look like this:

```
<<<<<<< HEAD
echo hello git world
=======
echo hello, world
>>>>>>> mybranch
```

Note not all conflicts are caught by Git: if a variable name is changed on one line and used on another, the two changes are to different lines so no conflict. Compiling and testing catches these sorts of conflicts.

Merging the commits from one branch onto the tip of (say) the main branch is one way to include your work in main, but Git has a different method as well, called a ***rebase***. Sometimes, using rebase is preferable to using merge. (See below and the rebasing chapter in the Pro Git book for information on rebasing.)

Normally, you open the conflicted file in an editor, examine the two versions of the conflicting lines, then decide which version (or some new version) of each conflict to keep in the file. (You can also run **`git mergetool`**, which should launch a GUI editor showing three versions of the file side-by-side-by-side.)

If the merge seems hopeless, you can cancel the merge operation (restoring both branches) using the command "**`git merge --abort`**".

To see which file(s) have conflicts, run **`git status`** (the "unmerged" files are the ones you need to fix).

Once all conflicts are fixed, you complete the merge by running **`git add`** (you made changes after all), then re-run **`git commit`**. The fixed/modified files become part of a new commit object with two parents: each of the branch commits.

Demo merge: create repo; make 3-line `todo.txt`; commit; make a branch `mybranch`; make a change and commit; switch back to main, make change to the same line and commit. Now try `git merge mybranch`. Then show status, show conflicted file. Then run `git mergetool`. Finally, (still on main branch) do an `add` and `commit`. That completes the merge operation!

Visualize repo with:

```
git log --graph --oneline --all --decorate=full
```

(ASCII art version) or "`gitk --all`" (GUI). **Homework:** try this same demo using your preferred IDE.

Once you resolve conflicts, before completing the merge (via a commit) you should re-run all your tests to make sure your resolution didn't cause any new problems.

Merging with vimdiff

When merging two branches (say branch A and branch B), Git performs what is known as a *three-way diff*. This means Git determines the most recent common ancestor of both branches and runs `diff` between the ancestor's versions of files and A's versions, then between the ancestor's version and B's versions of the files. The two sets of diffs are compared. If

there are no changes to the same lines of the same file, Git has no conflict and does a fast-forward merge.

If there is a conflict, both `diff` outputs are put into the file(s) that have conflict(s). You can manually resolve the conflict(s) using any editor, but there are GUI tools that can help. When you run `git mergetool`, Git tries to locate such a tool on your system to use. Often this will be the default tool `vimdiff`.

If you don't know `vim`, use some other GUI tool (see below) or a regular text editor. If you know the `vim` editor, you may still not know the command shown here.

You can find a short and excellent tutorial at www.rosipov.com, but here's the basic information you need:

`vimdiff` splits the console window into four panes, three in the top row and one in the bottom row. The top row windows show different versions of the file; from left to right: the LOCAL version (the one from the current branch), BASE (the version from the common ancestor), and REMOTE (the version from the branch you are trying to merge). The bottom pane shows the current version of the file, with the conflict markers.

Use `^w` to cycle the cursor from pane to pane. Use this to move the cursor to the bottom pane. Use the arrow keys to move the cursor into the conflict area. From here you can replace the conflict area with any of the other three versions:

`:diffg LO`	*replace with the LOCAL version*
`:diffg BA`	*replace with the BASE version*
`:diffg RE`	*replace with the REMOTE version*

You can choose to use none of those versions, or start with one of them and then make further changes (in the bottom pane) using regular `vim` editing commands.

When done, use the command "`:wqa`" to save your work and close all the panes at once. Next, check to see if vimdiff created a backup version of the file ("*`file`*`.orig`"); if so, you should move it out of your working directory (and out of your repo), or just delete it. (Or add to git config `mergetool.keepBackup false`.)

When done with all conflicts, run `git commit`, and the merge is done.

Although `vimdiff` comes with Git, there are other free tools available such as the much easier to use Meld, also TortoiseMerge and Kdiff3, and some

commercial tools such as Beyond Compare. (Make sure the tool is found on `PATH` or add to git config `mergetool.`*`toolname`*`.path`.) I like Meld, but on Windows it takes a long time to start up after a reboot.)

Merging with Multiple Repositories

A common case is that you have cloned a repo from (say) GitHub. At that point, there is a copy of the main branch (and/or other branches) on your local computer and on GitHub. When you make a new branch, it is only local. To allow teammates to view your work, you need to **push** your branch (and its commit objects) to the GitHub repo.

Suppose you work for a while and want to merge your changes into the main branch. Your teammates have reviewed your changed and everyone approves. So you merge your branch into main, right?

NOT SO FAST!

Your copy of the main branch may not be the same as the one on GitHub! That happens if others have merged and pushed onto main since you checked it out.

If you do merge with your older copy of the main branch, it will work without conflicts. But when you push your version of main up to GitHub, the other commit objects will seem to have vanished! (They are still there, for a while, and you can revert your change on GitHub to restore them.)

To avoid that issue, you must **fetch** a fresh copy of the main branch to your local computer, and only then do the merge. If your changes conflict with some of the other new changes, you will need to resolve them by making a new commit object. (This is when merging is not automatic, or "fast-forward" as Git calls it.) Finally, you can **push** your version of main to GitHub.

Summary: *Clone* from GitHub to your PC; make (local) *branches* for each of your changes; *push* your branches to GitHub for others to review. When your commits are approved, *fetch* or *pull* the current copy of main from GitHub, and *merge* your branch into that. Then *push* the new main branch to GitHub. (More details on cloning and working with remotes is below.)

> If using *pull requests* (PRs), you should clone on GitHub to your personal GitHub repo, then clone that to your PC. The extra work involved makes it easy to produce pull requests: as it is your personal repo you can push to it anytime without fear of conflicts. Then you can make a pull request very easily. If approved, it is also easy to merge in your changes.

- **Tags:** You can create **tag** names for commits. Unlike branch (and `HEAD`) references, tags don't get reassigned to newer commits. Tags are often used to name released versions of a project, such as "`v1.0`".

A simple tag on the currently checked out commit is created with:

```
git tag name
```

The rules for tag names are the same as for any Git reference name ("refname"), and can be found in the Git check-ref-format doc. Briefly, tag names are made from ASCII letters and digits, and a few special characters. They cannot contain spaces, "?", "*", "^", ":", "~", or "["; cannot start nor end with a dot, cannot contain "..", and have other restrictions as well.

You can add a commit ID or other reference on the command line, to add your tag to that commit instead of the currently checked-out one.

Such tags are known as *lightweight tags*. You can also have *annotated tags*, where you add additional information in the tag. You can have a creation date, the name and email of the tag creator, a tag message, and optionally, a digital signature of the commit object (created with `gpg`).

When you do a push or pull, you don't get all the tags automatically. (Only some tags will be fetched.) You must use the "`--tags`" option to also `fetch` or `push` all tags. (By default, `pull` has no option to get all tags but you can make a config change to your local repo to make that happen.)

Changing tags (by pointing them to a different commit object) is no problem if you never shared your tags with another repo. But there is no way to have Git change a tag "Internet-wide". It is probably best to simply add a different tag, and ignore the incorrect one.

Tag names in Git are often used as the end-user visible ***version number*** of something. There is a standard (okay, many of them) for version numbering. A common and good one is found at semver.org:

Given a version number MAJOR.MINOR.PATCH, increment the:

MAJOR version when you make incompatible API changes (is zero for all pre-release versions),

MINOR version when you add functionality in a backwards-compatible manner, and

PATCH version when you make backwards-compatible bug fixes.

In addition, usually some extra information can be included (after a dash), such as W.X.Y-Z, where the "Z" represents additional info, such as a *build number* (often the encoded date when the code was compiled) or strings such as "beta1", "rc3", and so on.

The tag for version "1.2.3" should be "v1.2.3" for automated tools to find the newest release version.

- You can examine some or all of the history of a repo using **`git log`** command. This has many, many options. I find it useful to define a custom log format, and a `git alias` command to generate a log using that format and my preferred options. Here's how you can do that:

  (*Show Triangles visualization and log.*)

```
# Create custom log format, use with alias:
git config --global pretty.myfmt \
 'Commit: %h, Parent(s): %p, Author: %an, Date:
%ad %d%n%B%n'
git config --global alias.mylog \
   'log --pretty=myfmt --full-history \
   --branches --reverse'
git mylog  # How simple to use!
```

- You can create a repo by making a copy, or "***clone***", of another repo. Such repos are known as ***remotes***, and references to their URLs are kept in your local repo. By default, the remote repo you cloned from is named "**`origin`**". The branch you get is named `origin/main`.
- **Remotes and Tracking Branches:** Besides references to commit objects in the local repo, you can also have references to other repos (*remotes*). A *remote reference* refers to the URI of some other repo. Using remote references, you can fetch the remote repo's commit objects (that you don't already have), as well as the related branch references and tags, from the remote repo without typing a URL each time. (With the right options, you can fetch all of the remote's references.) You can also push your commit objects and references to some remote repo. Use the **`git remote`** command to manage your remote references.

  As mentioned above, if you clone a remote repo, your local version will include a reference to the original repo, called **`origin`**. However, you can add, list, or delete remote references easily.

  > For various reasons, the remote repo is often called the ***upstream*** repo. You will find that term used in several Git commands. Usually, the upstream repo is the more authoritative repo, likely the one you cloned.

  So far this is easy to understand (easy compared to all of Git anyway). There is a wrinkle though. **When you fetch from a remote, none of your (local) branches will be affected at all!** So if your teammates have added two commits to the main branch at GitHub, and you fetch from that repo, your local main branch will remain two commits behind (assuming your main was previously pointing to the same commit as the main on GitHub).

What happens is this: Git adds the references from the remote and adjusts a local copy of the remote's branch reference. Typically, the name will be "`origin/main`" (*remote/branch*). After fetching the remote commits you didn't have, that reference gets updated, not your local `main` branch reference.

Such names (*remote/branch*) refer to a read-only copy you have locally, known as ***remote tracking branches***. Because these merely track the state of a remote's branch, Git won't allow you to add commits to it directly. Thus, it is always safe to `fetch` (there cannot be a conflict).

To actually update your local branches (say `main`), you need to merge into your (local) main from the remote/main tracking branch. Obviously, you should do that only after a fetch, to ensure your remote tracking branch is up to date. **Merging is always a local operation**; you are merging the remote/myfeature branch into your local myfeature branch, or your origin/main branch into your local main branch.

Many Git commands take branch names and/or remote names as arguments, but will default to `main` and `origin` if you leave them out.

In Git, a local branch reference can be annotated with the name of a remote tracking branch. (Actually, it uses entries "branch.<name>.remote" and "branch.<name>.merge" in the repo's `config` file.) Such a local branch is often called a ***tracking branch***. (Not to be confused with a remote tracking branch!) Because tracking branches keep a reference to the remote URI, you can use commands such as `git fetch` without specifying the remote location; Git will notice the remote tracking branch info and get the URI of the remote and its branch name.

As a convenience, cloning a repo automatically makes a (local) `main` branch and sets it as a tracking branch of the `origin/main` *remote tracking branch* (and the same for other branches). This means after cloning, you can just use git fetch/pull/push with no further arguments.

Any (local) branch can be set to track any remote tracking branch reference. You can change or delete that annotation anytime using the `git branch` command.

***Summary*: When cloning remote repositories (discussed below), Git creates two copies of the branches: as local (tracking) branches and as remote tracking branches.** A *fetch* operation will update the remote tracking branches only. Meanwhile, all your changes are made to the local branch only. That's why you want to update the remote tracking branch from the remote repo, before merging in your changes (from your local branch into the remote

tracking branch). Note a merge operation is always a local operation between two branches on your computer.

Pull: To update the local and remote tracking branches from a remote repo, you will have to perform a *merge* operation after fetching. **The operation *pull* combines a fetch and merge**. (You are merging `remote/branch` into your local `branch`.) In case of a conflict, resolve the conflicting files (by editing them), `add` them then `commit` the change, and the merge is done.

A push is different: Your repo contacts the remote one, uploads any commit object you have that they do not, then merely asks the remote to "set your main branch reference to the same commit I have my local main set to". By default, the remote Git will allow that if no conflicts are noted (equivalent to a *fast-forward* merge). If there is a conflict, the push fails; there is no merge. The "`-f`" (force) option can override the remote's rules; **it's ~~almost~~ always a bad idea to do this.**

What you need to do is, first do a `git fetch`. Then merge in your changes to your now-up-to-date main, resolving any conflicts. Now when you do the push, there won't be any conflicts at the remote end (unless some wise guy made new changes since your fetch; if so, repeat). (See stackoverflow.com for a very good explanation.)

When pushing a new branch to some remote, you should use the command "`git push` **`-u`** *`remote`*". The "`-u`" option causes Git to create a remote tracking branch for each local branch you successfully pushed and annotates the local branch (so it becomes a local tracking branch).

Summary: Say you want to share your repo's contents with others, or update your repo with their repo's contents. In Git, you can ***pull*** commits and references from a remote, or ***push*** your commits and references to a remote. (Sharing tags requires a separate command for some reason.) These push and pull operations are actually combinations of **fetching** and **merging**, just less typing. If there is a conflict when pushing, the operation will fail; you'll need to download the new commits from the remote repo, manually merge the changes, and then re-try the push. (Of course, you will need appropriate permissions to read and/or write the remote repo, before pulling/pushing.)

Pull Requests and Patches

It is rare that users will push changes to the main branch on the "official" repo for a project. It is much more likely you will need to have your code reviewed and approved before it gets merged. As a member of some project, you can create a branch from main for your change, then when ready for review, you can push your branch so everyone on the project can review it.

It is common to have few or no *collaborators* (those with permission to make changes to your repo) for a repo on GitHub (or some other upstream repo). How then do others contribute? If you do not have permission to push to some repo, you can send a request to someone with such rights. You request they pull in your branch and (after review and approval), merge it into main. Such a request is known as a *pull request*, or PR, which contains a link or URL to your repo and the branch or commit ID you want them to pull.

You should create a separate branch for each pull request, otherwise later commits into the same branch will become part of the older pull request. This is the normal best practice of making a branch for each change.

You can easily create pull requests from `Github.com` between two branches of some repo, or between two GitHub repos. It only takes a couple of mouse clicks. Such a request shows the changes you made and lots of other information. GitHub will also show if a merge would suceed or not. If it would, there's a button you can click to do the merge.

Because it is so easy to work with PRs and they show the specific changes made, it is very common to only review and merge PRs. Reviewing a branch doesn't show highlight the changes made by that branch.

> Many people think a "pull request" is specific to GitHub, but that isn't so. The term applies to any DVCS.

After (the team) reviews and approves your changes, a normal `git merge` is done from your feature branch into your local `main`. Then you push that into the remote repo's main branch. Or, if using GitHub and PRs, you can easily merge that PR. (This is why developers often clone a GitHub repo on GitHub itself, then make a local clone to work on. As mentioned earlier, you can always push to your GitHub copy of the repo, then easily make a PR on GitHub. The extra work comes from pulling in changes to your GitHub repo before pulling changes locally from that.

If problems are noted in the code review, the normal practice is to abandon that pull request, make changes, and make a new pull request. After a successful merge/push, everyone else must do a pull to update their copies of the repo.

To actually pull in the changes, the person you send the pull request to must have Internet access to your repo. Since few of us run public Git servers at home, you generally will push your work onto your GitHub account area (where you've previously forked the repo) and use that repo's URL. (This is known as ***fork-and-pull***.) So even if you don't use the PR workflow, you often will need a personal GitHub copy of the repo anyway.

Another approach is to use one group repo and everyone works in their own branches. You then create pull requests from your branch to the main branch, all within the one GitHub repo.

(Having all users push directly to main is risky, unless you are a small group and everyone communicates often. Using PRs is not only easier for reviews but a lot safer!)

Creating a pull request locally is done with **`git request-pull`**:

```
git request-pull -p start my-URI end > pull-req.txt
```

Where *`start`* is a reference to a commit already on both the remote (upstream) repo and your public (Internet accessible) repo, *`my-URI`* is the URI of your repo (where the commits are pulled from), and *`end`* which defaults to just `HEAD` and is usually omitted. ("`-p`" says to include the differences as a patch; see below.)

This command generates a request asking your upstream project to pull changes into their tree (repo). The request begins with the branch description, summarizes the changes and indicates from where they can be pulled. All commit objects from *`start`* through *`end`* are included in the request, which is then saved in the file `pull-req.txt`. It is common to add some text comments at the top to help reviewers understand the purpose of your PR.

How do you know which local commits have not yet been pushed to (or pulled into) upstream? The "`git status`" command won't help with this. You can use this command to see a list of such commits:

```
$ git log --branches --not --remotes
```

Patches

There are two ways to share your code made in a non-Internet accessible repo: You can do the fork-and-pull described above (with or without using PRs), or you need to send an email to someone containing the information needed to review the code and (if approved) make the changes.

The result is not an actual pull request, since it contains no URI that someone could use with `git pull`. Instead, the resulting text is just the differences (or "diffs"), also called a ***patch*** or a ***patchset***. The patch includes the differences for all modified files as well as information about added, deleted, moved, and renamed files. These differences are in a standard format called Unified-Diff.

> It is not uncommon to use the terms *pull request* and *patch* interchangeably. Many prefer to use the terms differently, to indicate the exact mechanism used to work with the request.

If your change is approved, the patch can be applied mechanically to the target repo and a commit made. You also need to add information to the list of diffs, so the human who reads your pull requests and patches can make sense of them!

> Using patches to share code predates version control systems. Then (and now) you can use the old Unix/Linux `diff` utility to make per-file patches. A Unix admin named Larry Wall (inventor of Perl) created the tool `patch` to automate applying such diffs.

You cannot create a patch using `git request-pull`, since that command requires the URI of the repo, and here we are discussing the case when your repo is not Internet accessible. In this case, you can generate the patch output to save in a file, that you then send as email.

There are several versions of the **`git diff`** command, depending on what you want to compare (the working directory, the staging area (index), HEAD, or some other commit). Assuming you committed your work on a branch named `feature`, you will want the differences between the tip of feature and main:

```
git diff -p main feature > mypatch.txt
```

> In the standard patch file format, added and deleted files show as changes to/from "`/dev/null`" which is a special, always-empty file from Unix. A moved or renamed file might show as an add and delete pair of changes.

You can create patches without Git, using Gnu `diff`:

```
LC_ALL=POSIX TZ=UTC0 diff -Naur old-version new-version
```

Run this command from a shell prompt. On Windows, use WSL or the shell installed with *Git for Windows* named "Git Bash" (there are other Linux environments available as well).

Another command to create patch files from commits is git format-patch.

You can apply a patch to the working directory (or the stage or both) with git apply.

(Show and review *pull requests and patches* in our class resources.)

Patch File Format

Patch files contain one or more `diff` outputs ("chunks"), concatenated. They can contain other text ("junk") at the top, bottom, or in-between `diff` outputs; junk is ignored by `patch`. So you can just use a saved email message with a patch in it, headers and all, and the `patch` command should still work. Typically, a developer will add some comments in front of each diff chunk in the patch, to help the reader understand the change without having to view some issue or examine the commit messages.

Other Git Features

- Some users don't like the shortcut of `git pull`, as it does too much automatically. Instead, you can run `git fetch` to update your local repo with the latest commit objects from the remote repo and will only update the remote tracking branches. This command has no effect on your working directory or your local branches. You can now see what's different before making any changes. You can merge in all or some of those commits, or **use the command `git reset --hard origin/main` to blow away all local changes, thus making the local repo's branch identical with the remote's**. (There is no undo for this!)
- When looking at a file, it can be useful to know when some line or block of lines was added or last changed, and by whom. You could then look at that commit's comments or contact the author. To annotate a file with a per-line comment showing the commit name (hex number), author and date, use the **`git blame`** command.
- **Using Git Stash:** A common issue is when you have modified files in the working directory but need to look at another commit or switch to another branch for something urgent, before finishing work and committing your modified files. Checking out a commit will overwrite (tracked) files in the working directory, so you may lose your work. If possible, just commit before switch branches.

  If a commit before checkout is not reasonable, Git includes a place where you can stash your work temporarily. (You can stash multiple times.) Suppose you're working on branch1. Then:

  ```
  git stash
  git checkout branch2
  ...
  git checkout branch1
  git stash pop
  ```

  (Or use `git stash list` to see a list of stashes, then `git stash apply` *`name`* to restore the selected stash.)

Git stash saves both the working directory and the index, then checks out HEAD. Note "`git stash`" is a quick version of the full command "`git stash push`" without any options allowed.

Stashing files this way is also useful if you've made many changes to the working directory, before discovering you are working in the wrong branch. In that case, you can stash your work, switch to the correct branch, and pop the stash. (By the way, creating a new branch with "`git checkout -b` *`newBranch`*" will not overwrite the working directory's files. So to save your work in a brand-new branch, you don't need to use the stash.)

Git stash can also be used to improve your workflow. Suppose you decide to change some interface by adding a new method. If you do so, your working code will show with an error every place that interface was used. (Yes, you could add a default implementation for the new method, but that's not always a good idea.) It is common to ignore the errors initially and fix them one at a time, making a new commit once the code is working again. The problem is that become a large commit; if an error is discovered later, you might need to examine dozens or hundreds of lines of code, from many files to locate the error.

Instead, use Git to make smaller commits this way: `git stash` will save your interface update and restore the working directory to the working version of your code as of the last commit (HEAD). Now go to the first place the interface is used and add in the required method. (You can copy the method declaration before you use git stash.) If you don't want to implement the method now, just make the body throw a `NotImplementedException` for now. Test this small change (all test should still pass since no existing code is using the new method yet) and commit. Repeat for the other uses of the interface. When all is well, use `git stash pop` to redo the interface change, test, then commit. You now have many small commits, each leaving the code in a working state, rather than one large commit. (See also `git add -p`, mentioned below.)

- **Using GitHub's Issue Tracker:** GitHub's issue tracker is a great way to organize your to-do list. You can add an issue for every user story (or use case if using more formal requirements). For every feature you think to add to your project, or every idea you want to explore, add an issue. The issues are numbered and optionally labeled, commented on, assigned, etc. My favorite feature is if you use a keyword followed by an issue number in a commit (or pull-request) message such as ""closes #123" or "fixes #123", when your commit is merged into main GitHub will automatically close the issue! Assigning issues to team members is a good way to divide up the work fairly.

In fact, GitHub has many useful features such as a wiki, Kaban-style project management, Actions, and many others.

Git supports many other commands and features, such as the important "**`rebase`**" command that allows you to combine commits, making your project's history a bit neater (example: "`git rebase -i HEAD~4`" to replace the last 4 commits with a single equivalent one). This is useful to make "neater" pull requests. You can also rebase a branch to change where it was based from. If your branch was based on some commit no longer the HEAD of main, you can rebase your branches commits onto the new HEAD.

Just as with merging, rebasing can run into conflicts. The remedy is the same: use git status to show the conflicted files, fix them up, add and commit the fixes. After a merge, you'd be done at this point but with rebase, you need to run the command "`git rebase --continue`" to finish up.

But `rebase` can be dangerous; **never** rebase if you've already shared (pushed) those commits or made a pull request of them! (GitHub won't even allow you to push rebased commits without the "--force" option.)

Use lots of small commits when working, but when done and ready to create a pull request, then consider rebasing. See atlassian.com/git/tutorials/merging-vs-rebasing for more information on rebase; a few more details are below as well.

The other commands mentioned here should be enough to get you started using Git and GitHub.

Git Issues:

When sharing code there are a number of issues to consider. One is the source code files' (and other text files') **encoding**. NetBeans defaults to UTF-8 and can be changed per project. Eclipse defaults to the OS native encoding (CP-1252 on Windows). But you can change the default and set per file encoding. (As of Java 18, the Java default is always UTF-8, so at some point (2023?) the IDEs will likely default to UTF-8 as well.) **You should prefer UTF-8 whenever possible.**

Another issue is **line endings** of text files, either LF (Unix style) or CRLF (Windows style). Git can ignore the issue or change line endings when checking files in and out; there's a setting for that. IDEs may have setting for this as well. It's an issue since different line endings appear as changed files to Git and may prevent a push or pull operation from completing.

Git issues include non-technical ones, such as a project's mission, organization, rules for participation, etc. (This is mostly for FOSS projects.)

It's a good idea to **document a project's conventions (a.k.a. *governance*) in a README file**. Common with GitHub (and elsewhere) is a text file formatted in a special way known as *markdown*. It is very readable plain text with some

conventions, allowing automatic conversion into HTML. Such text files usually have an "`.md`" extension. See Markdown home for more information. ("reStructuredText", often abbreviated as RST, is an alternative popular with Python developers; such files often have an "`.rst`" extension.)

In real life, a license is a good idea as well. GitHub can add these files automatically when creating new repos.

It is possible to merge multiple branches at once. The resulting commit will have many parents and the history will look like an octopus at that point. The "^" notation allows you to specify which parent you want to reference ("HEAD^2").

Summary of Common Git Commands:

(There's lots of such "cheat-sheets" available on the web. This list was adapted from one by Carla Schroder, posted on linux.com.)

Install Git. You need that first, of course.

To get help on any git command, run "**`git command --help`**".

Run "**`git config which name value`**" to specify your name, email, preferred editor (for commit messages), and other settings. For example:

```
   git config --global user.name "Hymie Piffl"
   git config --global user.email "hpiffl@example.com"
   git config --global core.editor "'C:\Program
Files\Notepad++\notepad++.exe' -multiInst -nosession
-notabbar -noPlugin"
git config --global merge.tool meld
git config --global mergetool.meld.path /c/Program
files (x86)/meld/bin/meld
```

The *which* setting can be "`--system`", "`--global`" (which means per user; usually that's the one you use), or "`--local`" (the default) for just the current Git repo (the file `.git/config`). You can view the settings with "`git config which --list`". If you omit *which*, the merged total configuration is shown.

Some IDEs may include a bundled version of Git. Using such an IDE may use different (internal) configuration files than the standard ones, so change configuration using your IDE and if necessary, the Git command line tool.

> Configuration attributes can be set per file as well. See the documentation for the .gitattributes file.

To first work on an existing project, create a new directory (say "myProject"), `cd` into it, then run "**`git clone https://[remote-repo-address].git`**". That will fetch a copy of some remote git repo, say from

Github.com. When done, you should have a directory full of files and subdirectories (including one named "`.git`").

To start a new project, create a new directory (say "myProject"), `cd` into it, then run "**`git init`**". You are now set to go create files (in the main branch).

To work on a new branch (rather than the "main" branch), run these commands:

```
git checkout main
git pull
git checkout -b myBranch
```

These commands set the current branch to `main`, populate the current directory with the latest version from the remote repo (automatically named "origin"), then creates a new branch. Note only after you edit then add files to your branch will anything change.

To see what branch you're on, and what files have been added, deleted, or modified, run the command "**`git status`**".

To delete a file from your current branch (only), use "**`git rm`** ***`file`***".

To see the history of a given file, use "`git log` *`pathname`*".

To see a list of all commands that modified the repo, use "**`git reflog`**".

To discard all changes to a file or restore a deleted file (if you haven't made a new commit yet), use these two commands:

```
git reset HEAD file
git checkout file
```

Note you can make as many changes as you want, and run git add as often as you like, before making a commit.

To save your changed branch to the remote repo, use the commands:

```
git commit -a -m "message explaining changes made"
git push origin myBranch
```

(The option "`-a`" updates any modified files already in the staging area; handy when you haven't created any new files that need to be added.) Note if someone has pushed a change before you, the push will fail! You will need to pull or fetch the new version of the branch and resolve any conflicts before trying the push again.

To merge in your changes, which is required if someone else committed their changes before you did (to the same branch), run "**`git mergetool`**". (See the box above, for details on Git merges.)

To undo a commit, you need to find the commit's ID using "`git log`", then run "`git revert` *`the-ID-number`*". This actually makes a new commit object

that restores the files to the same state as the commit you specified. (For a small problem, just fix and commit again.)

If you have not yet shared the commits, you can reset the references to a branch's tip to discard the most recent commits:

```
git checkout myfeature
git reset --soft HEAD~1
```

The commit object that used to be at the tip no longer has any references to it. Such *dangling commits* are *orphaned*, and get removed by Git automatically when it runs its garbage collection.

The "`--soft`" option says not to affect the staging area or the working directory. "`--mixed`" is the default, which means the staging area is updated with the version from the (now) HEAD, and the working directory is not affected. Finally, "`--hard`" means to update everything from the (now) HEAD.

So "`git reset HEAD`" will un-stage (undo your last "`git add`"), and "`git reset --hard HEAD`" will toss out all your changes since the last commit.

> Note `git reset` only affects files that are tracked, so new files you haven't added are not affected by a `git reset`.

Remember, resetting should not be done if the commit has already been shared with another repo (pushed or pulled)!

If you have not yet shared any commits, you can also completely delete or fix a bad commit from the project using `git rebase -i`. To delete one commit 3 commits ago:

```
git rebase -i HEAD~4
```

and then, just delete (or comment out) the lines corresponding to the commits you wish to delete, like so:

```
pick 2f05aba ... # HEAd~4 will be preserved
#pick daed25c ... # HEAD~3 will be deleted
pick 3371cec ... # HEAD~2 will be preserved
pick e2b2a84 ... #HEAD~1 will be preserved
```

You can also replace "`pick`" with "`drop`". To redo that commit, change "`pick`" to "`edit`". When you leave the editor, the bad commit will have been checked out so you can try to fix whatever it did wrong. Then add and commit to finish the rebase.

To leave unfinished work in a branch and switch to another branch, use the command "`git stash`". Run "`git stash list`" to see a list of your saved but

uncommitted stuff. When ready to resume work, run "`git stash apply` *`stash-ID`*" and your current directory is back where you left off.

Best Practices to Avoid Mistakes with Git (adapted from Carla Schroder)

Follow the conventions of the project you are contributing to. Each project should have written style and process guides to follow.

Keep your commits small, specific, and frequent. Make detailed commit messages. A good commit message describes what your commit does, and why. Start with a brief descriptive subject line followed by a line break. (The first line is often shown separately.) In the body of your commit message, describe what the problem your commit is solving (e.g., improvement, new feature, or bug fix, code improvement). Cite relevant issues (these come from a trouble-ticketing (a.k.a. issue tracking) system) or other related commit IDs. It is good to list any bug fixed with a standard comment you can easily grep for, say `Bug-Fixed:` followed by the bug ID from the issue tracker database.

> Despite best intentions, files often end up with multiple responsibilities. Sometimes you get into "the zone" and make many changes that should be in separate commits. One way to deal with that is by using git stash as discussed above, but there is another way I find easier.
>
> Once you realize you need to make two or more commits on a file or files with multiple changes in them, you can use **`git add -p`**. This command will launch an interactive editor window, showing each change in patch format. For each separate change (*hunk* is the term used with patches), you can say yes to include or no to exclude. Only the included changes will be included in the staged file! Now you can commit that. Repeat for each set of changes you want as separate commits.
>
> See the `git add` help page for more details and options.

Always keep the main branch HEAD as good, working code. Any commit leaving the code in a non-working state should be done in another branch only.

Best practice is to always have commits leaving the HEAD of all branches in a working state. Commits should be very small (a few lines) and it's fine to have lots of them.

Avoid long-lived feature branches. Having many branches will require lots of merging whenever a bug fix is introduced. The repo may become so complex it is hard to work with. Instead, keep feature branches short-lived. When done with one, merge it into the main branch ASAP. This style of development (with no long-lived branches other than main) is called *trunk-based* development.

Trunk-based development is considered by many to be a *devops* best practice for services. New features and other large or disruptive changes are made by having

feature flags, allowing selective enabling of features while the service is live. (Facebook and Google do this.) If you don't use devops or don't use a micro-service architecture for your software, feature branch development will likely work as well or better.

> A **feature flag** is a software development process used to enable or disable functionality remotely without deploying code. New features can be deployed without making them visible to users. See launchdarkly.com for an overview. Having too many such flags can cause problems!

Never work directly in your upstream branch. Always make your own working branch, or even your own GitHub clone, then locally clone that. Remember to always run `git pull` on your main branch before creating your working branch.

Suppose that while you were working away on your own "feature" branch, made from the (then) current HEAD of the main branch, a number of commits were made to the main branch. There are two ways to incorporate those commits into your current branch: merging and rebasing. You can merge in those commits (from the main branch) as you go. The result is often a complex tree for your branch's history, but everything is preserved.

Rebasing

Instead of merging, you can ask Git to ***rebase*** your feature branch off of the now current HEAD of the main branch.

> When you create a branch, the commit you started from is the branch's *base*. What rebase does is to move the whole branch to some other base commit, usually the new HEAD. So the new history appears as if you created the branch later than you really did. As if that wasn't enough, while rebasing some commits on a branch, you can *squash* them into a single new equivalent commit. So rebase is a powerful tool to clean up your local repo's history, before sharing code with outers. (Once commits are shared, doing a rebase on them is very dangerous!)

Rebasing edits the history of a project. In the common case, a rebase command replaces all your commits with new ones, based on the new main branch's "tip" (or some other specified commit). Suppose the main branch is on commit X, then you create a new branch feature and make two commits A and B on that branch. Meanwhile, a new commit C was added to main. When you rebase `feature` onto `main`, Git determines the patch (the differences) from X to A, and replays those changes on C, creating a new commit object for the result, D. Next Git determines the patch from A to B, and applies it to D, resulting in a new commit, E.

From X to E is a linear history with no forks (diamond shapes in the Git history). However, the end result is the same code as if you had merged feature onto main. If you had merged instead, the one new commit would have had two parents, B and C.

You can even merge ("squash") together some of the commits into a single commit (the patches from each are combined), cleaning up the history (say when you were fixing typos in three successive commits). This "lie" loses information about when the various changes were actually made, but results in a clean, linear history that is much easier to review:

```
git checkout feature
git rebase main
```

Here we've replayed the new commits from main onto the tip of feature. That may involve resolving a conflict (just as with merge). But when done, you can now:

```
git checkout main
git merge feature
```

There will be no conflicts and no "diamond" in the history.

Git supports interactive rebasing as well, allowing you to tweak the commit messages, condense (squash) two or more commits into one, and preform any needed merging. See www.atlassian.com/git/tutorials/merging-vs-rebasing for a nice tutorial on this. (*See rebase demo in the class resources.*)

It is common to squash commits on a branch before creating a PR or merging that branch into main. When working on a feature, nobody cares how many breaks you took or how many commits you made during the work. It is common to add "WiP" (*work in progress*) to the commit message to indicate this status of the commit. But reviewing many intermediate commits is difficult and annoying, compared to one all-inclusive commit for the new feature/change.

When ready for review, you squash all the commits you made when working on a single issue (that is, all the commits in one feature branch) into one commit, then push that branch (or create a pull request from that), for code review.

To squash commits using rebase:

- Run `git rebase -i HEAD~5`. This will open an editor window with the rebasing script in it showing the last 5 commits, such as this:

```
...
pick f344171 Added new feature X
pick ba20d9a Finished method foo WiP
...
```

```
pick df710b6 Unit tests for feature X WiP
```

- Change the word `pick` to `squash` for each commit you want to merge into the previous one. (Leave the first one alone; the subsequent squashed committed are rolled into that commit, making a new commit). Leave the commits you've already shared (or want to remain as separate commits) alone. Save and quit from the editor. **Never rebase commits you have pushed or otherwise shared!**
- When done, an editor should open to allow you to enter a commit message for the new commit. When done, save and close the editor.

That's it!

Fixing Mistakes with Git

If you haven't yet made a commit, just fix and then add the file(s) again.

If you've made a commit but haven't pushed changes to the main branch or made a pull request, you can run a number of commands to fix things up. However, **running these commands after a push is very dangerous and most certainly will cause problems in the main branch.** If you've created a pull request that hasn't yet been accepted, delete (cancel) that pull request, then you can safely use these techniques before making a fresh one.

If you discover a mistake after making a commit, just fix the mistake and then commit again. You can make as many commits as you like before creating the final pull request. However, a pull request made up of a large number of commits is more difficult to review and debug. Git includes commands, such as `rebase`, that allow one to "clean up" a large number of commits into a much smaller, simpler set.

You can **amend the last commit**. This replaces the previous commit instead of adding another one. Make and `add` your edits, then use the `--amend` option like "`git commit --amend`". This action overwrites the previous commit, and you can change or keep the previous commit message.

If you've already shared the commit with others, you cannot do this safely! The amended commit is actually a new commit with a different ID. Pushing it causes Git to notice the conflicting edits, and thus the push fails. The safest way to proceed is to create another commit with the corrections and leave the bad commit alone. See StackOverflow for more information on this. Note this applies to most changes you make; don't do it if you've already shared your changes.

You can un-commit your last commit with "`git reset --soft ^HEAD`". This doesn't change any of your files in your working directory tree, but rather

goes back in time as though your last commit did not happen. (Also restores the staging area.) Just fix up your files, add them, and then commit again.

To undo the effects of any previous commit, run "`git revert` *`commit-ID`*". If there have been many commits since that one, you will probably have to manually merge the result. If that looks too difficult, you can undo the pending revert with "`git revert --abort`".

Using "`git reset --hard` *`commit-ID`*" causes Git to discard all work since *commit-ID*. Use this with extreme caution (and have backups)!

Another, safer way to undo changes is using "**`git revert`**". Unlike `reset` (which alters the history by deleting commits), `revert` undoes changes by making a new commit to restore the HEAD to the state of some prior commit. (That is, the new commit un-does the changes by previous commits.) This way, you don't actually lose anything. Because it is safe, you can use `git revert` on commits you have shared. Sadly, you can't revert a single file.

You *can* reset or checkout individual files, as well as whole commits as shown previously. The command

```
git reset HEAD some-file
```

will unstage (undo the git add of) *some-file.* The working directory copy of the file is untouched by this.

To update the working directory with an older version of some file, use

```
git checkout HEAD~1 some-file
```

This command replaces the version of the file in the working directory with the version from the previous commit.

The `git checkout` [*`file...`*] command simply restores the files in the working directory to the version in the staging area. The `git checkout` *`branch`* command updates both the staging area and the working directory from the files in that commit.

Using Git Bisect

It happens sometimes that an error isn't discovered immediately, but only much later. If you don't have small commits, you may need to look through a small number of commits but each could be dozens or hundreds of lines long. If instead you follow best practice and make many small commits, the error could be in any of dozens or hundreds of commits.

One common (but bad) practice is to patch around the bug, never discovering when it was introduced. (Qu: Why would this be bad?) A better way is to figure out how to test for the presence of the bug, and check-out old commits and test them, until you find the one that introduced the bug. Git provides an interactive tool to

make this fairly easy, `git bisect`. You do not need a linear history to use this feature. This process can even be automated.

To start, issue the command `git bisect start`. You will see your prompt changes to show you are in a bug-hunting mode. You now tell Git that the current commit is bad with the command `git bisect bad`. (You can specify a different commit too.) Next you need to tell Git about an old commit you know doesn't have the bug; it might need to be your initial commit. You do this with `git bisect good` *`commit-id`*. (You can omit the commit id if you check out that comment first.) Let's assume there's 250 or so commits between them, any one of which might be where the bug was introduced. Not to worry! Git will tell you that you will find the bad commit is about 8 steps!

Git will figure out the commit that's half-way between the known good commit and the bad one, and check it out for you. Now you test for the bug. Let's say this commit is good and thus the bug was introduced in a later commit. You tell Git that with `git bisect good`. You're now down to about 125 commits to check. (Notice this is a binary search.) Git will now calculate a new commit to check, again half-way between the known good one and the bad one. When you find a bad commit, use `git bisect bad` instead.

After at most eight tests (in this example), Git will tell you the first commit with the error. That's the one to fix!

If you can create a script to test your code, you can run the bisection automatically with `git bisect run` *`myscript`* (instead of `git bisect start`). *Myscript* should exit with success (0) if the commit is "good" and non-zero (1) if bad. (Your script should return exit status 125 if the current commit is untestable for some reason. Git will skip it and chose another commit to test. *Note! This is a good reason to never check in a commit that cannot be compiled or run!*)

You can see the changes made in the commit which introduced the error with the command `git show`. Hopefully it was a small commit and you can figure out the error quickly. Note, you can use `git bisect` to find anything, not just bugs. To process is the same, only instead of testing for the presence of a bug, you test for the presence of the feature of interest. To support this use, Git allows the terms `old` and `new` instead of `bad` and `good`.

When done, use `git bisect reset` to end the debugging session and to checkout the former `HEAD`. Git keeps a reference to the last bad commit found. You can use it to end the session and check out the bad reference with:

```
git checkout refs/bisect/bad
```

How to fix the issue found depends. You can use reset, revert, or rebase to address the issue, or just add a new commit leaving the older versions buggy. Fixing the

bad commit is best only if you never shared any of the commits after that with anyone. If you have released older versions of your project to the world, you may need to fix each released version (especially if security is involved) with a new commit. You should also add a test case the catches this bug to your test suite; that may not always be possible though.

Using Git with IDEs

IDEs have different user interfaces and defaults for Git. You can also install Git for Windows, and use its simple (but limited) GUI, or launch a simplified "Bash" (command line). I believe NetBeans creates the `.git` repo inside of the NetBeans project folder. Eclipse prefers to keep it separate, and just keeps track of where the actual repo is placed, for each project. (I don't like and don't use that default; my Git repos are in the top of my Eclipse project folders.)

When working with IDEs, Git, and GitHub, keep in mind you have your IDE project folder, a local repo, and the remote repo. In either IDE, Git's working directory is linked to (or is the same as) the IDE's project's working directory. The steps you'd need to use to work with a project from a remote repo are:

- Create a local repo as a clone of a remote one.
- Create a project in your IDE.
- Link the project to the local repo. Usually there are wizards to do that; since you don't generally share IDEs' files, you will need to launch a new project wizard (using existing source from the local repo).

Demo simple Git use with IDEs. Then demo GitHub use (create practice repo).

The basic operations of Git are easy and mostly intuitive when using IDEs. Additional add-ons to your IDE may be available to make the task even easier. However, it is very important that you carefully read the *Using Git tutorial* that comes with your IDE! Failure to spend the time working through the tutorial will result in frustration later. (Eclipse Git users: see synchronization view icons and repository view icons.)

Eclipse Special Case

There is situation that Eclipse doesn't handle well: importing a Java project with existing sources from a Git repo (such as GitHub), but no Eclipse "`.project`" file. Because of how Eclipse is implemented, you can either import an existing Eclipse project (the repo must have a correct "`.project`" and other Eclipse files in it which contains the project's name and other info), or create a General project then later convert it to a Java project. If you try to create a new Java project from the repo with sources, the import wizard fails one way or another (either no project is created, or the project has no files, or some other weird error). This is a long-time open bug with Eclipse Git, but there's nothing they can do without major

changes to Eclipse itself (which is unlikely to occur). (NetBeans does not have this issue.)

My suggestion is **do not try to import a repo with source unless it has a valid Eclipse project file in it.** Instead, create a brand-new Java project the normal way, then create a local git repo for it ("Team->Share..."), and add a remote reference to the Git repo "`origin`" pointing to your GitHub repo. Then just do a Pull. The result should be the same as if you did an import and it had worked. Here's the steps:

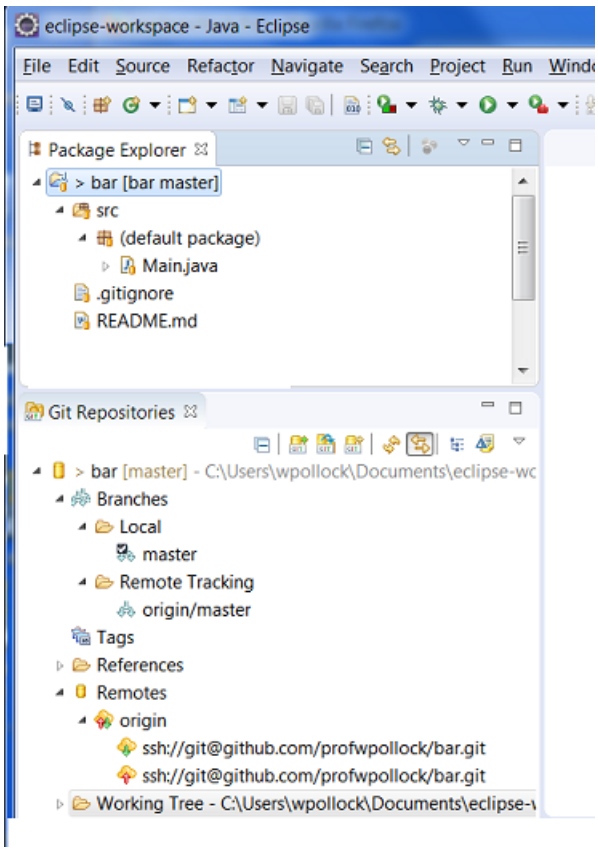
- **Create a new Java project in Eclipse**. Give it a name, then click "Finish".
- **Initialize a new Git repo for the project:**
  - Right-click on the new project and select "Team-->Share Project...". (Depending on what VCSs you have installed you may need to select "Git", then click "Next" to proceed.)
  - Click the checkbox for "Use or create repository in parent folder of project". (That's not the default since initially Eclipse was slow opening projects with large Git repos in them; that doesn't seem to be a problem anymore, and I prefer keeping all of a project in one place.)
    - Click on your project to highlight it, then click on the "Create Repository" button, and finally click "Finish".
    - Unfortunately, Eclipse creates a "`.gitignore`" file in your new repo's working directory when you do this. If the GitHub repo already has this file, your first merge will fail with a conflict. If the repo didn't have one, the initial fetch will fail. Just **delete the generated `.gitignore` file** by right-clicking the file in the "Package Explorer" and choosing "delete". Create the file later if you want one.
- **Connect your Eclipse Project with a GitHub Repo:**
  - In your web browser, go to the GitHub repo you wish to use. Click on the "Clone or download" button. Make sure the right protocol (HTTPS or SSH) is selected, then click the "Copy" icon next to the URI shown.
  - Back in Eclipse, right-click on the project and select "Team->Pull...". It should fill out everything correctly from the URI on the clipboard. If using SSH, you will need to set the protocol to SSH as well (since GitHub doesn't include that in the URL you copied).
  - Since your Git repo has no remotes defined, it will open the dialog to create one before attempting a pull. It should default to naming the remote "`origin`" (that's what you want), and default to using the URI on the clipboard. Now click "Next".
  - Since the information you entered on the last screen for the remote "`origin`" isn't saved for some stupid reason, you must create it again. Click the "New

Remote...” button, and fill in the dialog the same way as before. Click “Finish” to return to the “Pull...” dialog.

- Back at the “Pull...” dialog, click the checkbox for “Configure upstream for push and pull”, which should save this remote (named “`origin`”) as the default for pushing and pulling, and create remote tracking branches for each pulled branch automatically. Finally, click “Finish”.

> Note you can define other remotes the same way. For example, you can create a reference to your own repo on your GitHub account (and the group’s repo), or to a teammate’s repo. to share commits.

If the pull succeeds, you have a working Eclipse Java project with cloned source and all the remote references working. Your Eclipse setup should look something like this:



> To use Meld as the merge tool with Eclipse, follow these directions from stackoverflow.com.

If using a Maven project (recommended), you can run the command “`mvn eclipse:eclipse`” to add the correct Eclipse files, based on the info from your `pom.xml`. Then it is easy to import the project.

In-class program: Create a stand-alone non-GUI Java program that displays a 15x15 diamond pattern, using spaces and stars ("*"). Use for-loops and possibly if-statements. Show sample output. Time limit: 25 minutes.

Now that you know some of the non-coding steps and tools used in software development, it's time to discuss more Java! In your projects, use what you have learned here: Git, Maven, OWASP, check licenses, etc. Practice is the only way to learn this stuff, and these skills apply to all software development, not just Java.

Lecture 3 — Exceptions (*Back to Java-specific topics*)

Java code that detects a problem (such as the standard library code in the JRE) reports the error by **throwing an `Exception`**. If you do nothing, the `Exception` will abort your program. Exceptions are used to change the flow of control when unusual events (like an error) occur. Exceptions can happen when the system detects the unusual condition, or by a **`throw`** statement (used when your code detects a problem).

Exceptions are classes that extend **`java.lang.Throwable`**. Only objects of classes that extend `Throwable` can be thrown (or caught). Some `Exceptions` are called ***checked Exceptions*** (as opposed to ***unchecked***) and the compiler refuses to allow your code to ignore these. You must either (a) surround the code that might throw the checked exception with a try-catch block to handle the exception, or (b) declare your method `throws` this checked exception. Checked exceptions are for unusual conditions that can be anticipated, such as an `IOException` when reading from a flash drive and the drive is ejected. Here's an example of a try-catch block:

```
try { dangerous code }
    catch ( AException e ) { handler code for A }
    catch ( BException e ) { handler code for B }
    catch ( CException e ) { handler code for C }
    finally { this code always gets run }
```

If the *dangerous code* runs without problems, no handler code is used. On the other hand, if executing some statement in the *dangerous code* block causes a *BException* to be thrown, the rest of that block is skipped and the *handler code* for exception B is run.

Note the **`finally`** clause. **If present, that code will be run whether or not the dangerous code throws an exception.** This is a good place to put any cleanup code (for example, close open files). (See *automatic resource management*, or *try-with-resources* below.)

You can have a try-finally block with no catchers, try-catch blocks with no finally, or you can use both. (You must have at least one clause however, except as noted below.)

Here's an example of declaring the method throws some checked exceptions:

```
public void aMethod ( int anArg )
                    throws AException, BException
{ ... dangerous code ...  }
```

Here's an example of throwing an exception:

```
public void aMethod ( int anArg ) throws AnException
{  ...
   if ( something bad )
      throw new AnException("Explanation goes here");
   ...
}
```

Unchecked exceptions include ***RuntimeException**s* and ***Error**s*. Runtime exceptions can happen anytime and you don't need to try to catch them, but you can if you want. An example is a divide by zero error, or an array index out of range error. `Error`s are thrown when something serious happens that can't usually be handled, such as an internal JVM error.

The Java docs shows the exceptions listed the `throws` clause. These will include all checked exceptions, but may include some notable unchecked exceptions too. Show in Java docs the declaration of `BufferedReader.readLine()` and `Integer.parseInt()` (unchecked!) method.

Checked exceptions will be automatically documented by `javadoc` (since they're listed in a `throws` clause) but unchecked exceptions that are not listed, are not automatically documented. **You must document unchecked exceptions in the doc comments for your methods with "`@throws` *exceptionName comment*".** These are part of the public interface and thus must be documented.

Checked Versus Unchecked Exceptions, and When to Document Unchecked

Unchecked exceptions represent problems that are the result of a programming problem detected in your library ("API") code, and as such, the **API client code cannot reasonably be expected to recover from them or to handle them in any way.** Such problems include arithmetic exceptions, such as dividing by zero; pointer exceptions, such as trying to access an object through a null reference; and indexing exceptions such as attempting to access an array element through an index that is too large or too small. Other examples include cannot open some file or some record is not found.

In general, when a low-level method encounters any issue it cannot or should not handle internally, it should throw an unchecked exception.

One case where it is common practice to throw an unchecked exception is when the user calls a method incorrectly. For example, a method can check if one of its arguments is incorrectly `null`. If an argument is `null`, the method might throw a `NullPointerException`, which is an unchecked exception. In such a case

the unchecked exception is a part of the API definition (or *contract*) and should be documented in the Java docs for your API.

> Often, passing in `null` arguments or returning `null` is considered a bad practice and forbidden by the developers. In that case using an assertion to do a null check might be a better choice. Using the checker framework and annotating methods and parameters as `@NonNull` is another way to go. (`@NonNull` is actually rarely used as it is the default in the checker framework! You need to use `@Nullable` to indicate something is allowed to be `null`.)

There is no requirement to document unchecked exceptions (that are not part of the API's contract), and expert opinion is divided on whether or not to do so. After all, most thrown unchecked exceptions represent programming errors in the library code. I believe **all explicitly thrown exceptions should be documented.**

Check exceptions cause a problem in OOP: they break encapsulation! If a checked exception is caught and handled in main, but thrown five levels lower, all the methods between need to be updated to include the throws clause. For this fundamental reason, best advice today is to avoid using checked exceptions unless security or other considerations override clarity of code.

Exception Usage

Exceptions are handy to throw when a problem is detected but that code doesn't know what should be done about it. Reusable component classes in a library will do this. An example is the `Integer.parseInt(String)` method. Suppose the argument is not a valid number, what should this method do? (Pop up a GUI alert? Send a text message to the console? Send a log message? Rollback some database transaction in progress? Do nothing? ...) It depends entirely on the application that this method is used in! The method just throws a `NumberFormatException` and lets the application code catch and handle the problem.

In your own code, similar issues arise. You might have many methods that need to check input values (preconditions):

```
void foo ( int n ) {
if ((n < MIN_VAL) || (n > MAX_VAL)) {
    throw new IllegalArgumentException(...);
}
```

This sort of *range check* is common. (The *checker framework* includes range-check annotations for parameters.) If you use range checks in your code it may make sense to create a method for it:

```
private void rangeCheck ( int n, int lowLimit, int highLimit, String error ) {
  String format = "%s: %d is outside range %d - %d";
  if ((n < lowLimit) || (n > highLimit))
     throw new IllegalArgumentException(
          String.format(format,error, n, lowLimit, highLimit));
}
```

An advanced technique when throwing exceptions from a check method such as the above, is to adjust the stack trace of the exception so it shows the correct location in your code. Simply put your range check function call in a try block to catch the thrown exception and edit the stack trace:

```
try { rangeCheck(...); } catch ( final Exception e ) {
  final StackTraceElement[] st = e.getStackTrace();
  e.setStackTrace(Arrays.copyOfRange(st, 1, st.length));
  throw e;
}
```

Application code (main and the UI code) should *never* throw an exception. (Who do you think would handle the exception?)

Note that **uncaught exceptions will terminate the current `Thread`** (after printing a stack trace). (**Show `Thread.setUncaughtExceptionHandler()`.**)

Application code should use try-catch blocks to handle all exceptions it can. It *must* handle all checked exceptions. (You can cheat and declare `main` to `throws Exception`.)

One original motivation for using exceptions was to make code more readable, by separating the error handling code from the main flow of code. In languages without exceptions (rare today), code is often an unreadable series of function/method calls, followed by one or more if-statements to deal with errors:

```
error = A();
if error == A1, do this;
if error == A2, do that;
...
error = B();
if error == B1, do this;
...
```

Every function/method call needed code like that.

With exceptions, you can put in the main flow (often called the *happy path*) in a try block, and have all exception handling follow. That should make code easier to

read. (This matches how we often develop code: start with the happy path and add error handling later.)

In practice you still may need to do error handling per statement. Only now the code looks like this:

```
try { A; } catch (A1 e) { ... } catch (A2 e) { ... } ...
try { B; } catch (B1 e) { ... } catch (B2 e) { ... } ...
```

Such code is often seen when library code throws the same exception for all problems. So a catch block at the end of a series of method calls would not know which method call caused the exception.

Often, all exceptions from some method A or B get handled the same way:

```
try { A; }    catch (A1 e) { loger.log("A1 Exception", e); }
              catch (A2 e) { loger.log("A2 Exception", e); }
              ...
```

In such cases, the specifics of the exceptions can then simply be in the exception message and thus don't need specific types of their own. You can simply define A1, A2, and so on as extending (say) AException. Then the above code simplifies to:

```
try { A; B; } catch (AException e) { loger.log(e.getMessage(), e); }
```

For code you cannot refactor this way, you can just create a wrapper class that does A, catches the specific exceptions thrown, and convert them to a common exception type.

> **Define exceptions in terms of the caller's needs.** When you create non-application code (let's call it *library* code), you often create interfaces and abstract classes too. Consider creating one or more new exception types that would be useful to the users of your code.

When you create library code, you should check for and throw any exceptions that an application using this code might be able to handle. A common example is `IllegalArgumentException`: some method takes an `int` parameter but the value should be in a certain range only, or some object parameter shouldn't be `null` (although there is a runtime `NullPointerException`, you shouldn't use it here.) Here's an example:

Suppose you have a zoo application and you have an abstract `Animal` class with a `feedMe` method that causes the animal handler's text pager to display an appropriate message. But it is an error to invoke `feedMe` for an animal in the hospital. What should `feedMe` do if invoked when the animal is in the hospital?

Well, it shouldn't happen but it might. The answer is to create a new exception class (to be included in the same package as class `Animal`), perhaps

`AnimalNotAvailableException` (it is customary to name all exceptions as `SomethingException`). This might be either checked or unchecked depending on your overall design. (Remember that checked exceptions extend `Exception` and unchecked exceptions extend `RuntimeException`.)

> Current best practice is to use only unchecked exceptions (see below). But if you don't agree with that, you can follow this rule: If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.

Common idiom: often you don't merely wish to exit gracefully, you wish to retry the block:

```
login: for (;;) {  // infinite loop, or allow N retries
    try {
        getPassword();
        break login;  // exit loop if no problems
    }
    catch (BadPasswordException bpe) {
        System.out.println( "Try again!" );
}   }
System.out.println( "Login successful!" );
```

Another common anti-pattern (code smell) by newbies is to put a loop inside a `try` block, throwing an exception to exit the loop. However, this is a bad idea! Just use `break` or `return`.

Since Java 7 an improved version of try and catch is available. If the handler is the same for many different types of exceptions (but not all or you just catch `Exception`), you can group all the similar ones in a single catch clause:

```
  try { ...
  } catch ( final ClassNotFoundException
            | InstantiationException
            | NoSuchMethodException
            | NoSuchFieldException
            | IllegalAccessException
            | InvocationTargetException e ) {
     log(e); throw e; // the type of e depends on the
                      // specific exception thrown.
```

```
}
...
```

Summary

Exceptions are the standard way to handle potential problems in your code. You throw exceptions from reusable code (even when you reuse it yourself) and catch exceptions in your higher level code (such as main method). You can define your own exceptions, often useful when defining library code that others must call from try blocks.

Exceptions are not the only way to deal with error handling in Java. These other mechanisms (and some enhancements to Exceptions) are discussed next.

System.exit(), finally, finalizers, Cleaner, Shutdown Hooks, and ARM (*try with resources*)

What is printed by the following code?

```
public static void main (String [] args )
{  try { System.exit( 0 );
   } finally { System.out.println( "bye!" ); }
}
```

The answer is nothing is printed! The reason is that `System.exit` is not like `break` or `return`. Instead it causes the JVM to shut down at once. The `finally` clause never gets a change to run! Because of this, using a try block with `finally` doesn't always work to ensure some code always gets run when the JVM shuts down.

Initially, the Java designers thought the answer would be to override the `protected void finalize` methods for your classes. These get run by the garbage collector, but when shutting down, the JVM may not run GC. To try to guarantee these get run at the end, you could in theory execute `System.runFinalizersOnExit(true)`, or the equivalent `Runtime.getRuntime().runFinalizersOnExit(true)`. **It was realized later that this is very unsafe with multi-threading and is now deprecated. (*Show in Java Docs:* `Object.finalize()`*.)***

To guarantee some code will run when the system shuts down, you add a `Thread` whose `run()` method contains the code to run, as a *shutdown hook* for the current JVM, or **`Runtime`**:

```
class CleanUpHook extends Thread {
   public void run () { ...clean up code... }
}
class SomeClass {
```

```
    public void main ( String [] args ) {
       Runtime jvm = Runtime.getRuntime();
       jvm.addShutdownHook( new CleanUpHook() );
       ...
    }
}
```

A shutdown hook in Java is just a ready-to-run thread. Here I used a named class, but you can create one using an anonymous class or a lambda:

```
jvm.addShutdownHook( new Thread(() -> {...} );
```

Shutdown hooks are run even after `System.exit()` or after an external event occurs (such as hitting control+C). (To truly crash the JVM programmatically, you need to run `Runtime.halt()`.)

Since the JVM is in a delicate state when turning off, shutdown hooks must be lightweight (execute quickly), thread-safe, and not rely on system services (which may have shut down already) such as the AWT Event Dispatch Thread. (So no updates to the GUI can be done.) Note that you can have many such hooks, and they all run concurrently. (There is also a `removeShutdownHook` method, should you change your mind after adding one.)

To use a shutdown hook correctly (safely), you need to pause the shutdown until the rest of the program is at a safe stopping point, then shut down. Try this technique:

In the code that updates stuff (say the main thread in a simple app), modify the loops that do work to include testing a `boolean` flag to at a safe stopping point:

```
while ( ... && keepRunning ) {
    doSomething();
```

In the `main` method, you need to create that `boolean` flag correctly and initialize it:

```
static volatile boolean keepRunning = true;
```

This flag is set to `false` by the shutdown hook code. (If there is the only one shutdown hook changing the flag, then this is sufficient. Otherwise, use `AtomicBoolean`.)

Since the shutdown hook is a thread, you will need to pause it until the main code finishes its current task (you generally don't want to abort in the middle of a task such as a disk write, for example). To do that, you will need a reference to the thread that is doing the work in a loop, so add this to `main` as well:

```
final Thread mainThread = Thread.currentThread();
```

Finally, you can add the shutdown hook like this:

```
Runtime.getRuntime().addShutdownHook(
   new Thread( () -> {
      keepRunning = false;
      mainThread.join();
      // Do anything else you want here,
      // for example, saving state to storage
   }
);
```

This code sets the flag to `false` so the work will stop as soon as the current task finishes. Then the shutdown hook waits for that to happen. Finally, the code can do any steps it wants, such as persisting in-memory data to a file or database.

> Robert "Uncle Bob" Martin says that when writing code, write the `try` blocks first. A simple example he provides in *Clean Code* is writing a method that opens a file and reads a record of data from it. If the file cannot be read, or the requested record cannot be read from the file, the new method should throw exceptions.
>
> He starts by writing failing unit tests that invoke the (not yet written) method with arguments that should cause exceptions to be thrown. Next, he creates a stub of the method so the tests compile and run. Naturally, such tests fail since the method stub throws no exceptions. Next, `try` blocks are added in which the access code will be added later but for now throw appropriate exceptions. The test cases for that should now pass.
>
> Now that this is done, test cases for the non-exceptional case are written, which of course fail for now. Then code can be written inside the `try` block to attempt access for real without worrying about such errors.
>
> (When playing with code for learning or experimenting, I don't do this. Exception handling is ignored when exploring solutions by writing code. But for production code, I think Martin has the right idea.)

Try-with-resources:

Another Java 7 addition is **try-with-resources**, also called *automatic resource management* (or ARM). This is meant to simplify the correct way to clean up open resources such as open files, open network connections, open DB connections, etc.

In the common case of needing to run some code that "opens" a resource, and that must close that resource and the end of that block of code, the standard way is to use a `try` block with a `finally` clause. However, a problem was identified with this "idiom", namely that if the `try` block exits with a thrown exception and the code in the `finally` clause also throws an exception, then one of those exceptions gets lost!

The new Java 7 way allows you to initialize (or "open") resources in a block of code, which will automatically be closed when the block exits. (To support that, a new interface `AutoCloseable` was designed for files and other resources that can be closed automatically, and a way to *suppress* exceptions. These topics will be discussed later.)

Here's an example that works correctly, in both pre-Java 7 and in Java 7, that copies one file to another:

```
public void copy ( String src, String dest )
            throws IOException {
  InputStream i = new FileInputStream( src );
  try {
   OutputStream o = new FileOutputStream( dest );
   try {
    byte[] buf = new byte[8192];
    int n;
    while ( (n = i.read(buf)) >= 0 )
       out.write( buf, 0, n );
   } finally { o.close(); }
  } finally { i.close(); }
 }
```

That's pretty complex! With the new try-with-resources construct, the opening code is put in parenthesis. This simplifies the above code to:

```
public void copy ( String src, String dest )
            throws IOException {
  try (InputStream i = new FileInputStream(src);
       OutputStream o = new FileOutputStream(dest)
  ) {
      byte[] buf = new byte[8192];
      int n;
      while ((n = i.read(buf)) >= 0)
        o.write(buf, 0, n);
    }
}
```

This works because `InputStream` and `OutputStream` implement the new **`AutoCloseable`** interface, which defines the method:

```
void close () throws Exception
```

This method is called automatically on each "resource" object listed; they get called in the reverse order the objects are created. The resources in the parenthesis are a semi-colon separated list of `AutoCloseable` objects.

Here's a simple example (in resources):

```
public class OpenTheWindow implements AutoCloseable
{ public OpenTheWindow ()
  { System.out.println( "The window is Opening." );
  }
  public void close ()
  { System.out.println( "The window is Closed." );
  }
```

And here's a trivial example use of this class:

```
  public static void main ( String [] args )
  { try ( OpenTheWindow win = new OpenTheWindow() )
    { System.out.println( "Window is open." );
    }
  }
}
```

Using try-with-resources, you don't need to have any `catch` block or `finally` clause (the exception (pun intended) to the rule that you must have at least one of those). If you do have those, they are invoked *after* the `close` method(s).

Try-with-resources is a great addition in Java 7 that you should use frequently, so make sure you are familiar with how it works.

For those times when try-with-resources won't work for you, you can use the `Cleaner` class, introduced in Java 9 as a replacement for finalizers. While better than finalizers, class Cleaner still has some issues (such as it is implementation defined if any cleaner runs after `System.exit` is called). To use, you need to register a cleaner for an object. When the object becomes *phantom reachable*, that is, when the garbage collector is mostly done with it, a new thread is started to run the cleaner you registered. Note the cleaner code must not try to reference the object being cleaned (as it has already been partly destroyed. Note you can use Cleaner actions with try-with-resources: if the close method isn't called for some reason, the cleaner action will be, unless you used `System.exit`.

Summary: When you need to clean up a resource as soon as you are done, implement `AutoCloseable` by putting the cleanup code in a `close` method. Try-with-resources will call the `close` method automatically. Otherwise, you need to put cleanup code in a try block's `finally` clause. (It isn't a bad idea to put any cleanup code in a `close` method, even if not using try-with-resources.) If updated old code, or for extra safety, you can register a cleaner to run when the object is garbage collected. If you need to ensure the resources are closed when the JVM does, use a shutdown hook.

Suppressed Exceptions

If an exception is thrown from the `try` block, the `close` method gets called before the exception handler code or the finally block. So, what happens if an exception is thrown from within the `close` method? The original exception hasn't been handled yet.

If an exception is thrown from the `try` block and one or more exceptions are also thrown from the `close` method(s) called, then those exceptions thrown from the `close` method(s) are *suppressed*. That means Java will remember them but not do anything about them. You can retrieve such suppressed exceptions by calling the `Throwable.getSuppressed` method from within the `catch` clause for the original exception thrown.

Additional Details

All exceptions and errors are specific types of **`java.lang.Throwable`**. Only `Throwable` objects can be thrown or caught in a try block. `Throwables` have some useful methods use in a catch clause, such as `printStackTrace()` and `toString()`.

A common error is to declare local variables in a try block and attempt to use them later.

You can create your own exception classes easily, by extending existing exception classes:

```
public class MyException extends Exception {}
public class EOFException extends IOException {}
public class UncheckedException extends RuntimeException {}
```

Often you will add a `String` or other properties to your exception class, with methods that can be used in the `catch` block.

Java 1.4 added new features to `Throwables`, to allow you to ***chain exceptions***. This is handy when some inner code throws one type of exception but the public interface of your class doesn't allow that type of exception to be thrown. In this case, you chain the original exception (called the ***cause***) in another exception type:

```
public void pubMethod (int anArg) throws PubException
{  try {
      privateMethod( ... );
   } catch (PrivateException cause) {
      throw new PubException( cause );
   }
}
private privateMethod ( ... ) throws PrivateException
{ ... }
```

Note that `cause.toString` is automatically invoked above as needed.

Order of catch blocks

Exceptions and errors are arranged in an inheritance hierarchy. When some exception is thrown, the first catch block that can match the type, will. So you always need to arrange catch clauses for polymorphically similar types **from most specific to most general**:

```
try { throw new AccessControlException("...");  }
catch (AccessControlException e) { ... }
catch (SecurityException e) { ... }
catch (Exception e) { ... }
```

If you reverse the order of these three catch blocks, only the `Exception` handling catch block can ever be used; the other two become unreachable code.

Demo ExceptionDemo.java (Shows throwing and catching exceptions).

When to Create New Exceptions or Reuse Existing Ones

Qu: I am reading a file and checking each piece of data. If there is something wrong with the data, I do this:

```
throw new XException( "Error at row = " + ... );
```

What should "*X*" be? How does one decide if this is the sort of thing that should have its own type such as `BadDataException` or use some existing type like `IOException`?

The main problem for reusing rather than creating new exception classes comes when we need to handle the exceptions differently depending on where they occur. (This was mentioned previously, and the ugly solution was to put each method call in its own try block). For example, how would the calling code be able to differentiate between the `IOException` stated above, and any other `IOException`, like `FileNotFoundException`, etc.? By reusing a single exception type for multiple reasons, we lose the class type differences, which prevents having specific catch blocks and so specific handlers.

As an illustration, consider developing a `Model` class from which data could be Added/Edited/Removed, and each bit of data has a unique identifier. In this example, the model throws exceptions when:

1) adding of data if the identifier has already been used,

2) editing data when the identifier does not exist.

3) Removing data when the identifier does not exist.

So, what types of exception(s) should be thrown? Should we reuse and existing standard exception class, invent one new one, or have several new ones? Should the exceptions be checked or unchecked?

There is no answer that is best in all cases. In this example, I believe all three methods should throw specific kinds of exceptions to show the intent of their interface, rather than a generic `ModelException`. However, all exceptions can extend an `abstract ModelException`. That way, callers have a choice of handling each type of exception differently, or just catch the generic intermediate class. Here's an example of this idea:

```
class Model
{ void add( Id id, Object data ) throws
                IdentifierAlreadyExistsException
  { if ( contents.containsKey(id) )
    { throw new IdentifierAlreadyExistsException(id); }
    ...
  }
 void edit ( Id id, Object data ) throws
                NonExistantIdentifierException
 { if ( ! contents.containsKey(id) )
   {  throw new NonExistantIdentifierException(id); }
 }
 ...
}
class abstract ModelException extends RuntimeException
{ private Id id;
  ModelException ( ID id, String message ) {
     super( message );   this.id = id;
  }
   Id getId () {
      return id;
   }
}
class NonExistantIdentifierException
          extends ModelException
```

```
{ NonExistantIdentifierException ( Id id )
  {  super(id, "no data for this Identifier exists");
  }
}
class IdentifierAlreadyExistsException
          extends ModelException
{ IdentifierAlreadyExistsException ( Id id )
  {  super(id, "data for this Identifier already "
         + "Exists");
  }
}
```

Checked vs Unchecked Exceptions (one last time)

In the example above, I made all the exceptions unchecked exceptions as is current best practice. However, sometimes (but rarely) checked exceptions are a better choice: The code in a self-driving car, military system, medical equipment, power plant, or online voting system should have the compiler ensure all possible errors are caught, so such code should throw checked exceptions even if there's nothing the caller might reasonably do to recover from the thrown error. (This is the exception—pardon the pun — to the advice given earlier, that check exceptions should only be used when the caller can reasonably recover.) **In such cases, safety and security override concerns for more elegant code.**

If the error won't cause harm or serious issues if uncaught, and if you are unsure whether or not there is a reasonable way for the caller to recover, unchecked exceptions are more convenient for the user of your code (the caller can catch those or not as they wish). In such cases, I think it wise to document all exceptions that a method explicitly throws in the method's doc comment (and the throws clause).

> The designers of Java have been criticized for making `IOException` checked. It is thrown in over 60 methods from the standard library, occurs rarely, and in most cases, there is no reasonable way to recover from the problem (the program simply must abort). There is a through discussion on this at Oracle.com effective exceptions.

Lecture 4 — Jar Files, Sealed Packages, Signed Jars, and Security

Jar Files

Jar file formats and `jar` tool (manifest, index):

Except for the addition of a ***Manifest* file** (and some other optional files), **a Jar file is just a zip archive**. ("Jar" for "Java ARchive".) Jar files have extra information in the manifest file, used by the runtime. For Java SE, the extra information is stored in a text file called the *manifest* file. Jar files are used to package Java class files, resources such as multi-media files, and even other Jars (rarely with Java SE though).

A minimal manifest file is created and maintained automatically by the `jar` tool. A Jar's manifest file should have the name `MANIFEST.MF` in a directory named `META-INF`. Some other files, such as indexes and digital signatures, may also be present in that sub-directory.

The automatically generated minimal manifest file `MANIFEST.MF`:

```
Manifest-Version: 1.0
Created-By: 24.0.2 (BellSoft)
```

(Note the ending newline. The `Created-By:` value shows the version and the vendor.)

With the right manifest setting, a Jar file can be made *runnable* (executable or double-clickable) by either double-clicking on it from some GUI or running the command "`java -jar` *`nameOfJarfile`*". (Note, double-clicking a non-GUI application won't show anything!)

Jars with specific folders and files in them are used with application servers (Jakarta EE containers) and are then called web application Jars or enterprise Jars. Such Jar files may have different extensions than `.jar`, such as `.war` or `.ear`. Jars are also used to hold *JavaBeans*, create ***signed classes*** (for extra security), and ***sealed packages*** (another security feature).

Jar files can contain packages and be listed on `CLASSPATH`. They also enable you to bundle extra resources with your code, such as graphics, sound, or fonts.

Jar File Manifest

The manifest file is a plain text file that ***must*** end with a newline (so be careful when using Notepad or similar editors that don't automatically add a newline at the end). It starts with a *main* section followed by sections for files, packages, or URLs within the Jar, for which you want to add entries (*attributes*). Each section is a list of *manifest entries* "`attribute: value`". All sections except the main (first) one start with a "`Name:`" entry.

To make a runnable Jar, you only need to add a single entry to the main section:

```
Main-Class: Name.of.class.with.main
```

(See `jar -e main_class`.) Some additional entries you may see or wish to use are:

- `Class-Path:`
- `Implementation-Title:`
- `Implementation-Version:`
- `Implementation-Vendor:`
- `Implementation-URL:`
- `Sealed: true`
- `Name: Some-name`
- `Content-Type:`
- `Java-Bean:`
- `SplashScreen-Image:`

Some of these items are *per section*, which override the same entry from the main section. See Using Jar Files and the Jar file specification for more details on the manifest file format.

To add entries to the manifest file, you put the additional entries into a text file (UTF-8 encoding works best I think), not forgetting the newline at the end, and then use `jar` to merge your entries into the (automatically generated) manifest.

The basic use of the `jar` tool (See jar tool JDK documentation for details) is:

```
jar -{c|t|x|…}[-v]-f file [-m file] [-e main-class] \
   files_or_directories_to_include…
```

Where you can use one action (`-c` to create, `-x` to extract, `-t` to view the contents, or others) followed by various options. Use `-m file` to specify a text file of additional manifest entries to include in the generated manifest file. Use `-f file` to name the Jar file. (*Show jar --help.*)

Jar files can hold media, but this can be tricky. For (deprecated) applet and WebStart standards, the `Applet.getCodebase()` returns the URL of the Jar file, not the URL of the media file within. So it won't always work to locate media

files relative to the codebase. You should use the **`getResource()`** method shown earlier. (Old code may not! Be careful when using example code from old books or the Internet.)

> `Class.getResource()` and `Class.getResourceAsStream()` use the method `Class.getClassLoader().getResource()` internally. `Class.getClassLoader` returns the `ClassLoader` that loaded some class. However, it is possible in some cases for this to return the wrong class loader! If that happens, your classes and media files won't be found.
>
> In general, to load resources that are contained in the Jars included in a `jnlp` file of a `jnlp` (a.k.a WebStart) application, or in Jar files listed in the HTML `ARCHIVE` attribute of an applet, you have to use the `JNLPClassLoader`, `PluginClassLoader`, or `Plugin2ClassLoader` that has the knowledge of those Jars. (Note that Applets with different `CODEBASE`s *may* use different `Classloaders` even for the same class; it depends on the browser.)
>
> A more reliable way to get resources is:
>
> ```
> URL url =
>  Thread.currentThread().getContextClassLoader()
>                .getResource(".../pic.gif" );
> ```
>
> (Fortunately, all this is deprecated and you shouldn't need to worry about applets or WebStart.)

As mentioned previously, to run a program in a Jar, you can double-click it or run from the command line as `java -jar` *`file.jar`*. In this case, **Java ignores the CLASSPATH** environment variable. It will only look in that Jar. You need to set the `Class-Path:` *`list`* entry in the manifest (main section) instead.

> The global CLASSPATH is an anachronism. It is impossible to have one CLASSPATH that satisfies all possible compilations and all possible applications. Installers will meddle with it. Programs will work, then mysteriously stop working after you update something unrelated that changes CLASSPATH. It is best to use the `-cp` argument to `java` and `javac`; IDEs use a per-project setting.

Jars within Jars

While it is easy to add any file including other Jars to a Jar file, it may not work as you wish. As mentioned previously, the global CLASSPATH is ignored for executable Jars. Only the manifest entry for `Class-Path` is used. **However, there is no syntax in Java SE for listing *internal* jars on that classpath!** (There

are no relative pathnames used, no way to say "`/foo/bar.jar` within this Jar".) Note this *can* be done for Jakarta EE. An installer program to add external library Jars to the user's computer in the locations listed in the `Class-Path:` is one reasonable solution. But what if you want a stand-alone application that contains all its dependent code?

It can be done, but not easily. A complete self-contained Jar is sometimes called an *uber-jar* or *fat Jar*. One way to create one is to extract each dependent Jar and just add the `.class` files to that single Jar. There are many third party tools for this; some tools I know of include the Eclipse plugin FatJar, OneJar (used by FatJar), Google JarJar, Maven Shade (what I use), Capsule, and the Gradle build system (See this blog post). Maven can be configured to use OneJar; you can also use a custom ant task (uses Apache Ivy).

> Self-contained `.war` and `.ear` files work, but again are not easy to create and manage. Oracle has released a tool that can create self-contained Java applications (but large) `.exe` files for any Java application. Called jpackage, it is included with their JDK since Java 8. The resulting executable include the required JRE parts as well as a JVM, so there's no need to have any JRE installed to use. See the Java Packaging Guide for more information.

Sealed Packages

Packages within Jar files can be optionally ***sealed***. This means that all classes defined in that package must be archived in the same Jar file. This is a safety feature that prevents developers using the same package name in their own Jar files, from accessing package-private members of your packages.

You might also want to seal a package to ensure version consistency among the classes in your software.

To seal all packages in a Jar, you need to add this entry to the main section of the Jar's manifest:

```
Sealed: true
```

You can seal individual packages (when you have more than one per Jar file), by putting that entry in one or more packages sections in the manifest.

> Do not confuse sealed packages with the Java 17 new feature, sealed classes.

Security Concepts: Encryption, Digital Signatures, Hashing, PKI

> A good part of this information was obtained from the free Linux Foundation's *Secure Software Development Fundamentals* program (at edx.org).

Developers must know a few basic issues about modern digital security. This is necessary to understand digitally signing code and data, verifying signatures, managing passwords, or using secure protocols such as TLS and SSH to access remote resources. Here, I will over-simplify a bit, but it should be good enough to understand what you need to do, and to understand the terminology used so you can communicate with others.

> All crypto depends on unpredictable random numbers. Sadly, the default *pseudo-random number generators* (PRNG) in most computer languages are not unpredictable. For almost all security-related tasks, you need to use a *cryptographically secure* PRNG (CSPRNG) for crypto and security-related tasks, as these are not easily predicted. You should only use random number generators for security purposes if it says it is a cryptographic PRNG - and typically, its name will have "secure" and/or "crypto" in it. **In Java, only the class `SecureRandom` should be used when security is important.**
>
> Java 17 and newer have added additional support for PRNGs. For `SecureRandom`, you can use an even more secure algorithm by using `SecureRandom.getHardInstance()`. A `random` package has been added that allows you to create random number generators using your choice of over a dozen different algorithms.

Single key encryption (or *symmetric encryption*) is the fastest way to secure data. This method is also known as ***private key*** or ***shared key***. A common example is AES (*advanced encryption standard*). With this method, a publicly known method is used to scramble the bits and bytes of any data in a systematic way. A *key* is used to determine the scrambling done. Without this key, there is no way to quickly recover the data.

Keys are very large random numbers, hundreds of digits long. Even knowing the method, an attacker who gets a copy of the scrambled information must guess the key to de-scramble it. This would take so long (hundreds to trillions of years), the data is considered secure. Yet, with the key, the intended recipient can quickly de-scramble the data.

Scrambling data is known as ***encrypting***, and de-scrambling is known as ***decrypting***. Unscrambled data is called ***plaintext*** or *cleartext* while scrambled data is called ***ciphertext***.

The shared key method suffers two drawbacks:

- If the key isn't chosen truly randomly, the attacker may be able to try more likely keys first and decrypt the data in very little time. (Passwords have the same issue.)
- It is required to securely share the keys between communicating parties. This is usually slow and expensive.

Most shared-key algorithms today work on small chunks of data at a time, called *blocks*. The popular AES is such a **block cipher**. Such algorithms have these parameters: key length (should be at least 128 for AES in 2021), block size (generally you cannot change this), and *mode*. There are many modes known for block ciphers. You only need to know that **ECB (electronic cook-book) mode is very insecure** and should never be used. Today **Galois/Counter Mode (GCM) mode is considered secure and fast**. It is parallelizable (to take advantage of multiple cores) and adds an authentication code so it can easily detect if the wrong key is used. It is a good mode to use. There are other good modes as well; the important thing is to choose a mode wisely and to never use ECB mode in production systems.

> AES is U.S. government approved. Other shared-key ciphers include Twofish, ChaCha, and others.

Public-key encryption (or *asymmetric encryption*) is a newer method that uses pairs of keys, not just one. Sometime last century it was discovered that pairs of keys can be generated with the remarkable property that data encrypted with one key can only be decrypted with the other! Yet, even knowing the method used and encrypting key, an attacker can't determine the matching key except by trial and error.

To use this method, each party generates their own key pair. One of the keys can be sent easily to the other party; it doesn't need to be kept secret. This key is known as the ***public key***; the matching key is the ***private key*** which must remain secret. For party A to send data securely to party B, party A can encrypt the data using B's public key. The resulting data is well-scrambled and useless to anyone who doesn't possess party B's private key. To send a reply, party B uses party A's public key. (Notice how four keys are used, but only the two public keys ever need to be shared.) Eavesdroppers can only intercept the public keys, not the private keys which are never sent.

This method is known as ***public key encryption***. PGP and its modern descendant, GPG (or GnuPG), are popular examples. (These and similar programs follow the OpenPGP standard.) **The most common public-key method today is probably RSA**, but it needs a key length of at least 3,072 to be secure, which makes it slow. (An RSA key of 1,024 bits provides similar security to 80-bit AES; 3,072 bit RSA is equivalent to 128-bit AES.) A modern algorithm to replace RSA is elliptic curve cryptography (ECC), specifically Curve25519. ECC uses much shorter keys compared with RSA while providing equivalent security.

Public key encryption suffers from one drawback: it is very slow compared with single-key encryption. Real-world methods of secure communications only use public key encryption to share a large random number to be used for single-key encryption. Once shared, that number is used with shared-key encryption for the

rest of the session. When the session is over, the shared key is forgotten. This key is often referred to as a ***session key***.

This is how the protocol for HTTPS works. Originally known as SSL, the fixed, newer versions were renamed "TLS". Until 2013, only the 1.1 or older versions of TLS was available, and most web browsers (and Java) defaulted to SSL or TLS 1.0, even though these methods were known to be broken. Modern browsers should only accept TLS version 1.3 or newer.

> TLS is a protocol that uses many different crypto algorithms. Both parties must negotiate the algorithms to use and their parameters. (This is true of most security protocols.) Your software must be configured for the algorithms and parameters you are willing to use, in preference order. These are called *cipher suites*. For example:
>
> TLS_AES_128_GCM_SHA256, Protocol: TLS 1.3, Certificate type: ECDSA (P-256), TLS curves allowed: X25519, prime256v1, secp384r1, HSTS: max-age=63072000 (two years), and Certificate lifespan: 90 days.
>
> You can find good (and current) advice on TLS configuration from wiki.mozilla.org/Security/Server_Side_TLS and from the German Federal Security agency.

Another protocol that uses public key encryption with session keys is **SSH** (*secure shell*). You can use SSH keys to authenticate yourself to GitHub (and transfer files securely). See the Git project (#2) for more information about how to generate such keys and use them.

> Keys are too long to be typed in by humans. Keys are stored in files. In fact, passwords, keys, and other secrets (sensitive data) are often stored in files, or *vaults*. Such files should be password protected and encrypted using shared-key methods. **If the vault has a poor password that is easily guessed, the strength of the other keys in the vault won't matter.** (Git uses the host's vault to hold HTTPS credentials; on Windows that's "Credential Manager".)
>
> Sometimes keys are stored off-line, say on a flash disk. Other times they are stored online in unencrypted files (which is convenient but insecure).

Digital Signatures

Public key methods can be used for more than just sharing a session key. A person receives a message from the U.S. president; how can you tell if it is really from the president and not an impostor?

The trick is to have the president encrypt the message with their own private key before sending out the message. When received, you download the president's public key, and use it to decrypt the data. If you get a real message, you know the

message must have been encrypted with the president's private key, and furthermore, was not modified in any way after it was encrypted.

Encrypting data with one's private key is known as ***digitally signing***. There are several ways this could be done without encrypting the data. A ***digital signature*** is some data that can be used to validate some data (typically, a file) came from a trusted source and was not modified, and that can be kept separate from the data. (To verify, you need both the data and its digital signature.) There is a U.S. standard for digital signatures, DSA, which supports several approved algorithms.

Hashing is a technique for scrambling data. Unlike the encryption methods discussed above, hashing is one-way. You cannot decrypt hashed data. Most hashing algorithms have a fixed-size (and fairly small) output regardless of the size of the input.

Modern hashing algorithms are very fast and can handle large amounts of data quickly. Some examples include Murmer3, CityHash, SpookyHash, and xxHash. None of these are safe to protect secrets! They are mainly used to implement hash tables (HashMap) to detect accidental corruption, and to find duplicates.

Cryptographic secure hashing is used to store secrets such as passwords ("password derivation functions") and when generating digital signatures. When a user needs to be authenticated, the password they enter is hashed the same way and the results are compared byte-for-byte. If the hashes match, the user must have entered the correct password. This way the password never needs to be stored on a server, just the hash of it. Examples include SHA-256, SHA-3, and Blake.

(See Wikipedia for a list of hash functions.)

> Modern hardware renders most older hashing methods insecure; attackers can quickly guess trillions (!) of passwords to see if the hashes match.
>
> Using per-use ***salt*** extends hashed passwords by adding additional random data (the salt). Even this is not secure anymore (2020). You need to use an *iterative* (or *key-derivation*) method to make guessing take too long to be effective.
>
> While not as secure as a digital signature, another use of hashing is to verify the data you have now is exactly the same as when the hash was generated. This provides a simple and fast way to check if downloaded files were corrupted or not. If the data uses a shared key with the data (instead of random salt), the result is a *message authentication code* (MAC or HMAC).

Always store sensitive date either encrypted or cryptographically hashed. Even if an evil-doer gets hold of the data, they cannot do much with it.

For hashing secrets such as passwords and keys, it isn't enough to use a good hashing algorithm; you must use it in a way the make it resistant to various attacks.

(These systems are called key derivation functions.) For managing passwords, today the best advice is to use an iterative, salted hashing function. Best advice (2024) is to use **Argon2id**, followed by scrypt or yescrypt, then bcrypt, and the previous "best advice" choice of PBKDF2-SHA512 as a last resort.

With all security algorithms and protocols, there are parameters you must set to use them correctly. If the parameters are poorly chosen, the security is compromised. Find good advice on the Internet for your chosen crypto's parameters. For example, for Argon2id you can look at crypto.stackexchange.com, https://tools.cisco.com/security/center/resources/next_generation_cryptography, and NIST.SP.800-57pt1r5.pdf (especially part I - General).

(Security recommendations change over time, so you need to check before choosing an algorithm/protocol or parameters, each time you include such security code in your software and periodically thereafter, as part of a code security audit.)

Java 24 has many enhancements for multiple types of crypto systems.

Important Rules for Crypto:

1. **Never invent your own security algorithms or protocols.** It takes a PhD in security and math (and deep knowledge of modern hardware) to do this correctly.
2. **Never implement security algorithms or protocols.** There are special considerations for security software that most developers are unaware of (e.g., constant-time comparisons, erasing secrets from memory), making such software extremely difficult to implement correctly.
3. **Never use ECB mode for block ciphers.**

What you should do is: find a popular and well-regarded implementation of the modern secure algorithms you want to use. For example, I generally use (as of 2023) Bouncy Castle. Don't forget to lookup the parameter settings to use as mentioned above.

TL;DR (the bottom line): In 2024, use TLS 1.3 or newer, AES-256, and either SHA-512 or (for secrets such as passwords) Argon2id. Regularly (e.g., annually) check NIST to see what algorithms are approved; they deprecate old ones if no longer sound, and add new ones from time to time. Also check regularly for current advice on setting crypto parameters correctly.

Remember that hashing is also used for non-security purposes, such as lookup tables (hash tables). Since cryptographically strong hashing is slow, these other tasks often use faster hashing algorithms. Just be careful not to use these accidentally when a secure hash is needed, a common error!

PKI: Trust and Certificates

Anyone can generate a pair of keys, put one of them on a website someplace, and send a message digitally signed, claiming to be Bill Gates or anyone else. How can you tell who really sent a message?

This problem isn't new with computers. One historical solution has been adapted to modern technology: the system of *public notaries*. If parties from different cities needed to conduct business via the mail, how could they tell if the person who signed the contract just received is who they claimed to be? People couldn't be expected to know everyone's signatures well enough to spot forgeries. But, each town has a small number of well-known people, or public notaries, whose signatures were seen all the time by many people, and therefore hard to forge. So all you did was have your local notary witness your signature, and sign also. The notary's signature says "this person who signed this document is indeed Bill Gates" (or whoever it was supposed to be).

Today the digital equivalent of public notaries is the well-known ***certificate authorities*** (or "CA"s for short). Each of these CA's generates a key pair. The CA's public key is widely distributed; Windows, Java, and web browsers maintain lists of the ones thought to be trust-worthy. (*Show in browser.*)

The way certificate authorities work is known as the ***public key infrastructure***, or "PKI". What happens is an organization or individual generates a key pair. The public key, along with the identity of the person or organization, their contact information, and other information is used to create a document in a standard format known as a ***certificate signing request***, or CSR.

The CSR is sent to a chosen CA, along with proofs of identity (and payment). After checking your identity, the CA will add some data (their identity, a certificate serial number, a validity period, and other data) to the CSR. The result is then digitally signed using the CA's private key, and that digital signature is added. The result is a ***certificate***.

To use a certificate issued by a CA, party A can digitally sign anything, then send the data, the digital signature, and the certificate to party B. The recipient (party B) can easily check the certificate is valid and unaltered, and therefore the public key it contains is in fact the one issued to the sender (party A). Now when you validate the digital signature, you have trust that not only is the data unaltered since it was signed, but that you know who signed it.

> Some CAs provide limited use, free certificates for testing or educational purposes, with limited or no identity proofing (low-security certificates). You can also sign your certificate yourself (usually with a second key pair) while testing and developing. Such certificates are called ***self-signed***. If

> you see some self-signed executable when surfing the Internet, **be very careful!!**

Today, malware infects software. To combat this, many vendors that allow third-party software require that software be digitally signed. Windows does this with all code (in "S" mode) and Java allows it.

While you can use self-signed certificates for testing and development, you should not use them in production. You can obtain free website certificates from Let's Encrypt and specialty certificates (say for code or document signing) from other vendors (generally not free.)

> Many TLS library APIs do not fully verify the server's TLS certificate automatically. For example, they may allow connections to a server when there is no server certificate, they may allow any certificate (instead of a certificate for the site you are trying to connect to), or they may allow expired certificates. These are common mistakes.
>
> **Always use a well-known library for all crypto!**

Publicly accessible sites can have their TLS certificate and settings thoroughly tested by the free www.ssllabs.com site.

Signed Jar Files

Show signed code demo in class resources.

Since Applets and WebStart were deprecated, it is rare to need to sign Java code. (In Java 25, the Java Security Manager part of the JRE will be gone!)

Still it is useful to know the two commands used and the general method of digitally signing data. With Java, you use the command:

```
keytool -genkeypair -alias mykey
```

To generate a key-pair with the name (the *alias*) of "`mykey`". The keypair will be stored in the default *keystore*, the file `.keystore` in your home directory. I prefer to use additional command line arguments to specify the name and type ("PKCS#12", with the extension of "`.p12`") of the keystore to use.

You can also add the argument "**`-validity` *`days`***" to specify for how long your key is valid. While learning, use short durations (for our class, 3-4 months is fine). For production use, 3-5 years was common; today, shorter periods are preferred.

Next, use `keytool` to generate a ***certificate signing request*** **(CSR)** document. This is a standard text document that contains information about you, your organization, and your public key (but not the private key!). You can generate a CSR from a key-pair like this:

```
keytool -certreq -alias mykey -file mykey.csr
```

To generate the certificate you need, you submit the CSR, along with various proofs of identity (and your payment) to your CA of choice. Then wait until they have generated the certificate; usually that takes days or weeks since they must verify your identity. Note this command is interactive and will prompt you for the information you need to enter.

> To use self-signed certificates, you don't need to generate CSRs or use any CA. Keytool can do this.

When done, you download your signed certificate from the CA and import it into the same keystore that holds the matching private key:

```
keytool -importcert -alias mykey -file cert_file
```

Make a backup of the keystore and keep it safe! Also, do not lose the password! If you lose access to your private key, you will need to purchase a new certificate.

Once you have done these steps, you can sign Jar files as often as needed. The command is to sign `MyApp-unsigned.jar` is (all one line):

```
jarsigner -signedjar MyApp.jar -tsa URL
   MyApp-unsigned.jar mykey
```

The "TSA" is a *time stamping authority*. This adds a digitally-signed time stamp to your certificate. That is useful if the certificate is ever revoked; with the time stamp from a trusted authority, systems can tell if your code was signed before or after the certificate was revoked. Many TSA's will do this for free, you can find a list of free TSA URLs at gist.github.com/Manouchehri/... and elsewhere. For example, http://sha256timestamp.ws.symantec.com/sha256/timestamp.

See the Java tools documentation for all the options with these two commands.

> In 2021, the Linux Foundation, Red Hat, Google, and others have come together to make code signing simple and free, the way Let's Encrypt has done for web site certificates. The new service is at sigstore.dev, although the services isn't yet available (3/2021).

Lecture 5 — Additional (Advanced) Java Features

Configuration Settings, System Properties, and Environment Variables

In many programming languages, software can be configured in many ways without needed to recompile the code. Configuration settings can come from a file (often read at program startup), from the command line, from environment variables, or from a configuration server. Types of such data include:

- usernames,
- network port number and IP addresses,
- file names,
- application modes,
- credentials for external services (such as database usernames and passwords),
- logging levels and log destinations,
- license and other keys,
- other parameters (such as tax rates, formatting details, etc.)

> Configuration servers fall into two groups: configuration management of servers, such as Puppet, Chef, Ansible, and Salt; and application configuration data servers including Zookeeper, Consul, and Etcd. Application configuration servers are designed for enterprise applications run in a data center, and often include other services and features.

For just application configuration data, I like etcd. Using such a server means your distributed application can have configuration updated dynamically, without a restart. This is very important for enterprise applications, which cannot afford downtime to change a configuration setting. Additionally, many modern applications run as a large group of virtual machines or (more commonly today) containers. Without a central server the various VMs/containers don't run a consistent configuration. However, the applications must be written to use such a server. (Using dynamic configuration is one feature of *cloud-ready* or *cloud-native* applications.)

Often multiple ways are used: defaults are compiled in, a file is read if present to override the defaults, and command line arguments to override that. In addition, ***environment variables*** (named values) are often used. (Consider Git: it uses compiled-in defaults, three configuration files (system, per-user, and per repo), and command line arguments.)

Until Java 5, the JRE had no way to read and pass the environment variables from the underlying OS. Instead the JRE has a similar concept built in, called ***system properties***. These are named values your program can access. Many are set automatically (such as `line.separator`), and some (such as a `user.name`)

are hidden if a security manager is used (as with applets). (*Show list from System.getProperties*.) Note some properties are security sensitive and unavailable to Applets/WebStart unless the code is signed. Normal desktop apps have no security restrictions by default.

You can define your own properties, or override the system ones. On the command line you can pass multiple "**`-D`*`name=value`***" options (no spaces in there). You've seen this before, with Maven.

It is also possible to use a Java properties file (discussed later) to load and save many properties. To get the value of a system property, use `System.getProperty(`*`name`*`)` (or safer, `getProperty(`*`name, defaultValue`*`)`).

Environment variables are a traditional way to use configuration data; for example, Java uses `CLASSPATH`. (*Review environment variables, show set.*) Java 5 added support to read the environment with `java.lang.System.getEnv()`. *Show `ShowProps.java` demo (includes environment variables too)*.

Boxing and Unboxing — *Since Java 5*

Java 5 added auto-wrapping: ***autoboxing/unboxing***. This just means that primitive values will be automatically converted (typecast) into the associated object types (the immutable wrapper classes in `java.lang`) and vice-versa, as needed. For example, without autoboxing:

```
Integer num = Integer.valueOf(5);
int sum = 2 + num.getInt();
```

But with autoboxing, this can be written as:

```
Integer num = 5;
int sum = 2 + num;
```

One special case to worry about is `null`. There is no `int` equivalent, so this:

```
Integer num1 = 0, num2 = null;
int i1 = num1;      // no problem.
int i2 = num2;      // throws NPE!
```

can mysteriously throw a `NullPointerException` (NPE). This is especially mysterious when using collections such as a `Map`, as the NPE is thrown from deep within the standard library. **If a null value is possible, don't use auto-unboxing without a try-catch block for NPE.** (**Show boxUnbox.java**.)

In Java 8, this issue was addressed using new Java features and a new class, `Optional` (discussed later).

The current Oracle JRE implementation of `Integer.valueOf` (used when converting an `int` to an `Integer`) pre-creates `Integer` instances for integers in [-128, +127] to make autoboxing more efficient. (The range is not specified in the Java docs for this method, but you can look at the source code.) Consider this code:

```
Integer a127, b127, a128, b128;

a127 = 127; b127 = 127; a128 = 128; b128 = 128;
if (a127 == b127) System.out.println("127 works");
if (a128 == b128) System.out.println("128 works");
```

This will print "`127 works`" but **not** "`128 works`"! The autoboxing essentially has the compiler rewrite this to:

```
a127 = Integer.valueOf(127);
b127 = Integer.valueOf(127);
a128 = Integer.valueOf(128);
b128 = Integer.valueOf(128);
```

Due to the implementation quirks, this becomes similar to this:

```
a127 = Integer.private_preallocated_Integer_127;
b127 = Integer.private_preallocated_Integer_127;
a128 = Integer.valueOf(128);
b128 = Integer.valueOf(128);
```

The solution to this is to remember to use `.equals` for objects, not "==", when when using autoboxing.

Interfaces

In Java I, we covered the basics of declaring and using *interfaces*. Some details were omitted. In addition, in recent Java versions additional features were added to interfaces.

An interface is a contract, a guarantee that objects of any class that implements an interface have some behaviors (public members). As such, you can form collections of objects of mixed classes, as long as they implement the same interface.

The specification of a class's public members is referred to as its API.

Here's a simple example as a refresher:

```
public interface Transport {
   double getWeightLimit ();  // in pounds
   int getMaxNumOfPassengers ();
}
```

The abstract method defined in an interface is implicitly `abstract` and `public`. It is allowed but not recommended to use these modifiers.

Not all interfaces define any required API. Such interfaces are known as ***marker*** (or *tagging* or *annotation*) interfaces. These are used when it is enough to annotate a class with metadata. (These predate Java annotations.) An example of a marker interface is `Serializible`. (*Show.*) Classes marked with this interface are compiled in a way that allows their objects to be serialized (saved as a stream of bytes) and reconstituted later or on a different JRE (i.e., a different computer).

Like classes, compiled interfaces result in `.class` files. Also like classes, interfaces can inherit from other interfaces. **Unlike classes, an interface can have multiple parents.** For example:

```
public interface Readable {
   String read ();
}
public interface Writable {
   void write ( String data );
}
public interface Editable
              extends Readable, Writable {
}
```

Qu: What if two parent interfaces declare the same method? Ans: Not a problem, only one implementation of that method is needed.

You can override an inherited abstract method, but doing so is rarely useful. (You can refine the return type, or restrict which exceptions are thrown.)

Modifiers such as ~~`strictfp`~~, `final`, or `static` are not legal for abstract methods declarations in interfaces. (The methods of a class that implement an interface's abstract methods can use these modifiers.)

In addition to abstract methods, interfaces can contain constants. **All constants are implicitly declared as `public`, `static`, and `final`.** (Interfaces with `int` constants were commonly used before `enum`s were added to Java; `enum`s are discussed later.) It is not a problem if two or more parents inherit the same constant. Only one copy of that constant is created in classes that implement the interface. (It would be an error if the constants had different values however.)

Interfaces can contain nested interfaces and classes. Since they are implicitly `public` and `static`, their presence doesn't affect objects of classes that implement the interface. Here's what this looks like:

```
interface Demo {
  void aMethod ();
```

```
  int NUM = 42;
  interface AnInterface {};
  class NestedClass {};
}

class Foo implements Demo {
  public void aMethod () {
   AnInterface b = new AnInterface(){};
   NestedClass c = new NestedClass(){};
  }
}

class Bar implements Demo.AnInterface {
}
```

Qu: Why would anyone nest types (interfaces or classes) in an interface? Ans: This declares an interface named `Outer.Inner`. There is no further association with the enclosing class, except that code which cannot access `Outer` cannot access `Outer.Inner`. This is an acceptable style if you expect `Inner` to be used only from the outer class (or more specifically, from the methods of a class that implements `Outer`), so that you do not create a new top-level name. An example is `Map.Entry`.

As of Java 8, two additional interface members are allowed. You can define default methods and static methods.

`Static` methods in an interface need not be implemented by classes that implement the interface. (This is because objects don't contain the `static` methods in any case.) A `static` method in an interface is `public`. This ability allows the programmer to put a method that is related to the interface in that interface. Prior to Java 8, such methods were put into some other "utility" class. For example, see the `static` methods in class `java.util.Collections`. Such extra classes are no longer needed. (But the current utility classes will remain, since cleaning up the library might break older programs.)

A *default method* is something new to Java. Only interfaces can define these. Default methods contain a complete method body. The signature is preceded with the `default` keyword (it's a modifier). Any class that implements the interface doesn't need to implement default methods, although they can override them if desired. The syntax looks like this:

```
public interface Foo {
   default void someMethod () {
      System.out.println( "Default" );
   }
}
```

```
public class Bar implements Foo {
   public void aMethod () {
      someMethod();
   }
}
```

The point of default methods is to allow one to enhance an existing interface with new methods. Just adding a new abstract method would cause all existing programs to fail to recompile, as they would need to implement the new method too. (The compiled classes will still work, as long as nobody invokes the new method on their objects.) In Java 8, many existing interfaces were updated, such as Collection (added a sort method).

If a child interface extends another interface which contains a default method, it can redeclare the method without the `default` keyword. The method becomes an abstract method, and any class that implements the child interface will need to provide the method. The child interface can also redefine the default method (provide a new method body). Also, the child interface can reuse the parent's default methods even if they have the same name, by qualifying `super` with the correct parent:

```
public class Bar implements Foo {
   public void someMethod () {
      Foo.super.someMethod();
   }
}
```

This is needed because with multiple parents, there might be multiple default methods with the same signatures.

In some respects, Java interfaces provide multiple inheritance since a class can inherit default methods from two or more interfaces that it implements. Java avoids the issues that multiple inheritance causes by using three rules:

- A method in a class or inherited from a parent class always takes precedence over default methods. If no such method is there, a default method (of the correct signature) will be used.
- If some class inherits default methods from two interfaces `Foo` and `Bar`, and interface `Bar` extends `Foo` and overrides the default method, then the default method from `Bar` is used. We say that the default method of the sub-interface (the child interface) is more specific than the one of the same signature, and the more specific one is always used.
- If some class inherits default methods from two interfaces `Foo` and `Bar` that do not extend each other, then you will get a compiler error saying

> the reference to the method is ambiguous. You can fix that by implementing the method in the class, which is the method that the compile will pick due to the first rule. In that method's body, you can refer to either or both of the inherited default methods using "`Foo.super`" or "`Bar.super`".

A ***functional interface*** is an interface that contains a single abstract method. (They can also contain `static` methods, default methods, or constants, but only one abstract method.) The annotation **`@FunctionalInterface`** is like `@Override`. It tells the compiler that the interface is intended to be a functional interface. The compiler will then complain if the interface has multiple abstract methods (or none). Functional interfaces are used with another Java 8 feature, Lambdas.

Best practice for designing interfaces is to keep them focused on one small abstraction.

Initialization Blocks

Constructors are not the only way an object's properties can be initialized. The properties (***fields***) can be initialized at the point of their declaration. Furthermore, one constructor can invoke another (via **`this`**) or a parent class' constructor (via **`super`**). (Discussed later.) However, if there is complex initialization code that should be run in every constructor, duplicating the code or keeping all chained constructors in sync is a burden. (It is sometimes possible to put the code in a private ***no-arg*** constructor and invoke that from the others, but not all classes should have such a constructor.) Constructors are not invoked to initialize `static` properties of a class either. Finally, *anonymous inner classes* cannot have constructors at all!

The obvious way to solve these problems is to initialize fields directly rather than from constructors. But this isn't always the best way: sometimes several fields have related values, and it can be difficult to write expressions that would work. Sometimes the initialization depends on a complex set of statements (e.g., user authentication, database lookup, or file reading and parsing) that can't be combined into a single expression. The initialization expression can't use a method that throws a checked exception, as there is no place to put a try block.

One way to solve this issue is to initialize each field directly, using a method call. The method can be made private and as complex as required. **However, invoking methods on an object *before* it is fully constructed is dangerous and can result in *race conditions* where some fields are used before they are fully initialized.** (If the method in class X uses class Y, which uses class X, you have a cycle that the compiler can't always detect. This can happen more often than you'd think.)

To make these cases easier to maintain, rather than force you to write complex constructors and expressions, Java supports ***initialization blocks***. These are block statements (a.k.a. compound statements) placed outside of any method. Use initialization blocks for more complex initialization than can be written on one line, when no constructor arguments are needed, and there is some reason not to provide a no-arg constructor. Here's a trivial example:

```
class Foo
{  private static int nextID = 0;
   public final int idNum;  public String name;
   { idNum = nextID++; }
   Foo () {  name = "<unknown>"  }
   Foo (String name) {  this.name = name;  }
}
```

Here, `idNum` will be set regardless of which constructor is used. It is possible to have many such initialization blocks; they are executed in the order they appear before any constructor is run.

Initialization blocks can throw a checked exception only if all constructors of the class are declared to throw that exception. (*Internally, it's as if the code in the blocks were copied into each constructor. Sort of.*)

(The most common use for these (if any use is common) is to initialize anonymous class objects.)

A ***static initialization block*** is similar to a non-static one, except it is declared `static`, can only refer to static members of the class, and can't throw any checked exceptions. These are run once *when the class is first loaded into RAM* by the `classloader`.

It was possible (but not recommended) to run Java code with no main method, with Java 6 and earlier:

```
class NoMain {
  static {
    System.out.println( "Looky, no main method!" );
    System.exit(0);  // stop "main not found" error
  }
}
```

This compiled. When run, the class is loaded, causing the static initialization block to run. This won't work since Java 7.

(Qu: What happens when all constructors are declared `private`? Ans: only `static` methods and `static` initialization code of that class can create objects.)

Static initialization blocks are frequently used to initialize values that are expensive to lookup each time (say from a database), or when some statement must be done exactly once. For example, the JDBC API uses static initialization blocks to load the database driver code into the JVM. You can run `Class.forName(`*`name-of-driver`*`)`. (This isn't needed anymore, but still works.)

In a sense, the compliment of static initialization blocks are ***shutdown hooks***. When the current value held in `static` properties was loaded from disk/database, it must be saved to disk/database when the JVM shuts down.

Refresher on Constructors:

The first executable line in any constructor may be a call to another constructor of that class, something like:

```
Foo () {
   this(17);
   ...
}
```

Here the no-arg constructor invokes the `Foo(int)` constructor. When that is constructor finishes, the rest of this constructor's body is executed. Recursion (having one constructor invoke itself, or have constructor A invoke constructor B and B invoke A) is illegal.

Ultimately a constructor is run that doesn't start with a call to `this`. Before running the body of that constructor, a constructor of the parent class is run to initialize all the inherited fields. Normally the parent's no-arg constructor is invoked, but you can specify a different one if the first line in your constructor is "`super(`*`args`*`);`".

Java since version 22 does allow statements before `this` or `super` in a constructor! (See JEP 447.) The point is to allow a constructor to validate (or possibly transform) arguments passed to `super` or `this`. You still cannot access fields before they are initialized though.

(**Show InitBlock.java.**) These are the 12 steps to initialize an object `f` from the code `Foo f=new Foo();`

1 The Classloader locates `Foo.class` file on disk, and any dependent classes (e.g., superclasses).
2 Byte code is verified.
3 Class objects for dependent classes are found on disk, created, and initialized (each using these same steps 1-6).

4 Sufficient memory is allocated for all Class object's fields, including all static fields in the class Foo.

5 All static fields are initialized to default (zero, false, null) values.

6 Static fields and static initialization blocks are executed in the order they appear.

The remaining steps are per-object:

7 Sufficient memory is allocated for all instance fields, included inherited ones.

8 All instance properties are initialized to default (zero, false, null) values.

9 The parent class' instance fields are initialized (including any initialization blocks).

10 The parent class' constructor is run to finish initializing inherited fields.

11 Instance variable initialization expressions and initialization blocks are executed in the order they appear.

12 The body of the matched constructor is executed.

Steps 1–6 occur only once in the lifetime of the JVM. The rest occur each time a new object is created.

Clone Method and Copy Constructors

The `clone` method is designed to duplicate an object. This is only needed for mutable objects. (Qu: Why don't immutable (read-only) objects need `clone`?) This method comes from class `Object` and you need to override it. However, the method will throw `CloneNotSupportedException` unless your class implements the **`Cloneable`** interface.

In your method, you must use `super.clone()` to get a new object, which will be a *shallow* copy of the current one. Next you need to change any fields in the copy that refer to mutable objects, with copies of those objects. Finally, you can return the clone. (***Show `CloneDemo.java`***.)

It is sometimes better to implement a copy constructor instead of (or in addition to) a `clone` method. A ***copy constructor*** is a constructor that takes an object of that same class as an argument:

```
public class Foo {
  public Foo ( Foo f ) {...}
  ...
}
```

Then to make a copy, use code such as:

```
Foo f1 = new Foo();
Foo copy = new Foo( f1 );
```

Note that `Object.clone` returns an `Object`. Prior to Java 5, your `clone` method also had to return `Object`. But with covariant return types, you can return the correct type.

> Even for mutable classes, `clone` isn't generally a good idea. Java just doesn't handle clone or deep copies well. You can't properly implement `clone` if your class defines any mutable fields as `final` or if it uses generics. Serialization and unique object IDs can also be a problem. There is a well-regarded library that provides deep copies via serialization: Apache Commons Lang3 using the `SerializationUtils.clone()` method, but this is somewhat of a hack. (See also `DeepCopy.java` in the resources, which works similarly.)
>
> Consider alternatives: using immutable objects only, implement a copy constructor, or implement a static factory `copy` method in all classes.

Advanced Object Construction Techniques

First Design Patterns

(Design and design patterns are covered much later in this course, but you should know something about them up front.)

In programming, as in any endeavor, the obvious way to accomplish some goal isn't always the best way. An incorrect solution may appear fine at first, but over time you will realize the problems that approach has caused. Fixing the problems later is usually very expensive.

People who have encountered the same task in the past may have found better solutions than the obvious ones. If they share them publicly, others can use them. Solutions to common programming tasks can be called *design patterns*. If you are familiar with some, then when programming you will (hopefully) recognize when those patterns are useful. There are many catalogs of patterns on the Internet (see our class resources for some) and some excellent books are available. (The classic *gang of four* book on this topic is difficult to read. Consider Heads-First books for more accessible material.)

> An ***anti-pattern*** is a known-bad choice. Knowing some of these will allow you to avoid making those same errors.

Complex Object Construction

Suppose you need to initialize several related fields while allowing some optional constructor arguments. The straight-forward way would be to have a constructor take required arguments, then have users change the options via `set` methods. This is easy to code but hard to use: it also allows inconsistent state and puts the construction burden on the users of the class, with no checking that the user has set

all related properties correctly and consistently. This also makes it impossible to define an immutable class (which cannot contain any `set` methods). How can you make your API safe and easy to use?

An alternative is to use the ***telescoping constructor pattern***. In this pattern, you create one main constructor that takes all required and optional values as arguments. Then you create other "convenience" constructors that take only some (or none) of the optional values (while taking all the required ones). These constructors invoke the main constructor (or one of the other convenience constructors, a technique sometimes called *constructor chaining*) and supply the default values for the missing arguments. (*Demo.*) You can use private static final values for the defaults, so you only need to define them in one place and can use them from all the convenience constructors.

With more than a few constructor arguments, this pattern becomes unwieldy. For one thing, all constructor signatures must be unique, so you cannot have two constructors taking (for example) two `int`s or two `String`s.) Even if you could, for a class with 10 optional arguments, you would need to implement over 1,000 convenience constructors for each possibility!

When there are many optional constructor arguments, an elegant constructor pattern is the ***builder* class** pattern, combined with the ***method chaining*** pattern. A *builder* is a "helper" class (often nested). It has many methods to set the properties, some required and some optional. Once all the fields are set, the builder object is used to construct the "real" object. The "real" class has a single constructor that takes a builder object. A niftier way is to nest the builder class in the "real" class, make the "real" class constructor private, and have a static `build()` method in the builder that invokes the private constructor and returns the resulting "real" object. Often the final method is named `create`; I've even seen "doIt" once! (*Show java.util.Calendar.Builder*.)

> A related pattern is to use a helper class that contains all optional parameters and set/get methods for each. In effect, you are bundling the options into one object. `java.awt.GridBagLayout` uses this approach, with the `GridBagConstraints` helper class. (*Show*.)

Chaining method calls is a commonly used technique, not just with the builder constructor pattern. It is also known as having a *fluent API*. In general, method chaining looks like this:

```
result = obj.method1().method2().method3();
```

All methods except the last return `this` (or at least a reference to an object of the same class). None of the ***intermediary methods*** can return `null` and all of them should return a reference to the same object (or at the least, a reference to an object of the same class). The final method in the chain can return anything; it is called a

terminal method. When used with the builder (constructor) pattern, it is common to name the intermediary methods the after the fields they initialize.

When used with the builder pattern, method chaining provides an elegant approach to creating objects. You create a builder class method for each optional argument with the same name as the field holding that value. Each of these methods initializes that field from the argument, then returns `this`, allowing the methods to be chained. The terminal method then constructs the object using the builder. (Show **`StudentBuilder.java`**.) Note that chaining method calls as shown in the example won't work if any of the methods might return `null`!

Another commonly used constructor technique is not to have any public constructors at all. Instead, provide a static method that builds and returns an object. This is called a ***static factory pattern***. Having a named method has several advantages over a constructor. For one thing, you can't have two constructors with the same signature, but you can have two methods with different (and more descriptive) names. Unlike using `new` and a constructor, factory methods don't need to return a new object every time. It can return a reference to some existing object if that would make sense (as it does for immutable classes). Factory methods don't have to return an object of the exact same class; they can return objects of some subclass (a child object, e.g., having class `Dog` return a `Schnauzer` or some other type of object, depending on the arguments passed in). (*Show java.text.NumberFormatter*.)

The code would look similar to this:

```
class Foo
{ ...  // private fields
  private Foo () { ... }
  public static Foo getInstance () {
    Foo f = new Foo();
    ... // initialize f
    return f;
  }
}
```

The disadvantage to using factory methods with private constructors is that without accessible constructors, you cannot extend the class. (Constructors could be declared `protected` in this case, to allow that.)

(There are other object creation patterns as well; we will discuss one later.)

> **Warning!** No constructor should invoke arbitrary methods (of that class). Methods may not work until an object is completely constructed. It is safe to invoke private or final methods, if you're very careful! A violation is `java.util.Random`, whose constructor invokes `setSeed()`.

Static import — *Since Java 5*

```
import static java.lang.Math.*;  double x = sin( PI );
```

Don't overuse: use only to improve readability, when the class is obvious.

Enums — *Since Java 5*

In many situations, the *type* of some variable can be defined with a small set of possible values. For example:

- Survey scores: stronglyAgree, agree, neutral, disagree, and stronglyDisagree
- Playing card suits: clubs, diamonds, hearts, spades
- Choice of pizza toppings
- New car colors
- Error or status codes
- Day or month names

and others. Traditionally you make variables that need to hold such information of type `int`, and define some integer constants with useful names:

```
interface SeasonConstants {
   int SPRING = 1;
   int SUMMER = 2;
   int FALL = 3;  // or AUTUMN
   int WINTER = 4;
}
```

And use them like this:

```
   int currentSeason = SeasonConstants.FALL;
```

This is a bad idea because there is no automatic way to make sure `currentSeason` only is assigned the four legal values, constants don't report correctly in error messages, they don't print right, and have other nasty traits as well. (It's also a bad idea since `static final int` constants are replaced by their literal value in the compiler. If you change one, you must recompile all code everywhere that references that constant! You won't get any error or warning either; you just have to know.)

For many years, Java experts had tried to convince programmers that the correct way was to create a new class `Season`, with four public constant objects and no public constructor. This has so many problems and was so complex it never really caught on.

Java now provides an easy way to create a new type by enumerating (i.e., listing) the possible values:

```
enum Season { WINTER, SPRING, SUMMER, FALL }
```

which creates a new class Season whose variables can only refer to these values:

```
Season now = Season.FALL;  System.out.println( now );
```

You should use enum types any time you need to represent a fixed set of constants. Enums are `final` classes and the values are objects.

You cannot use inheritance to extend enums. (All enums implicitly extend `java.lang.Enum`. Since Java does not support multiple inheritance, an enum cannot extend anything else.)

You can add enum objects to collections, and use the `toString` method to display sensible output. Enums are comparable (thus a list can be easily sorted) and serializable (and can thus be used with Jakarta EE/RMI).

Enums include several useful methods: **`values()` returns a list of all values**:

```
for (Season s : Season.values() )  ...
```

```
enum Day { MON, TUE, WED, THU, FRI, SAT, SUN };
import static Day.*;
for ( Day weekDay : EnumSet.range(MON, FRI) ) ...
Day d = WED;
```

In general, an enum looks like this (values are created using the default no-arg constructor):

```
enum name {
   val1, val2, ..., valN;
   additional properties and methods go here
}
```

You can also provide constructors to initialize each value. Here's an example:

```
public enum USCoins{ PENNY(1), NICKEL(5), DIME(10),
QUARTER(25), HALF_DOLLAR(50); // Add other coins too
   private final int value;
   USCoins ( int value )  { this.value = value; }
}
```

The constructor for an enum type must be package-private or private access. It will be used to construct the constants that are defined at the beginning of the enum body. You cannot invoke an enum's constructor yourself.

See **`java.lang.Enum`**, `java.lang.annotation.ElementType`, and `java.util.enumSet`.

Enums have a number of automatically generated methods (in addition to methods inherited from `java.lang.Enum`; which methods are generated and

which come from the class depends on the version of Java used). One important generated method is:

```
static type[] values()
```

This method returns an array of all values, in declared order.

Since these methods are generated automatically by the compiler, you won't find them documented in the Java docs!

In addition to the normal methods you'd expect to be inherited (such as `toString` and `equals`) from class `Object`, two methods are inherited from class `Enum`: `int` **`ordinal`**`()` (returns the ordinal position of the value, starting at zero, of the declared order of values), **`name`**`()` (returns the `String` name of the constant, same as **`toString()`**), and *`type`* **`valueOf`**`(String)` (returns the appropriate enum value; throws `IllegalArgumentException` if there is no constant of that name). (**Show CoinPurse.java**.)

Enums have many nice features. They are immutable, singleton, discoverable, ordered, comparable, equatable, hashable, and serializable and deserializable from `String`. Also, enums are very efficient using `EnumMap` and `EnumSet`, fast with `HashMap` and `HashSet`, and can be use in `switch` statements. That's pretty good!

> Storing enums in a database is problematic since the ordinal values may change over time as values are removed/added. If planning on DB storage you should consider a "permanent" ordinal value for each enum that never gets reused. That is, never delete one (okay to mark it as "deprecated"), and only add new ones at the end.

An `EnumSet` is implemented as a "bit map" and should be preferred to using such bit fields. If the Set grows beyond 64 elements, then the `EnumSet` switches implementation to a more traditional (but much less efficient) one.

Switch Statement Enhancements

This is as good a point as any to discuss the new features of the `switch` statement. Many features have been added over the years and `switch` is now easier to use and more powerful.

The switch statement, enhanced in Java 5, allows a convenient use of enums. Note that the `case` values don't have to be qualified with the `enum` class' name; it is determined from the switch's control value automatically:

```
enum Shape { RECTANGLE, CIRCLE, LINE };
Shape drawing = ...;
switch (drawing) {
  case RECTANGLE: g.drawRect(x,y,width,height); break;
```

```
  case CIRCLE: g.drawOval(x,y,width,height); break;
  case LINE:   g.drawLine(x,y,x+width,y+height); break;
}
```

Java 14 includes major enhancements to the `switch` statement. A new form of switch eliminates the need for the `break` statement. The above switch becomes:

```
switch (drawing) {
  case RECTANGLE -> g.drawRect(x,y,width,height);
  case CIRCLE    -> g.drawOval(x,y,width,height);
  case LINE      -> g.drawLine(x,y,x+width,y+height);
}
```

(Note the "arrow" instead of the colon.) With this new form, you can also use a comma-separated list of case labels:

```
switch (foo) {
  case X:
  case Y: doSomething(); break;
  case Z: doSomethingElse(); break;
}
```

Becomes:

```
switch (foo) {
  case X, Y -> doSomething();
  case Z    -> doSomethingElse();
}
```

In the common case of each case assigning to the same variable you can do this, a ***switch expression***:

```
int num = switch (foo) {
  case X,Y -> 5;
  case Z   -> (int) Math.pow(2, 3);
  default  -> { int t = ...; break func(t); }
}
```

Notice the `break` statement in a switch expression takes an argument, as an expression must evaluate to a value.

Java 21 enhances switch with *patterns* of the sort allowed with `instanceof`. For example:

```
static void testTriangle(Shape s) {
    switch (s) {
      case Triangle t when (t.calculateArea() > 100) ->
        System.out.println("Large triangle");
      case Triangle t ->
```

```
                System.out.println("Small triangle");
            default ->
                System.out.println("Non-triangle");
        }
    }
```

As you can see, you can switch on the type of the argument to `switch`, and add Boolean expressions (called *guards*) with the `when` keyword too. Using the "`->`" syntax means no fall-through, so no `break` is needed. For more details, see JEP 441.

> You can use primitive types too. Note that if you have exhaustive case labels, you cannot also have a default. For example:
>
> ```
> int num = ...;
> switch (num) {
>     case int i when i > 0 -> ...
>     case int i -> ...
>     default -> ... // No good! Previous case covers all possible values
> ```

Annotations — *Since Java 5*

Annotations (also called *metadata*) allow you to add information to your code that the compiler or some other tool can use. **An annotation can be used anywhere a Java modifier (e.g., *public*) can be used; that is, on any declarations.** (Style: annotations first.) Java 8 allows annotations on any use of a "type" (e.g., `new Foo()` expressions).

> Think of an annotation as a Post-It® note you put on some piece of code.

For example, the annotation **`@Deprecated`** can be used as a method modifier to inform the compiler the method, although legal, is discouraged in favor of a new (unspecified) method. With this annotation, a warning gets generated by the compile if other code tries to use the deprecated fields or methods.

> In pre-Java 5, a comment above a method could contain the word "`@deprecated`", (lowercase "d") and the compiler would issue warnings if other classes' code used that method. This is a rare example of a compiler altering its behavior due to a comment. (Most compilers completely ignore comments.)

The annotation **`@Override`** can be used as a method modifier when overriding inherited methods. This annotation tells the compiler you intend the method to override something and not overload or define a new method. The compiler generates an error if you make a mistake like this:

```
class Foo {
  @Override
  public boolean equals( Foo f ) { ... }
}
```

should produce an error, but wouldn't without the annotation. (Qu: Do you see the error? Ans: The argument must be of type `Object`, not of type `Foo`.)

JUnit shows a good use of annotations. All you do to define a test method is precede the method with the `@Test` annotation. That's it! When you later run the JUnit framework (giving it the class containing your tests), it finds the test methods by looking for methods with this annotation. TestNG uses this too.

Other types of annotations can be defined (by anyone) and used to automatically generate "boilerplate" code. For example, `@Property` might be used to mark some properties of a class as JavaBean properties, and could automatically generate the `get` and `set` methods required. Or an `@RMI` annotation could be defined that would cause stub and skeleton classes to be generated automatically.

In Jakarta EE, EJBs use annotations to automatically manage objects, define SQL database tables, to initialize instance properties (this is called *interjection*), and other things.

> There is a very cool library Lombok that uses annotations to generate entire classes from just the final fields you write! Lombok will be discussed further, later in the course.

You can write your own *annotation processors*, found in CLASSPATH automatically by the compiler. However, creating such processors yourself is very difficult; it is a dark art.

Standard Java SE Annotations

Java defines these annotations as standard (see `java.lang` (scroll to bottom) and `java.lang.annotation` package): `@Deprecated`, `@Documented`, `@Inherited`, `@Override`, `@Retention`, `@SuppressWarnings`, and `@Target`. Several more predefined annotations were added in Java 7 and Java 8.

> A programmer should know and use **`@Deprecated`, `@Override`**, **`@SuppressWarnings`**, **`@SafeVarargs`**, and **`@FunctionalInterface`**; the others are only useful when defining your own annotations.

We already discussed `@Deprecated` and `@Override`. The annotation `@SuppressWarnings` does what it says; however, the list of warnings is not standardized. But every compiler warning belongs to one of two standard *categories*, **`deprecation`** and **`unchecked`**. The `unchecked` warning can

occur when interfacing with legacy code written before the advent of generics. To suppress multiple categories of warnings, use the following syntax:

```
@SuppressWarnings({"unchecked", "deprecation"})
```

> With `@SuppressWarnings`, the list of warning that can be suppressed varies depending on the compiler used. There is a list for Eclipse in the on-line docs (search for *suppresswarnings*). NetBeans uses (apparently) the JDK ones. You can see a list of those by "`javac --help-lint`. All implementations support "all".

The Java 7 `@SafeVarargs` annotation, when applied to a method or constructor, asserts that the code does not perform potentially unsafe operations on its varargs parameter. (Varargs includes methods like `printf` which take a variable number of arguments; this is discussed below.) When this annotation type is used, unchecked warnings relating to varargs usage are suppressed.

The Java 8 `@FunctionalInterface` annotation indicates that the type declaration is intended to be a functional interface (which can use *lambda* expressions).

Annotations other than these standard ones won't do anything on their own; you'll need to pre-process your source code with ~~`apt`~~, `javac`, `javadoc`, or some other tool that recognizes those annotations. Before annotations, you marked methods deprecated using a special Java doc method (the compiler peeks at these): `/** @deprecated ...`*`reason`*`...*/`.

(Note this only works in doc comments, and uses a lowercase "d".) You should document why the method is deprecated and what should be used instead, in the doc comments.

> Note! It is not an error to use a deprecated method or variable from within that same class!

Using Annotations

Annotations can optionally be passed values called ***elements***. These values are then available to the extra tools that pay attention to them. For example, to say what warning messages to suppress, use: `@SuppressWarnings(value="unchecked")`. Here *value* is an element of the annotation, which may define as many of these as desired. Each element can be of a simple type or an array.

As a convenience, when there is only one element defined and it has the name `value`, you can omit the name. So the above can be simplified to: `@SuppressWarnings("unchecked")`. Also, if there are no elements at all, you can omit the parenthesis too.

Using annotations is simple; it is defining custom ones that gets tricky. To use existing annotations, just place one or more directly before the code you are annotating. If there are no *elements* (arguments to annotations are called "elements") passed to the annotation, you don't need any parenthesis. So both of these work the same way:

```
@MarkerAnnotation class Foo { ... }
@MarkerAnnotation() class Foo { ... }
```

(Annotations that take no element arguments are called *marker annotations*, much like marker interfaces.)

If an annotation takes exactly one value (for an element named `value`), you can just pass the value (of the correct type). If there is more than one element being passed, or you didn't name the one element "`value`", you must use the syntax *name=value, ...* For example, an annotation that takes one `String` element could be used either of these ways:

```
@OneItemAnnotation("XYZ") class Foo { ... }

@OneItemAnnotation(value="XYZ") class Foo { ... }
```

If the value is an array, you can use the literal array notation like this:

```
@Reviewers( {"Moe", "Larry", "Curly"} ) class Foo {...}
```

(*Show example on Website.*)

Defining Your Own Annotations

To define your own annotation, you create a special type of interface. An annotation type with no elements is termed a marker annotation type. Each ***element*** (a *property*) of an annotation is defined just like a method declaration, but don't let that fool you. Element declarations must not have any parameters, generics, or `throws` clauses; they're not really methods at all.

Element types are restricted to primitives plus `String`, `Class`, `enum`, `annotation`, and arrays of those types.

```
@interface AnnotationName {}  // A marker annotation
                              // with no content
```

In annotations with a single element, the one element should be named **`value`**:

```
@interface AnnotationName {
   elementType value();
}
```

(You can call an element anything you like; but the name `value` is treated specially so you should stick with this convention.)

Annotations can have as many elements as you like:

```
@interface AnnotationName {
   elementType elementName1();
   elementType elementName2(); // ...
}
```

Again, you can name these anything. If your annotation has one commonly used element, you should name it `value`.

You can specify default values for elements too:

```
@interface AnnotationName {
   elementType elementName() default literal;
   ...
}
```

You can specify if Java doc comments should show the annotation or not:

```
@Documented
public @interface RFE {
   int    id();
   String description();
   String assignedTo() default "[unassigned]";
   String dateRequested() default "[unknown]";
}
```

> Note, only things that show up in the Java doc (HTML) output can show annotations. That is, the annotations on local variables (or private members) won't appear, since those don't get processed (by default) by `javadoc`.

Here'a a use of the `RFE` annotation:

```
@RFE( id = 123, description = "Enable telepathy",
 assignedTo = "Hymie Piffl",  date = "2/30/2010" )
public static void readMinds () { ... }
```

When defining your own annotations, you can add other annotations to them, to define their properties. These additional annotations are:

- Some annotations are for internal use only. Other annotations are part of the public interface of the class (or whatever) and users of your code need to know about them. To cause `javadoc` (or other similar tools) to include an annotation in the generated documentation, annotate your annotation with **`@Documented`**. (Remember, if you annotate something within a method, it won't show in the Java docs regardless.)

- You use **`@Target`** to restrict where your annotation can be applied. The default is that anything can be annotated. But you can restrict your annotation to local variables, fields, methods, classes, packages, method parameters, etc. (See `java.lang.annotation.ElementType`.) For example, to restrict some annotation to be used only on methods, use:

```
@Target( ElementType.METHOD )
@interface MyMethodAnnotation { ... }
```

- **`@Inherited`** shows that the annotation may be inherited (i.e., mark the parent with the annotation, and the child will show as having it too).
- **`@Repeatable`** (new in Java 8) This means the annotation can be used multiple times on the same thing. For example:

```
@Reviewer("Moe")
@Reviewer("Larry")
@Reviewer("Curly")
class Foo { ... }
```

  To allow this, you need to declare the `@Reviewer` annotation with `@Repeatable`. **In addition, you need to declare the element `value` as an array.** The whole thing would look like this:

```
@Repeatable
public @interface Reviewer {
   String[] value ();
}
```

- You can control if your annotation will be part of the `Class` object loaded into RAM (**`RUNTIME`**), in the `.class` file but not the `Class` object (**`CLASS`**), or just in the source code and not in the `.class` file (**`SOURCE`**). This is known as the *retention policy*. `CLASS` is the default. For example:

```
@Retention( RetentionPolicy.SOURCE )  // ibid
```

To use *reflection* to check for annotations, the policy must be `RUNTIME`.

Here's one last example, inspired by some discussions on the `comp.lang.java.programmer` newsgroup (11-12, 2013).

When using `@Override`, there is no element to say which class's or interface's method you are overriding. If your class implements several interfaces, and/or you have a deep inheritance tree, it can be difficult to remember which one you were attempting to override when you get a compiler error. Of course, you could just put that information in a method's comment; Java doc will show it as well. But we can also create a new annotation to record from where an overridden method originates:

```
@Documented
@Target(ElementType.METHOD)
//@Retention(RetentionPolicy.RUNTIME) //For Reflection
@interface From {
   Class<?> value();
}
```

And use it like so:

```
@From(Foo.class) @Override public void aMethod(){}
```

Annotation Processing

Code with annotations can be processed to perform various tasks, including code checking (throwing compiler errors or warnings if the checks fail) and code generation. In previous versions of the JDK, you needed to use the tool `apt` (*annotation processing tool*), but current Java supports annotation processing from `javac`, which includes several standard options to control the process.

Processors are located by means of service provider-configuration files named `META-INF/services/jakarta.annotation.processing.Processor` on the search path (CLASSPATH by default, although annotation processors can be listed on the command line or found using a separate PATH setting). (No doubt the weird name is based on the expectation that annotation processors will be packaged in Jar files.) Such files should contain the names of any annotation processors to be used, listed one per line. Alternatively, processors can be specified explicitly, using the javac "`-processor`" option.

Taking advantage of Java's new annotation abilities, U of Wash. released a Checker Framework with many, many annotations that are quite useful; Oracle recommends them. To use, you install the annotation framework (or use an IDE plugin) and tell the compiler to process them. To use it automatically with `javac`, you need to add some Jar files to your CLASSPATH (or specify a *processorpath* on the command line); currently this is done by providing a custom `javac` command (that internally calls your real javac).

> It is simpler than it sounds: unzip the archive someplace and add the resulting ...`\bin` folder to the FRONT of PATH. The Checker Framework's `javac.bat` file invokes their Java program with your arguments. Their Java program reads your command line and does all sorts of compile time checks (and possibly modifies the source to include additional runtime checks), before invoking the default `javac` command.

After scanning the source files and classes on the command line to determine what annotations are present, the compiler queries the (annotation) processors found to determine what annotations they each process. When a match is found, the

processor is called. A processor can claim the annotations it processes, in which case no further attempt is made to find other processors for those same annotations. If any processors generate new source files, then another round of annotation processing occurs: Any newly generated source files are scanned, and the annotations processed as before. This continues until no new source files are generated. (Modifying the current `.java` file is not possible; however, annotation processors can modify generated `.class` files using tools such as JavaAssist or Byte Buddy.)

See How Annotation Processing Works in the Java 8 SE docs, or this 3-part tutorial for more info.

Some of the Checker Framework annotations that are especially useful include:

- `@NonNull` – The compiler can determine cases where a code path might receive a null value, without ever having to debug a NullPointerException.
- `@ReadOnly` – The compiler will flag any attempt to change the object. This is similar to Collections.unmodifiableList, but more general and verified at compile time.
- `@Regex` – Provides compile-time verification that a `String` intended to be used as a regular expression is a properly formatted regular expression.
- `@Tainted` and `@Untainted` – Identity types of data that should not be used together, such as remote user input being used in system commands, or sensitive information in log streams.
- `@m` – Units of measure ensures that numbers used for measuring are used and compared correctly (same units), or have undergone the proper unit conversion. (TODO: Add a demo of this to class resources.)
- `@Format` – Provides compile-time verification of format strings used with `printf` and similar methods.

Covariant Return Types

Normally when *overriding* inherited methods, not only must the signature match but the return type as well (the so-called *full signature*). **Covariant return types** since Java 5 let you override a superclass method with the same signature (still), but with any return type that is a subtype of the superclass method's return type (in other words, "is-a" the parent's return type). (This is harder to say than to use.)

About the only use of this is for `clone` but it can be used elsewhere. Here's an example of overriding showing the use of covariant return types:

```
class Employee {
  public Employee aMethod (...) { ... }
}
class Manager extends Employee {
```

```
@Override
public Manager aMethod (...) { ... }
```

Varargs — *Since Java 5*

Varargs means a variable number of arguments. Some methods such as `printf` require this. In the past, a Java programmer had to bundle variable arguments in an array of Objects and pass that (so the number of arguments was not variable).

It is true that multiple arguments must still be passed in an array, but the Java 5 varargs feature automates and hides the process. So, for example, the `java.text.MessageFormat.format` method had this declaration:

```
public static String format ( String pattern,
    Object[] arguments );
```

but with varargs the method can also be declared like this:

```
public static String format ( String pattern,
    Object... arguments );
```

The three periods after the final parameter's type (an "ellipsis") shows the final argument may be passed either as an array or as a series of arguments. Java will bundle the actual arguments into an array and pass that. So you can invoke this method simply as "`format("string", arg1, arg2, arg3);`"

Varargs can be used only in the final argument position. (That is, any required arguments are listed first.)

Any type can be used, not just `Object`:

```
void aMethod( String s, int...)
```

You can use varargs to declare `main`:

```
public static void main ( String... args)
```

(See also `String.format` for another example.)

(*Show `VarArgs.java` demo.*)

(The following was taken from Joshua Bloch's *Effective Java.*) The only issue to be careful about is when using varargs with generic type arguments, which would look something like this:

```
static void dangerous ( List<String>... lists )
{ ... }
```

This is also dangerous:

```
static <T> T[] makeArray( T... args ) { return
args; }
```

This is because arrays don't know about generics. Inside of the method body, if you add any items to the array or return a reference to the array or one of its elements, the compiler won't have sufficient information to know if you (or whoever calls your code) did that safely or not. If not, a `ClassCastException` will be thrown when code access the added element. Because of this, the compiler will generate weird-looking warnings on every method invocation and the declaration.

If you do use varargs with generic types, then your method body must never try to assign into the array or return a reference from the array. If you follow that rule, your use is safe. Since the compiler cannot tell if your method body is safe or not, it always generates the annoying warnings. In Java 7, the **`@SafeVarargs`** annotation was added to the language, so you can tell the compiler "trust me, this method is safe!" and the compile won't generate the warnings.

Here's an example of a safe use of varargs with generics: you pass in a number of lists and a single list of all the elements is returned, an operation known as *flattening*:

```
@SafeVarargs
static <T> List<T> flatten ( List<? extends T>...
lists ) {
   List<T> result = new ArrayList<>();
   for ( List<? extends T> list : lists )
      result.addAll( list );
   return result;
}
```

(The notation "`List<? extends T>`" just means a `List` of type `T` items or items that extend `T`.)

Reflection

For the most part Java programs are written with class names and methods known at the time you write the code. But in some situations, you need to write library code that instantiates classes and invokes methods, when the exact class or method name isn't known until runtime! This happens when *frameworks* must use call-backs, when supporting plugins, when creating tools (e.g., testing), and in a few other situations. Using reflection, you can examine classes, instantiate objects, and create `Method` objects and pass them around (and eventually, invoke them).

> Complex code with dependencies (coupling) between classes tends to evolve over time. Such systems can use IoC (*inversion of control*) patterns and frameworks such as Spring, to obtain references to actual objects needed by other classes. A type of IoC is the factory pattern, another is called *dependency injection* (DI). Often, DI uses reflection to determine a class's

correct constructor arguments, based on configuration files that you can change even while the program is running. DI was integrated into Java 8, however no implementation is provided by Oracle's JRE; you need to add a DI framework (usually, downloading a Jar file on CLASSPATH is sufficient).

These design patterns are discussed later in the course.

Java provides a *reflection* library for this, **`java.lang.reflect.*`**. These classes can analyze a class given just the class name. An example is Jakarta EE framework, in which an application server must determine the methods and properties of *JavaBeans*.

Reflection is powerful but slower than invoking methods directly on objects known at compile time. Besides implementing IDEs such as Eclipse, reflection can be used to write generic array handling methods, manage *cross-cutting* dependencies (such as references to objects commonly used in many/most classes, such as a logger or a messaging service).

Even more rarely used than simple reflection is Proxies. A `Proxy` can be used to define new classes at runtime that implement some set of interfaces. An example might be a proxy that provides tracing, then invokes the interfaces' methods. You then use the proxy object in place of any original.

The `Class` Class

The **`getClass`** method returns the class for any object. You can also use the **`.class`** property that all classes have, e.g., "`Date.class`" is a `Class` object for class `Date`. (**Show java.lang.Class.getXYZ methods.**) Note primitive types have classes too (sort of), so "`int.class`", "`float.class`", etc. all work.

This `Class` object has several useful methods: `getName()` to print a class's name and **`getResource()`** to locate and return a URL for some resource (a non-class file, such as a graphic). There is also a static **`forName()`** method to load into memory any class given its name.

```
Object obj = new java.util.Date();
Class cl = obj.getClass()
System.out.println( cl.getName() );
```

Given a class object you can create instances with `Class.newInstance()`:

```
Object construct ( String className )  {
   try {
      Class<?> cl = Class.forName( className );
      return cl.getDeclaredConstructor().newInstance();
```

```
    } catch ( ClassNotFoundException e ) { ... }
}
```

Three points: `forName` can throw checked exception, so use a try block. `getDeclaredConstructor` uses the constructor that matches the parameter profile passed as arguments; here, the *no-arg* constructor is used. Finally, note "`Class<?>`": this is *generics* meaning a class of any type. (Without the "`<?>`" you would get a compiler warning about using raw types.)

Show the classes in `java.lang.reflect`: `Constructor`, `Method`, and `Field`. These are created with various `get`*`XYZ`* methods of class `Class` (show).

The class type names are the internal *mangled* names that use a special code to name class types (documented in the JNI spec):

B (byte), C (char), D (double), F (float), I (int), J (long), S (short), Z (boolean), L*classname*; (object types), [*type* (array of *type*). I think "V" for void is missing from that chart.

(**Show `ReflectTest.java`**.)

```
obj = new Double[5]();
System.out.println(obj.getName());
```

Method names use this mangling as well, for example "`f(ILjava/lang/String;)V`" means "`void f (int, String)`". The JDK includes a tool to show the mangled names (used by JNI, the *Java Native Interface*) of the members of any class, for example:

```
javap -s java.awt.Label
```

Java 7 adds more power and convenience with additional classes in the `java.lang.invoke` package, such as class `MethodHandle` and `LambdaMetaFactory`.

> Note that reflection doesn't work well with lambdas. There is a hack that uses undocumented features of Oracle's JDK, but for now, you cannot create a `Method` object for a lambda, and there is no `Lambda` class in `java.lang.reflect`.

Lecture 6 — I/O and Files (Persistent Storage)

Data is kept in memory is not *persistent* (we say it is *volatile*). When a JVM (or any) process ends, all the data held in that process' memory is lost. For longer periods of time than a single run of some program, data must be copied to non-volatile, or *persistent*, storage. Once an application restarts, it can read the data it needs from storage back into memory for fast access.

Today we use disk storage, which stores information in named *files*. Even a database management system ultimately stores data in files. Developers need to know how to access such storage. There are many different ways, each with different pluses and minuses.

Persistent storage needs to support the basic operations of *create*, *read*, *update*, and *delete*. These are referred to as **CRUD**. Many applications are little more than a user interface and some persistent data; these may be called CRUD (but not cruddy :-) applications. There are various standard ways to access such data, including directly accessing files, using a (object) storage management system, and using a data server or data management service. Here we will focus on accessing files directly; databases are discussed afterward.

To read the data directly from files, a programmer must know the *format* of the files: line-oriented text, XML, HTML, CSV, GIF, JAR/ZIP, etc. (If text, you must know the encoding as well.) If care isn't taken when writing the files, it will not be possible to read the data later.

A common task of programmers is designing file formats (and documenting them). There are thousands of file formats in common use today, and many applications use new ones that are convenient and efficient for that particular application.

> Related to file format design is *message format* design, where you must design and document the format of messages passed on a network as part of some protocol.

Some simple choices are plain text files, XML files, JSON, YAML, and TOML (all of which are *structured* text formats), binary files, and database files. **When possible, choose a text (non-binary) file format**. Such files are easy for humans to read, making the task of debugging simpler. It is easier to have processes exchange data as text, avoiding many hardware issues (byte orderings: big-endian, little-endian, and others).

A common text file format is to have one line per record, with a delimiter character between fields (e.g., CSV or TSV format), although there are many definitions for these such as RFC-4180 for CSV. Often the first line is the field names.

Sometimes a non-text character is used as a delimiter, so no quoting of fields is needed. While simple and easy to work with, mistakes in the data can be difficult to detect automatically.

> Text files for transmission on the Internet are defined by old RFCs that state text files have lines terminated by a CR LF sequence. However, POSIX defines a text file as a file that doesn't contain any NUL byte, each line ends in a NL character, with a max line length (of about 2k; that is not standardized). While POSIX defines ASCII text files as those text files with data limited to legal ASCII bytes, today text files may be encoded as UTF-8 (common and most likely to work) or as some other encoding. (This whole issue is confusing; I wish the world would just standardize on UTF-8. Java did in Java 18!)

Another possible format for text data is XML. XML is a text format designed as a data interchange format (that is, to pass data between systems), but is not line oriented. Instead, the records and fields are delimited with *tags*. Using XML is powerful since it is easy to automate the checking of these files (thus, errors such as failing to quote a delimiter character are avoided, or at least easily detected) and to describe the format (using a DTD or schema) in a machine-readable way, making automatic file validation possible. Thus, XML files are *self-describing*. XML files have the nice feature that no matter what the format, you use the same API to read and write these files. Some popular text editors support DTDs (or XML Schemas) and make it easy to edit such files.

A downside is the formats tend to be more complex and harder to work with (by humans), and more verbose. Here's an example of XML:

```
<person id="12345">
  <name>Hymie</name>       tag data is: Hymie
  <nickname></nickname>    OR: <nickname/>
</person>
```

> See Java XML processing - JAXP security guide for information on how to enable necessary security features when processing XML.

Another popular choice is **JSON** (*JavaScript Object Notation*), first used for "AJAX" (a.k.a. "Web 2.0") programming as a data interchange format. This is a fairly well-defined text format (but like XML, not line oriented) defined by RFC 7159. See also JSON.org. It has been described as "The Fat-Free Alternative to XML"; meaning an easier to work with format. It is a list of name/value pairs, where the value can be another JSON object, a string or number, or an ordered list. Firefox uses JSON to backup bookmarks. Here's an example (from Adobe Labs):

```
{ "id": "0001", "type": "donut", "name": "Cake",
 "ppu": 0.55, "batters": {
```

```
    "batter":
    [{ "id": "1001", "type": "Regular" },
     { "id": "1002", "type": "Chocolate" },
     { "id": "1003", "type": "Blueberry" },
    ] },
  "topping":[
    { "id": "5001", "type": "None" },
    { "id": "5002", "type": "Glazed" },
    { "id": "5007", "type": "Powdered Sugar" },
    { "id": "5006",
      "type": "Chocolate with Sprinkles" },
    { "id": "5003", "type": "Chocolate" },
   ]
}
```

JSON support is included in Jakarta EE ("JAX-RS") but you need a 3rd party library for Java SE. Popular choices include **google-gson**, JSON.Simple, and Jackson. (Class resources have demos for gson, org.json, and Jackson.) (*TODO: add JEE JAS-RS demo.*) Note Jackson is very powerful, fast, and popular, but it can be complex to use.

(Of course, you can always write your own library to read and write such files.)

> JSON is related to No-SQL data management systems, which use non-RDBMS to handle more efficiently certain types of data common on the Internet, and which do not need the guarantees relational DBs provide.

Although easier for humans than various XML formats, JSON is not as simple as some other formats. (More details on XML and JSON can be found in the class resources.)

If you find you don't want the structure of XML or JSON, but still need structured data that is easy for both humans and machines to use, you can try YAML or TOML. (See this short YAML tutorial.)

All structured text has security issues. XML External Entity processing should be disabled. CSV fields that start with certain characters such as "=" or "*" (even if preceded by whitespace) can cause processing engines such as Microsoft's Excel to run arbitrary code or even send files on your disk to a remote server! JSON, YAML, and TOML have fewer security issues but still have some.

Text Encodings

Depending on what you want to input or output, the data may need conversion. This is because Java uses Unicode (UTF-16) internally for text, but most OSes store text files in different formats: **ASCII**, **ISO-8859-1**, **CP-1252** (on Windows),

and **UTF-8**. These are known as the file's ***encoding***, and there are hundreds of different ones in use.

> Early versions of Java used USC-2 internally and only supported Unicode characters with codepoint values less than 64K. With Java 5, Strings changed to use UTF-16, which uses 2 or 4 bytes per character.
>
> Since Java 9 and forward, the JVM can optimize `Strings` using a new Java feature called *compact strings*. Instead of having a `char[]` array, `String` is now represented as a `byte[]` array. Depending on which characters it contains, it will be created with either ISO-Latin-1 encoding (if all characters in the string has codepoints less than 256) or UTF-16 encoding (for all other strings), to produce either one or two (or four) bytes per character.
>
> Note these optimizations are invisible to the programmer; the language specifies UTF-16. But older `String` and `Character` methods assume exactly two bytes per character in the internal storage. Because of that, programmers should always use the newer character and codepoint methods to work with `Strings`.

You can easily convert Strings to/from other encodings, such as UTF-8:

```
// Java String to UTF-8:
byte[] utf8data = "my string".getBytes("UTF-8");
// UTF-8 to Java String:
String myString = new String(utf8data, "UTF-8");
```

Because the encoding process can also be seen as a mapping between characters and bytes, the term ***charset (the preferred term in Java)*** or sometimes *charmap* is used instead of encoding. The names of encodings/charmaps/charsets are case-insensitive. Each one has a canonical name and zero or more aliases; your programs can use any alias. Java guarantees a small set of standard encodings, but most JREs support many more. (*Show CharsetDemo.java*.)

> Besides converting between bytes and characters (the encoding), data may need additional conversions depending on the format, for example, **XML**, **JSON**, **zip** (compressed), **encrypted**, etc. And that doesn't count conversions of data units (kilometers to miles, Fahrenheit to Celsius, and so on).

Conversion between Unicode text and encoded text files is done using **`InputStreamReader`** and **`OutputStreamWriter`** classes. All the classes in the `java.io` package whose name ends with "`Reader`" or "`Writer`" convert between Java characters and bytes; the classes whose names end in "`Stream`" do not.

When creating a new Reader or Writer, you can pass a `Charset` as a constructor argument. If you don't, the OS's native encoding will be used thropugh Java 17. Java 18 changed the default to UTF-8 (yea!). (Show encodings from **`java.nio.charset`**.)

> More details about encodings, Unicode, and related matters are provided later in the course. For now, either use your OS's native encoding or (my preference) use UTF-8 everywhere. Note that as of version 18, Java defaults to UTF-8 for file I/O regardless of the operating system settings.

Binary file formats can be the most efficient to work with and are certainly most appropriate when the data itself is not easy to translate to text (e.g., sound and graphics). However, such files are difficult to debug, read, and write. Also, binary files often can be encoded as required (say to text) before transmission across a network (as email attachments are encoded as text, for example). Finally, binary files can suffer from byte order issues.

> Text files can also suffer byte order problems if not using an encoding that uses single byte code units. This is why some Unicode files begin with a *byte order mark*, which sometimes incorrectly get shown by web browsers. Also, binary data such as images and icons must be encoded as text; **base-64** is often used. Such encoding increases the size of the data by about a third.
>
> An additional issue with text formats is number representation; CSV does not distinguish between numbers and text that happens to be all digits. JSON does, but has only one number type with no difference between Integers and floating-point values. This is a problem with dealing with large Integers. JSON only fully supports Integers less than 2^{53} (about 17 digits). This is because internally they are represented as type double which has a 53-digit mantissa. This became a problem with Twitter for instance, which uses 64-bit numbers to identify tweets.

Because of the space (and also time when moving data across a network) overhead of text formats, in addition to the issues shown in the box above, binary formats are also common. One possibility is to simply compress the text format. Another is a special binary encoding designed to save space. Apache Thrift, originally developed by Facebook, and Apache Protocol Buffers (*protobuf*), originally developed by Google, are examples. These come with standard libraries for defining a schema for your binary data, and to read/write such data given a schema.

> Having a schema for data is usually a good idea, since otherwise changes to the data ("schema evolution") are only known by the code that reads/writes it, and mistakes become common.

There's really a lot more to the story of data storage. Today, data is often stored in cloud-based storage systems such as Amazon's S3, a type of key-value store, that typically uses Amazon Athena for queries. Such queries costs money depending on the amount of data accessed, so data formats are chosen to minimize the amount of data accessed for a given query. A very popular format that does this is Parquet. When not using the cloud, you can forgo traditional filesystems and use a distributed object store for your disks, such as Ceph.

> There are three popular binary formats today: Parquet, Avro, and ORC (Optimized Row Columnar), all open source Apache projects. These file formats allow splitting the files across multiple disks, store the data's schema in the files themselves (*self-describing data*), and are byte-order insensitive. (*Show Avro demo.*)
>
> A *columnar-storage* file format is popular for large key-value data sets (that use schema-at-read query engines such as Apache Drill). Parquet and ORC both store data in columns, which makes them high-performance for read-heavy uses, and cost-effective when using cloud storage. Avro is row-oriented and best suited to write-heavy uses.

Magic Numbers are a string of bytes normally found in the beginning of binary files. They are sometimes known as *file signatures*. These should be unique so they identify the file format. (Text files often have a first line that does the same job.) While there is no authoritative registry for these, you can find a list in wikipedia, GitHub. and elsewhere.

Here's a good list of rules for creating magic numbers for your own binary file formats (Taken from Zack Weinberg's blog post):

- MUST be the very first bytes in the file
- MUST be at least four bytes long, eight is better
- MUST include at least one byte with the high bit set
- MUST include a byte sequence that is invalid UTF-8
- MUST include a zero byte
- MUST be followed by a version number.

Here's a template you can follow for a good 8-byte magic number:

0xDC 0xDF X X x x (0x01 0x00 or 0x00 0x01)

Explanation: 0xDC 0xDF are bytes with the high bit set. Together with the next two bytes, they form a four-byte sequence that cannot appear in any valid ASCII, UTF-8, Corrected UTF-8, or UTF-16 (regardless of endianness) text document, and regardless of a BOM.

X X x x: Four ASCII alphanumeric characters naming your file format. Make them clearly related to your recommended file name extension. Four characters

because The world is running out of useful three-letter acronyms. If you don't need four characters, pad at the end with 0x1A (aka ^Z). The first two of these (the uppercase Xes) must not have their high bits set, lest the "this is not text" declaration be weakened. For the other two (lowercase xes), use of ASCII alphanumerics is just a strong recommendation.

0x01 0x00 or 0x00 0x01: This is to be understood as a 16-bit unsigned integer in your choice of little- or big-endian order. It serves three functions:

(1) It includes a zero byte, reinforcing the declaration that this is not a text file. (2) It demonstrates which byte ordering will be used throughout the file. (It does not matter which order you choose, but you need to consciously choose either big- or little-endian and then use that byte order consistently throughout the file.) (3) If one day you discover that you need to alter the file format in a totally incompatible way, and yet it is still meaningfully the same format, so you don't want to change the name characters, then you can change the 0x01 to 0x02.

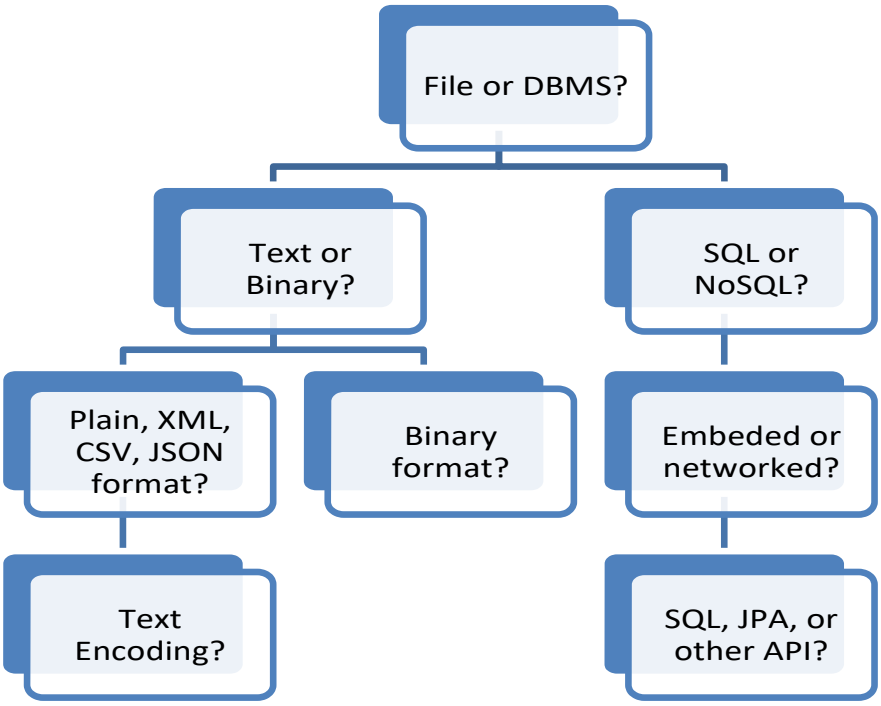
Many real-world file formats do not follow these rules. For example, Java class files have a four byte magic number whose hex values spell out "CAFEBABE".

DBMS A different choice is to use a (possibly SQL) database (database management system), and store data as records and fields of (standard) SQL data types. Then you can let the DBMS worry about file formats!

Using a DBMS to manage your persistent data has many benefits. It is common that files might be accessed by several processes at the same time; a DBMS ensures the data doesn't become corrupted in this case. SQL permits "ad-hoc" queries against the data; with regular files you would need to write or modify programs to do that. DBMSs support an authorization and authentication system to protect the data, and can provide audit logs (showing who did what and when). A properly tuned DB can be very efficient (special indexes are kept, and special caching is done). (Keep in mind not all DBMSs are "relational" nor do they all support SQL.)

> Many factors go into the decision of file formats: how much data is there? Must it travel across the Internet? How important is automatic processing? How important is human comprehension? Will you need transactions, access control, or other DBMS features? Will other (perhaps legacy) applications need to read the files (e.g., a report generator)?

Persistent Data Choices



Here we will concentrate on direct access to text and binary files, both local storage and across a network. Databases will be discussed later.

Once you have decided on a basic format for your files, there are other considerations:

- Have a header (first line for text files) that defines the type of file. File names change, so you need to be able to identify your file format regardless of the file name.
- Add a version number to the header. File formats may change over time, so having a version number is important.
- Consider what happens to your data if transmitted over a network (serialized). Such a process reduces your data to a sequence of bytes, which may or may not be reassembled in the correct order at the remote end. (This issue is known as *byte order*, and it comes up with binary formats, or multi-byte text encodings.)

Security Considerations

Data is precious. You need to protect it. You often have legal obligations to do so, such as for *personal identifying* (PI) data, for medical data, and financial data. These considerations apply to any persistent storage of data, not just files. The main concerns are unauthorized access (*data breaches*), and data modifications (corruption). Consider the security needs of your data:

- Does access to the data need to be audited?
- Do changes to the data need to be authorized?
- Must the data be time stamped?

- Should the records and/or the whole file be digitally signed?
- Should a checksum (such as CityHash or SpookyHash) be used to ensure data validity (i.e., to detect data corruption)? Note these do not protect against attackers, you would need a cryptographically secure hash for that, such as SHA-3.
- Should any part of the data be encrypted?

To address these issues, you should:

- **Consider having proper access controls**. Set up user accounts and permissions, so only the proper authenticated and authorized users (or processes) can read or write the data. Tote the data must be protected from other users and processes on your servers. Many operating systems provide a permissions system to prevent such access; be sure to use it!
- **Consider encrypting the data**. This is required for many types of data. (The data must be kept encrypting when traversing networks as well.) This can protect the data from other users and processes on your server.
- **Consider not storing data**. For example, in many applications it is illegal (in a regulatory sense) to store more than the last six digits of a credit card number unencrypted, as per the PCI-DSS standards.)
- **Consider regulatory and legal requirements** for storing data. You might not need it for your application, but industry or government regulations or best practices may require storing additional data. (See the GDPR box, below.)
- **Consider how long to retain data**. Again, there are legal requirements that mandate you retain some data for a given time or forbid you from retaining data after a period of time. (To achieve this goal, you often need to store a timestamp of when the data was added and/or last changed.)
- **Consider storing a message digest or digital signature** for each record (or whole data sets, if efficiency is a concern). A message digest such as SHA-256 can prevent or at least detect data corruption. A digital signature does that, as well as prove who last modified the data.
- **Validate the data source.** Many formats, both text and binary, require instantiating arbitrary classes in order to reconstruct objects in memory. If attackers can pass modified data to your code, they can cause your program to instantiate any classes they want, which in turn may allow them to run arbitrary code (CWE-502). If the data is not digitally signed, your code should validate the file and only allow certain allow-listed classes to be instantiated.
- **All external data read or written by your code should be normalized (if text), sanitized, and validated, before use.** How to do this is discussed later in the course.
- **Be sure your system handles weird data!** Names are especially problematic. Some people have only one name, some have 4 or more. The family name isn't

always last. Names can contain accent marks, hyphens, spaces, numbers, and more. Some names are a single letter (many validation systems think the user entered an initial). Some names are very long.

> I read (2/2025) of a man working in IT security, who ordered a vanity license plate *null*. After he got a parking ticket, he started to receive hundreds of tickets for unrelated cars! Apparently traffic police were in the habit of typing "null" if the license plate was unknown, and their system associated all of the citations with this guy. Seems their system couldn't tell the difference between a database null value and the string `null`. (*Show little Bobby Tables comic.*)
>
> Tip: Not sure how careful Google is, but if you ever open you own restaurant, consider naming it "Restaurants near me".

- **Document your data policies**, including the procedure to request copies (law enforcement needs that) and restore-from-backup requests.
- Besides using a data management system or files, you could **use *cloud storage***.

> **Consider the requirements for personal data under the GDPR.** You can only collect and keep personal data with the user's consent. Also, with few exceptions data can only be used for the single purpose it was collected for. Thus software developers must **document the purpose of any persistent personal data obtained, along with the date and manner of consent**. For example, if you collect a user's phone number or email for the purpose of two-factor authentication, it is illegal to use that info to send text alerts later. You would need to get additional consent for that use. Additionally, you must provide a way for customers to withdraw consent. These requirements mean additional data must be kept, since it can be audited anytime.
>
> This is a big (expensive) change for many companies who currently do not track consent (and may have bought email lists, which was legal before GDPR). Several large companies have opted to block their services in Europe, as they fear updating all their databases and policies would prove costlier than losing all those customers.
>
> A similar measure is CCPA (AB-375), a law affecting companies doing business in California (and thus the whole US). This law is effective 1/1/2020.
>
> Although Brittan has left the EU, They have an equivalent law (that is based on GDPR) and is often called the UK GDPR.

java.io — Accessing Data as Streams of Bytes

The Java input and output classes are in packages **`java.io`**, with additional classes used with I/O in `java.net`, (e.g., `URL`), `java.util` (e.g., `Properties`, `Scanner`, `Formatter`), and others.

Java I/O uses a ***stream*** abstraction: data is converted to a sequence of bytes and moved to/from your program and external data *sources* and *sinks* (files, stdio, network connections, etc.). All I/O is viewed as the movement of single bytes, one at a time, through an object called a Stream. Stream I/O is used for contacting the outside world. It is also used internally, for turning objects into bytes and then back into objects.

To perform I/O, a program ***opens*** a stream, which causes the JVM (and OS) to allocate some buffers for it and a *file handle*. Only after opening a stream can your program read from or write to it. When finished using the stream, you must ***close*** it. This *flushes* (writes) the data still cached in memory to the data sink and then frees up the resources that were reserved for that stream.

Data can be passed into your program via an **`InputStream`** or passed out of your program via an **`OutputStream`**. (*Show and discuss the methods.*) These are the most basic I/O classes (and are abstract). Streams can only be accessed in sequential order. That is, to see the last character of a file that is read from an `InputStream` the entire file must be read first.

The source (or destination) of a stream need not be a file. It could be a network connection, a database, or even a RAM buffer.

> When reading data to memory (or from storage), the OS can signal your process it has reached the end of the data, usually called the ***end of file*** or **EOF**. A Java programmer doesn't have to worry about how the OS does that; the `java.io` classes in the JRE know when EOF is reached and will in turn signal your program in a standard Java way.
>
> When typing input at the keyboard (using *standard input*) you can simulate EOF by typing ^Z on DOS systems and ^D on others. Due to buffering on some systems, you may also need to hit enter.

Using `InputStream` and `OutputStream`

The input stream interface defines the method **`int read()`**, which reads one byte of data if any is available (if not the method returns "`-1`" to indicate EOF). Similarly, for output streams there is a **`void write( int b )`** method (takes an `int` but only writes the least significant `byte`).

Reading and writing data can *block* a thread until the operation completes; there is no automatic time-out. (Other threads can still run.)

Another useful method for input is **`available()`** which returns the number of bytes currently buffered by the OS (and thus available for reading by your program without blocking). You can use another (overloaded) `read` method to read all available data without blocking like this:

```
int numBytesAvailable = in.available();
if ( numBytesAvailable > 0 )
{  byte[] data = new byte[numBytesAvailable];
   in.read( data );
}
```

When finished reading or writing a stream, you need to **`close`** it. This will both **`flush`** the stream (empty any buffers) and free up the stream's resources in the OS. (Until a stream is flushed, the data written to it may still only reside in RAM!)

Decorator Design Pattern (as used with java.io classes)

The **`java.io`** package has many classes that can be used to provide conversions or provide additional features for performance (such as **`BufferedReader`**) or convenience (such as **`PrintWriter`**`.println`).

As with many complex problems, the solution is often simplified by breaking a big problem up into smaller problems, each of which can be easily solved and implemented (and tested), then combined. The ***Decorator pattern*** is the object-oriented version of this ***decomposition*** idea and used for tasks such as I/O. Using streams of data, decorators act as *filters* on the stream of data.

Java I/O is designed as a few simple *byte-at-a-time* input/output classes, plus many decorator classes. The different decorator classes may be mixed to combine the various conversions and features. For example, you can open a text file for writing:

```
FileOutputStream fos =
  new FileOutputStream("outFile.txt"); // byte stream

OutputStreamWriter osw =   // character stream
  new OutputStreamWriter(fos, "UTF-8" );

PrintWriter out =
  new PrintWriter( osw ); // add convenience
```

Normally you would add these decorators in one step:

```
PrintWriter out = new PrintWriter(
   new OutputStreamWriter(
      new FileOutputStream("outFile.txt"), "UTF-8" ) );
```

This can be re-written using a convenience class to create a writer (character) stream from a file:

```
PrintWriter out = new PrintWriter(
  new FileWriter("outFile.txt", StandardCharsets.UTF_8)
);
```

You can input/output whole objects or use encrypted data by adding different decorators. (Review `DataOutput` and `PrintWriter` methods.) To efficiently read character input from a GZIPed (compressed) file, just combine the correct decorators: one for buffering (adds efficiency), and one to decompress the bytes, and one for character-at-a-time (converts characters to bytes). The code looks like this:

```
try
{ InputStreamReader in = new InputStreamReader(
    new GZipInputStream(
        new BufferedInputStream(
          new FileInputStream( "somefile.zip" )
        )
    ), "UTF-8"
  );
}
catch( IOException e )
{  System.err.println( "Cannot open \"file.name\"." );
   e.printStackTrace();
}
```

This idea is very powerful and flexible (and not just for I/O; consider Java Window: use decoators to add borders, a menu bar, title bar, scroll bars, etc.). You just mix and match the decorators to get the feature mix you need. That's because each I/O decorator is designed to take an `InputStream` as an argument and return an `InputStream`. Each decorator is relatively simple because it does only one thing. Thus decorators are easy to write, debug, and modify independently from the others.

You must **be careful when using chained method calls that create new objects, rather than return a reference to a single updated object**. All those objects take up resources and won't be closed unless and until the garbage collector runs. For example:

```
PrintWriter out = new PrintWriter(
  new OutputStreamWriter(
    new FileOutputStream( "outFile.txt" ),
       "UTF-8" ) );
```

With this code, there is no easy way to close the `OutputStreamWriter` or the `FileOutputStream` objects once you are done with them. Either make sure the GC runs periodically (or manually), or use code such as this:

```
FileOutputStream fos =
   new FileOutputStream( "outFile.txt" );
OutputStreamWriter osw =
   new OutputStreamWriter( fos, "UTF-8" );
PrintWriter out = new PrintWriter( osw );
...
out.close(); osw.close(); fos.close();
```

Newer Java has enhanced the `try` block and added the interface `Autocloseable`, to address this issue.

Ideally each decorator class is completely independent from the others, so that the decorators can be mixed in any order for the same effect. Often however there are constraints on the order of decorators, and some don't make sense together (but even so it's a great idea).

Scanner and Formatter

Java5 has added new powerful I/O classes (in `java.util` package): a **`Scanner`** can read from any (character) input source, parse the text, and convert the results to primitive types. Here's an example of reading a text file containing lots of Integers separated by whitespace:

```
Scanner sc = new Scanner( new File("nums.txt") );
while ( sc.hasNextInt() )
{  int num = sc.nextInt();  }
```

The source for a `Scanner` can be a `File`, `Path`, `InputStream`, `InputReader`, or even a `String`. You can control the delimiters used too.

Qu: What does this "almost a one-liner" do:

```
// Run using Java 23 or newer, as:
//   java --enable-preview Demo.java

void main () throws Exception{
  String data = new Scanner(
    new FileInputStream( "data.txt" ) )
      .useDelimiter("\\z")
      .next();
  IO.println(data);
}
```

Ans: Prints the whole file. This is because the delimiter pattern of `\z` (lowercase) matches the end of input. See NIO below for a simpler way.

The **`Formatter`** class is the reverse of `Scanner`; it formats primitive types into strings of text. There is also a convenience method `String.format` for this:

```
String s = String.format(
  "name: %-15s, age: %03d, birthday: %D "
     + "Salary: $%,.2f%n",
  "Hymie", 29, new Date(), 33501.99 );
```

Files

To read or write some file you need to create a stream from it using `FileInputStream` or `FileOutputStream`. These take `File` objects in their constructors.

Remember that for text files, you need to wrap the byte-at-a-time stream with some `Reader` or `Writer` class, to convert to/from Unicode characters. A pair of convenience classes allow you to skip a step: **`FileReader`** and **`FileWriter`** can be used to read and write text files. (Show File demos.)

> In Java 24, the interface `java.io.Reader` gained a new `static` method, `Reader.of(CharSequence)` was added to allow efficient reading of characters from any `CharSequence` (such as a `String` or `StringBuilder`). The efficiency comes with the cost of no built-in synchronization, so don't share these between threads.

A **`File`** object represents either a file or a directory. It is assumed that files are organized into one or more ***filesystems***. Each filesystem has a top directory, called the ***root***. On Windows systems, each filesystem root is assigned a drive letter.

Files are named by providing a ***pathname***. An ***absolute*** pathname uniquely identifies a file on a computer (of course you may have short-cuts to that file with different names). A ***relative*** pathname names a file relative to some directory. This may be your default (or current working) directory from the DOS command line, or your *home* directory in Java. On older Windows systems your home directory is usually "`C:\WINDOWS`". You can determine your home directory with: **`String homeDir = System.getProperty( "user.home" );`**

On Unix/Linux an absolute pathname might be "`/tmp/file.txt`" while on DOS systems you might use "`C:\\temp\\file.txt`". **Note the double backslashes in DOS pathnames!** Here's some examples:

```
File f = new File( "data.txt" );
FileInputStream fis = new FileInputStream( f );

FileInputStream fis =
   new FileInputStream( "data.txt" );
```

```
// A DataStream has methods to read/write any
// Java primitive:
DataInputStream dis = new DataInputStream(
   new BufferedInputStream(
      new FileInputStream( "data.txt" ) ) );

// A zip file is really a compressed archive; each
// entry in the archive is one file:
import java.util.zip.*;
ZipInputStream zis = new ZipInputStream(
   new FileInputStream( "data.zip" ) );
ZipEntry entry;
while ( (entry = zis.getNextEntry() ) != null )
{  System.out.println( entry.getName() );
   zis.closeEntry();
}
zis.close();
```

Class `File` can be used with either files or directories, and allows you to create files (and directories), rename files (and directories), and delete files (and directories). This class can be quite tricky (non-intuitive) to use! **Some changes to files will invalidate the object, so you must close and reopen it.** It is a better idea to use the NIO.2 classes for these tasks.

Use **`File.separator`** to construct *pathnames*. (Else remember to double the backslash on Windows; however modern Windows (and Macs) allow the Unix standard forward slash to be used.) **(Show methods and `separator` field of class `File`.)**

```
File myNewDir = new File( "tempDir" );
if ( ! myNewDir.exists() )  myNewDir.mkdir();
System.out.println( "Path: "
   + myNewDir.getAbsolutePath() );
```

Note that nearly all I/O code throws checked exceptions, and thus must be within `try` blocks, not shown in these examples. Remember to use the newer try-with-resources.

A final example I stole many years ago:

```
// JavaWorld file renaming example:
import java.io.*;
public class FileRename
{ public static void main(String args[])
  { try
     { // Check if test file in current dir
       File f = new File("dummy2.txt");
       String name = f.getName();
       if (f.exists())
         System.out.println(f.getName() + " exists.");
       else
         System.out.println(f.getName() + " doesn't
exist.");
       // Attempt to rename with a different extension
       int dotIdx = name.indexOf('.');
       if (dotIdx >= 0)  name = name.substring(0,
dotIdx);
       name = name + ".tst";
       String path = f.getAbsolutePath();
       int lastSep = path.lastIndexOf(File.separator);
       if (lastSep > 0)  path =
path.substring(0,lastSep);
       System.out.println("path: " + path);
       File f3 = new File(path + File.separator +
name);
       System.out.println("new name: " + f3.getPath());
       if (f.renameTo(f3))
         System.out.println("Rename Successful.");
       else
         System.out.println("Rename failed.");
       // Delete the file
       // Fix 1: delete via the "Filename" not File
       if (f3.delete())
         System.out.println("Delete Successful.");
       else System.out.println("Delete Failed.");
     } catch (Throwable t)  { t.printStackTrace(); }
  }
}
```

Notice how ugly and complex this is! The `java.nio.file.Path` class is the modern (since Java 7) replacement for class `File`, and has a much

better API. Paths are discussed later (NIO.2), but briefly you can **create a `Path`** using: `Path p = Paths.get(pathname or URI);`

Most methods of `File` objects are replaced with static methods of the class **`java.nio.file.Files`** (*show some of these*). This design keeps class `Path` simpler.

A more complete example of working with CSV files is found in the resources.

Random Access Files

In addition to stream access, files can be accessed in non-sequential order as a **`RandomAccessFile`**. Such files make use of a ***current position marker*** that you can move about. (It works like a bookmark.) The next read or write operation will occur at the marker position. This allows you to skip to a random point in the file and begin reading/writing from that point, without having to read all the data that came before. Note that read and write operations also change the current position marker.

Random access files are most useful when each record has exactly the same number of bytes. (Then a simple calculation can determine the offset of any record.)

The marker is a **`long`** that indicate the byte ***offset*** from the beginning of the file that the next I/O operation will use. For this reason, the marker is usually called the ***file offset*** or ***file pointer***. To move the marker, use the **`seek`** method. To find out where the marker is set, use the **`getFilePointer`** method. To find the size of a file, use the **`length`** method. These methods work with bytes not characters!

To read from the beginning of a file: `in.seek(0L); in.read(b);`

To append to the end: `out.seek(in.length()); out.write(b);`

`RandomAccessFiles` are useful for storing fixed size records without using a database. If each record is, say, 250 bytes, then to read the 5th record you move the marker to position 5 * 250 and read the next 250 bytes.

When writing to a random access file, remember you are over-writing previous content, not inserting new content!

`RandomAccessFiles` can be constructed from `File` objects or from Strings (pathnames). They can be opened in `"r"` mode (read-only) or `"rw"` mode (read-write). It is an error to open a non-existent file for reading. If you open a non-existent file for read-write, the system will attempt to create it for you.

You can truncate or grow a random access file with **`setLength`** method. (If you grow it, the new contents are undefined by Java!)

File Selection Dialog: JFileChooser

In GUI swing programs you can **`JFileChooser`** to pop up a dialog box to allow the user in select a file. (AWT has a **`FileDialog`** class that is similar.) See the Java docs and also the on-line tutorial.

Typically, you display a file chooser in response to some user action, such as when a user clicks a button or a menu choice. The user must select a file or cancel the request before doing anything else. This most common type of interaction is known as a ***modal*** interaction, and it's the way the AWT `FileDialog` class works. `JFileChooser` also supports ***non-modal*** interactions. In this case, the chooser is always on display and an action listener processes selection events.

```
import java.io.File;
import jakarta.swing.*;
public class FileChooserDemo {
  public static void main ( String [] args ) {
    SwingUtilities.invokeLater( new Runnable()
    { public void run ()
      { JFileChooser fileChooser =
            new JFileChooser( "." );
        int status =
           fileChooser.showOpenDialog( null );
        if ( status == JFileChooser.APPROVE_OPTION ) {
          File selectedFile =
            fileChooser.getSelectedFile();
          System.out.println( "Selected: "
             + selectedFile.getParent()+ " --- "
             + selectedFile.getName() );
        }
        System.exit( 0 );
      }
    });
  }
}
```

The program displays a file chooser dialog. After the user selects a directory and file, the dialog identifies the directory and displays the filename.

`JFileChooser` uses ***single-selection mode*** by default, however you can customize this to allow ***multiple-selection mode***. It can also be customized to allow the user to select files only or directories only, and you can provide a ***file filter*** to limit what files are shown in the dialog (only "`*.txt`" files, or only "`myConfig.ini`", etc.). You can control if file icons should show, the fonts to use, etc.

Another way to customize `JFileChooser` is to add an ***accessory panel***. This is an area of the chooser where you can place a component to coordinate information related to the current selection. For instance, if the current selection is an image type, the component can be an image previewer. For audio files it could be a sound player, etc. Or you can just display a `JLabel` with file information (properties). With each change in the file selection, different information is connected to the accessory panel.

A new method,

jakarta.swing.filechooser.FileSystemView.getSystemIcon(File, int, int)

is available in Java 17 that enables access to higher quality icons when possible. It is fully implemented for the Windows platform; however, results on other platforms might vary and will be enhanced later.

`java.nio` Package

As of Java 1.4, a completely new set of I/O classes was added in **`java.nio`** with many new features and performance improvements, mostly to take advantage of new OS non-blocking I/O features that are now common.

> The API for working with files is still too complex. Java7 includes many additions to NIO (called NIO2), to make working with files and directories much simpler.
>
> Sadly, programmers must maintain old code, so the old ways of working with I/O are discussed (in detail, since I wrote these notes when the old ways were the only ways). For newer code, or if you have a chance to refactor old code, I recommend using the NIO.2 classes.

With the new I/O (or `nio`) package, the abstraction is not a stream of bytes. Instead, the metaphor used is input or output of a *block* (or *buffer*) of data. You input or output a whole array of bytes at a time. This is much more efficient than invoking the operating system for each byte, as with streams. In fact many of the older `java.io` classes have been re-implemented using `java.nio` classes. So you may see a performance improvement even when using the older classes.

Each operation produces or consumes a whole block of data in one step. Processing data by the block can be much faster than processing it by the (streamed) byte. (For one thing, the underlying OS works this way.) But block-oriented I/O lacks some of the elegance and simplicity of stream-oriented I/O.

Channels and Buffers

`Channels` and `Buffers` are the central objects in NIO, and are used for just about every I/O operation.

`Channel`s are analogous to streams in the original I/O package. All data that goes anywhere (or comes from anywhere) must pass through a `Channel` object. **A `Channel` object represents a data sink or source**: a file, a stream of I/O (from the keyboard or to the console window), a network connection, etc. You create a `Channel` object associated with some file or network connection, just as you used to create `InputStream`s or `OutputStream`s.

> These ***channel*** objects support fancy buffering, file locking operations, non-blocking I/O, character encoders/decoders, the ability to view a file as an array (*memory-mapped files*), and other new stuff.

`Channel` objects are two-directional; you can read or write to one. (`java.io` streams are either input or output.) The stream classes in `java.io` now have a method `getChannel()`, so you can create an output or input stream as normal and then get the `Channel` object that represents it:

```
FileInputStream fin = new FileInputStream("file.txt");
FileChannel fc = fin.getChannel();
```

A `Buffer` is essentially a container object. All data that is sent to or read from a channel must first be placed in a buffer. **A `Buffer` object passes data between a `Channel` and your program:**

- **To output data**, you put data into a `Buffer` object, and write the `Buffer` to some `Channel`: `myBuf.put(`*`data`*`); Channel.write(myBuf)`
- **To read data**, you read into a `Buffer` from a `Channel`, and then access the data within: `Channel.read(myBuf); myBuf.get(`*`data`*`)`

The most commonly used kind of buffer is the `ByteBuffer`. A `ByteBuffer` allows get/set operations (that is, the getting and setting of bytes) on its underlying byte array. But a `ByteBuffer` is not the only type of buffer in NIO. There is a buffer type for each of the primitive Java types: `ByteBuffer`, `CharBuffer`, `ShortBuffer`, `IntBuffer`, `LongBuffer`, `FloatBuffer`, and `DoubleBuffer`.

Each of the buffer classes implements the `Buffer` interface. With the exception of `ByteBuffer` they all have the exact same operations, differing only in the type of data they deal with. Because `ByteBuffer` is used for most standard I/O operations, it has some additional methods that are unique.

> Actually, these are just *view* Buffers: convenience wrappers around `ByteBuffer`s. As with streams, ultimately I/O is done with bytes. But it can be convenient to have a `Buffer` that seems to work with `int`s or `char`s instead of `byte`s. Such *view Buffers* provide a nicer interface that

wraps a `ByteBuffer`. There are utility methods for creating view Buffers in `java.nio.channels.Channels` class. These can be used to create view Buffers for any primitive Java type.

You can also create specialty *view buffers* from a `ByteBuffer`, using the "`as...`" methods. For example to make a read-only `Buffer`, use: `buf = buf.asReadOnlyBuffer()`. You can even create a "view" of a part of Buffer using `slice` buffer. Writing to the resulting slice `Buffer` will modify a part of the underlying `Buffer`.

You create a `ByteBuffer` by reserving some memory for it: `ByteBuffer.allocate(`*`numberOfBytes`*`)`. Once created you can use it's methods to put in bytes or primitive data (using convenience conversion methods, just as with class `Scanner`) or to pull out data.

Another way to create a `Buffer` object is to use the `ByteBuffer.wrap()` method to create a `ByteBuffer` that uses some `byte` array you've previously created:

```
ByteBuffer buf = ByteBuffer.wrap("data".getBytes());
```

***Direct versus non-direct* Buffers:** Some operating systems provide a fast I/O system known as *direct I/O*. Java calls such `Buffer`s "direct-mapped". These can be slower to allocate and deallocate but are faster for reading and writing. Also the `byte` array is not allocated in the program's *heap* and copied into the OS's buffer each time; the OS's buffer is used directly.

Since not all OSes support this type of I/O, an attempt to allocate a direct `Buffer` may result in a regular, non-direct buffer. Use the `isDirect` method to determine this, if you care.

To read data: `fc.read( buf ); int i = buf.getInt();`

Notice that you don't need to tell the channel how much data to read from or write to the buffer. Each buffer has an internal way to keep track of how much data has been read and how much room there is for more data.

One feature different from stream I/O is that **the `read` method doesn't block—if no data is currently available the method returns zero. When the end of the input has been reached, the `read` returns -1.** (A stream will block, just sit waiting forever for more data.) So you usually use a loop like this to read data:

```
int r = fc.read( buf );
while ( r != -1 ) {
  do something with the data just read into buf;
  r = rc.read( buf );
}
```

A shorter idiom is often used:
```
while ( rc.read(buf) != -1 ) {do something;}
```

To write data: `buf.putInt(17); fc.write(buf);`

(There are many `get*` and `put*` methods in a `ByteBuffer` you can use.) To use NIO, you must understand a bit more about buffers and their methods. Let's dive in!

Buffer Internals

Besides an array of bytes, A `Buffer` has a number of properties:

A `Buffer` has a fixed ***capacity*** which is the maximum number of bytes that the `Buffer` can hold.

A `Buffer` has a ***limit*** which marks the end of the data in the `Buffer` (it is the index of the first element in the array that should not be read/written). So if a 1,024 byte capacity buffer contains 300 bytes of data, the limit is set to 300 (and the usable data has indexes from 0-299).

A `Buffer`'s ***position*** is the index of the next element to be read/written. So when (say) 300 bytes of data is first read into some buffer, the position is set to 0 and the limit to 300. If you fill a buffer with (say) 300 bytes and want to append more data, it will be written at the position (300).

Many `Buffer` types also support a ***mark***, which is a bookmark. Once set, you can later `reset()` the *position* to the mark.

You can examine or set the *position*, *limit*, and *mark* with various `Buffer` methods. (*Demo this*.)

Now that you know about the limit and position, there are two methods you must understand in order to use Buffers: `clear` and `flip`.

The `clear` method prepares a `Buffer` for a channel `read` operation. Without the `clear`, a `read` (remember this reads from the channel into the buffer) or `put` operation attempts to add to the end of the previously read data in the buffer (i.e., add at *position*, up to the *limit*). So if you previously read a whole `Buffer`'s worth of data, the position==limit==end of the buffer, and no bytes can be read in! Use `clear` before using `Channel.read` or any `ByteBuffer.put` method calls. **The `clear` method sets the *limit* to the *capacity* and the *position* to zero.**

The `flip` method prepares a `Buffer` for a channel `write` operation. Once you've filled a `Buffer` with data you you want to writer out, you might try to use `Channel.write` or one of the `Buffer.get` methods. But after the last read (into the buffer), the *position* is set to the *limit* (i.e., the end of the data in the

buffer) so no data would be written! Use the `Buffer.flip` method before writing data from a `Buffer`. **The `flip` method sets *limit* to the current *position* (i.e., the end of the available data), then sets the *position* to zero.**

You can copy a file with this code:

```
FileInputStream fin = new FileInputStream("src");
FileChannel fcin = fin.getChannel();

FileOutputStream fout = new FileOutputStream("dest");
FileChannel fcout = fout.getChannel();

ByteBuffer buf = ByteBuffer.allocate( 1024 );

buf.clear();

while ( fcin.read(buf) != -1 ) {
   buf.flip();
   fcout.write( buf );
   buf.clear();
}
fin.close(); fout.close();
```

Non-blocking I/O for Servers — Network I/O [*Skip to NIO.2*]

Students only need to know these features exist, for now at least. I have added some details and some whole topics, because I find them interesting and also to provide students with a starting point if someday they need to know about this stuff.

As for non-blocking I/O, as of Java 21 you can use *virtual threads* for about the same performance as asynchronous I/O but without the complexity!

The non-blocking (or *asynchronous*) I/O operations provided by NIO allows higher performance in data processing and offers better scalability; however you don't see much advantage until you have to handle requests from hundreds or thousands of users at once.

With standard blocking I/O, the standard client-server model creates one thread per request. When waiting for I/O (network problems, time-outs, keep-alives, ...) the thread is blocked. With too many clients you get too many threads to be handled efficiently; the server can become slow, drop requests, or crash.

One of the advantages of asynchronous I/O is that it lets you do I/O from a great many inputs and outputs at the same time. Synchronous programs often have to resort to polling, or to the creation of many, many threads, to deal with lots of connections.

With non-blocking I/O you don't need to create one thread per request. Instead a thread will handle multiple requests, making I/O calls without waiting for the reply before handling the next request. So a few dozen threads can handle many thousands of requests. But writing such code is harder! (Note: Currently (2008) Tomcat uses blocking I/O but Glassfish doesn't.)

Normally, when your code makes a `read()` call, the code blocks until there is data to be read. Likewise, a `write()` call will block until the data can be written. Asynchronous I/O calls, on the other hand, do not block. Instead, you register your interest in a particular I/O event: the arrival of readable data, a new socket connection, and so on. The system tells you when such an event occurs. With asynchronous I/O, you can listen for I/O events on an arbitrary number of channels, without polling and without extra threads.

Due to time limitations we won't discuss this in detail. Briefly then, a `Selector` object is where your code registers its interest in various I/O events, and it is the object that tells you when those events occur. You pass a `Selector` object to various `Channel register` method calls. These calls tell the channel to send an event via the `Selector` to registered listeners when the events of interest (that you registered for) occur. When that happens, you typically use a loop to iterate over the `Set` of I/O events that have just occurred, to process each one.

> A number of frameworks are available to make this easier, such as Mina from Apache and Grizzly from Sun.

Memory-Mapped I/O

Modern (general purpose) operating systems use *virtual memory*. This allows you to *map* a file into memory, treating the file as an array of bytes (an NIO `MappedByteBuffer`), which in many cases is much faster (and more convenient) than traditional I/O. With this you can read/write huge files; the system will *swap* in the parts of the file as needed. Memory-mapped I/O is quite a common programming technique, but it isn't suitable for all cases.

You use a **`MappedByteBuffer`** like any other NIO Buffer, using various `get` and `put` methods to read from or write to the buffer. A `MappedByteBuffer` doesn't use heap memory! Rather, it uses *shared memory* provided by your OS; this makes it easy to share the file between multiple processes. No actual memory is allocated up front for this, making it fast to create. Rather, when you attempt to access a range of bytes not currently in memory, the OS will fetch that block (a *page*) at that time using what is known as a *major page fault*. (You can pre-load the file's contents into memory up front, but that can be slow.) Due to its design, memory-mapped I/O can handle files larger than your available memory.

So Memory-mapped I/O avoid the much slower *open, read,* and *write* system calls and only fetches data from the disk when it is needed (via a page fault). However, memory-mapped I/O has issues similar to multi-threading. The file can be modified by other processes while you are attempting to read from it or write to it, causing corruption or data loss. Attempting to access parts of the file that were deleted (truncated) will cause data unavailable errors. Frequent page faults can affect performance and not just for the current process. Finally, as you write to the buffer, the data is not flushed back to the underlying file automatically. but the flush operation (the `force` method) can take a long time and frequent use can also hurt performance. Yet, not flushing data frequently enough means other processes may be using stale (old) data. (Note most of these are non-issues if your program is the only one using the file.)

How to Use:

The **`FileChannel.map`** method returns a type of direct `ByteBuffer` known as a **`MappedByteBuffer`**. Your program can directly access that `Buffer`'s byte array to read data or write data. When done, use the `force()` method to flush any changes to the in-memory array back to the file. (There's also a `load()` method to pre-load the array, to try to speed up subsequent read operations.) (S*how demo.*)

Here is some timing data (Cay Horstmann's *core Java 2 v1*, 7th Ed., page 689) for reading a large file and computing a CRC32 checksum on it:

Random Access	162 seconds
Plain Input Stream	110 seconds
Buffered Input Stream	9.9 seconds
Memory Mapped	7.2 seconds

Scatter/gather I/O

Scatter/gather I/O is a method of reading and writing that uses multiple buffers rather than a single buffer, to hold data. A scattering read is like a regular channel read, except that it reads data into an array of buffers rather than a single buffer. Likewise, a gathering write writes data from an array of buffers rather than a single buffer. Scatter/gather I/O is useful for dividing a data stream into separate sections, which can help implement complicated data formats.

> Scatter/gather I/O is useful for dividing a piece of data into sections. For example you might be writing a networking application that uses message objects, with each message is divided into a fixed-length header and a fixed-length body (e.g., ATM packets). You create one buffer that's just big enough for the header and another buffer that's just big enough for the body. When you put these two in an array and read into them using a scattering read, the header and body will be neatly divided between two buffers.

Likewise, a gathering write is useful for forming a single data stream from a group of separate buffers. Each buffer can contain one field of some record, and you then write the whole record at once. (Unlike with a scattering read this will work even when the fields are not fixed length.)

File Locking

With NIO you can create locks for a whole file or part of one. Locks can be *exclusive* (useful for writing) or *shared* (useful to prevent others from writing while you are reading, but permits others to read as well). Locks can also be *mandatory* or *advisory*, depending if they are or are not enforced by the JVM (or the OS). Locks can be on entire files or just a range of bytes within the file (called *record* or *row locking*).

Different OSes support different types of locks. Some only mandatory, some only advisory, and some depending on the file in question! Some OSes only support *exclusive* locks. Some do not support record locks. Thus you might try to create a shared mandatory lock and get handed an exclusive advisory one! Also, some OSes don't support file locks on memory-mapped files.

For maximum portability you should only attempt to use advisory, exclusive, whole file locks.

Sharing files between multiple processes is just not a great idea. There are other ways for processes to communicate with each other; search the Internet for *interprocess communications* (or IPC).

Your code needs to *acquire* a lock manually, then access the data, then *release* the lock. Here's some sample file locking code:

```
RandomAccessFile raf =
   new RandomAccessFile( "usefilelocks.txt", "rw" );
FileChannel fc = raf.getChannel();
FileLock lock = fc.lock();   // Exclusive lock for whole file
...;   // Do I/O here
lock.release();
```

Other threads that try to lock the file this way will block until the lock is released by whatever thread has it, or it will return an error. (You can lock just a part of a file by passing additional arguments to `lock`.)

NIO.2 — Convenience Classes for NIO and "old" IO

One of the new packages for NIO2 (the "new" NIO, added in Java 7) is `java.nio.file`, which contains many useful classes and interfaces. (There isn't actually any package named `nio2`.) The interface `Path` allows one to easily create `File` or Stream objects from *pathnames*. You can create a **`Path`**

object to refer to a file by its *pathname*. A `Path` can be manipulated in various ways: you can alter it, extract pieces of it, or compare it to another `Path`. Much of the standard library that takes `File` arguments has been updated to take `Path` arguments too. (If not, you can convert from one to the other.)

Once a `Path` object has the correct pathname you desire, you can check the existence of the file corresponding to the `Path`, create the file, open it, delete it, change its permissions, and so on, using the static methods of class `java.nio.file.Files`. To create a `Path` object, you use code like this:

```
import java.io.*;
import java.nio.file.*;
...
// Create an object to represent the local file
system:
FileSystem localFS = FileSystems.getDefault();
// Create a Path from a filename:
Path myFile = localFS.getPath( "Data.txt" );
// To read from file:
Scanner fileIn = new Scanner( myFile, "UTF-8" );
// To write to the file:
PrintWriter fileOut = new PrintWriter(
  myFile.toFile(), "UTF-8" );
```

You can also walk through filesystems easily using `java.nio.file.Files`. Class `Files` has many useful methods, far more than the older `File` class, for working with files (*show in java docs*). See openjdk.java.net/projects/nio/presentations/TS-5686.pdf (slide-show) for more info on NIO.2, also this tutorial from IBM DeveloperWorks. and the Oracle File I/O tutorial for Java 7.

Here are some examples of using `java.nio.file.Path`:

```
Path source = Path.get("foo");
Path target = Path.get("/otherdir/bar");
source.createFile(); // create if doesn't exist
// copy source to target:
source.copyTo (target, REPLACE_EXISTING);
// rename or move atomically:
source.moveTo (target, ATOMIC_MOVE);
source.createLinkTo (target); // link: source -> target
URI uri = source.toURI(); // URI form of path

Path qpath = Path.get ("/path/to/queuedir");
DirectoryStream someDir =
```

```
 qpath.newDirectoryStream("*.qmsg");
try {
   // DirectoryEntry is a FileRef
   for (DirectoryEntry entry: someDir) {
      processQueuedFile (entry);
      entry.delete();
   }
} finally {
   someDir.close();
}

// change file permissions and attributes:
FileRef file = ...
PosixFileAttributeView view =

file.newFileAttributeView(PosixFileAttributeView.class)
;
if (view == null) {
   throw ... // file doesn't support POSIX view
}
PosixFileAttributes attrs = view.readAttributes();
System.out.format("%s %s %s %d\n",
   PosixFilePermission.toString
(attrs.getPermissions()),
   attrs.getOwner().getName(),
attrs.getGroup().getName(),
   attrs.getSize());
// set read-only by owner
try {
   view.updatePermissions
(PosixFilePermission.OWNER_READ);
} catch (Exception e) { ... }
```

Show file demos.

Serialization

Not all files are text! Java supports the input and output of entire objects using `ObjectInputStream`s and `ObjectOutputStream`s. This uses a process known as ***serialization*** and enables Java to save objects to a file (usually with the "`.ser`" extension") or a database; this is known as *object persistence*. Serialization also supports distributed applications, where objects are accessed from different computers and so must be sent across a network. This last use is very important in RMI and enterprise applications. Object serialization uses the `Serializable` and `Externalizable` interfaces.

Only classes implementing the **`java.io.Serializable`** interface can be serialized and deserialized automatically, serializing all parent objects as well. It automatically traverses references between objects, saving and restoring entire graphs.

Marking a class with this interface causes the compiler to "inject" (my word) a pair of methods, `writeObject` and `readObject`, which you can override if you remember they are there.

> There are two additional methods injected that you can override: `writeReplace` and `readResolve`. These are used to write a different object or return a reference to a different object than the one the system is trying to write or read. The `readResolve` method is important and is discussed in more detail below.

`Serializable` objects are restored by reading them from an `ObjectInputStream`. To read an `int`, a `String`, and a `Date` from a stream (written previously using `ObjectOutputStream`):

```
ObjectInputStream ois =
  new ObjectInputStream(
    new FileInputStream("stuff.dat") );
int i = ois.readInt();
String today = (String) ois.readObject();
Date date = (Date) ois.readObject();
ois.close();
```

(Writing to an `ObjectOutputStream` is similar.) Reading an object does not run a constructor for that object. Here's what happens: Memory is allocated for the object and initialized to zero. No-arg constructors are invoked for any non-serializable (parent) classes. Then the fields of the serializable classes are restored from the stream starting with the serializable class closest to class `Object` and finishing with the object's most specific class. Note that no existing objects are used, so you may end up with duplicate objects if not careful!

Since Java 14, you can annotate fields (and methods) in a serializable class with the @Serial annotation. Use of this makes your intent clear and allows the compiler to issue useful warnings.

Class Versioning

The serialization runtime associates with each serializable class a *version number*, called a **`serialVersionUID`**. This is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization. If the receiver has loaded a class for the object that has a different `serialVersionUID` than that of the

corresponding sender's class, then deserialization will result in an `InvalidClassException`. **A serializable class can declare its own `serialVersionUID` explicitly by declaring a field named `"serialVersionUID"` that must be `static`, `final`, and of type `long`:**

```
private static final long serialVersionUID = 42L;
```

If a serializable class does not explicitly declare a `serialVersionUID`, then the serialization **runtime will calculate a default `serialVersionUID` value for that class** based on various aspects of the class, as described in the Java(TM) Object Serialization Specification. However, it is strongly recommended that all serializable classes explicitly declare `serialVersionUID` values, since the default `serialVersionUID` computation is highly sensitive to class details that may vary depending on compiler implementations, and can thus result in unexpected `InvalidClassExceptions` during deserialization. It is also strongly advised that explicit **`serialVersionUID` declarations use the `private` modifier** where possible, since such declarations apply only to the immediately declaring class — `serialVersionUID` fields are not useful as inherited members.

Serialization poses a problem at the remote (receiving) end: the object copy is "reconstituted" but not loaded by the `classLoader`. This can pose a problem in some situations as you may end up with duplicate objects, where you only wanted a single object (the *singleton pattern*).

Marking a class as serializable will do the correct thing most of the time. However, some situations don't work well (such as when the superclass is not serializable). In some situations, you can mark the objects as `Externalizable`, which allows the objects to be serialized and deserialized, but doesn't do it for you. You have total control over how to do these.

Discuss the **`transient`** modifier for fields that should not be serialized: fields that aren't part of the logical state (i.e., that can be re-created from the other data), cached data, or security info that can't be sent across a network for security reasons.) `Static` fields aren't serialized either, so using such fields in distributed programs is of limited value (instead, use singleton pattern, or store such data in a central place such as a database).

`readResolve`

Constructors often check for valid constructor arguments. What happens if you create an object using `java.io.ObjectInputStream.readObject`? None of your constructors are invoked!

Java provides a pseudo-constructor you can define for this situation (serialized object construction) called **`readResolve()`**. This method should be treated as a

constructor and not forgotten (e.g., when making constructors `private`, make this `private/final` too) when designing your classes. (It should nearly always be `private`, even if constructors are `public`.)

For `Serializable` classes the `readResolve` method allows a class to replace/resolve the object read from a stream. The method is defined as follows:

ACCESS-MODIFIER Object readResolve() throws ObjectStreamException;

The `readResolve` method is called when `ObjectInputStream` has read an object from the stream and is preparing to return it to the caller. `ObjectInputStream` checks whether the class of the object defines the `readResolve` method. If the method is defined, the `readResolve` method is called (*after* the object is constructed!) to allow the object in the stream to designate the object to be returned. You can return a different object than the one just constructed. The object returned should be of a type that is compatible with all uses. If it is not compatible, a `ClassCastException` will be thrown when the type mismatch is discovered.

Use `readResolve` to ensure you only have a single instance of some serializable singleton. For example, suppose you have the following singleton class:

```
public class Microsoft
{  public static final MONOPOLY = new Microsoft();
   private Microsoft () { ... }
}
```

This is the standard pre-Java5 way to create a singleton; only one instance of `Microsoft` is ever created. Next suppose you serialize this class:

```
public class Microsoft implements Serializable
{ ... }
```

This change destroys the singleton property! Each time the `Microsoft` object is de-serialized a new instance is created. In fact, up through Java 1.1 it was impossible to create a serializable singleton! `readResolve` was added in Java2. You can use it this way to restore the singleton property of class `Microsoft`:

```
public class Microsoft implements Serializable
{  public static final MONOPOLY = new Microsoft();
   private Microsoft () { ... }
   private Object readResolve()
                   throws ObjectStreamException
   {  return MONOPOLY;
   }
}
```

Another use of `readResolve` is to invoke your constructors rather than the default constructor used. You again write `readResolve` to return a different object than the de-serialized one:

```
public class Foo
{  Date start, end;
   public Foo (Date start, Date end )
   { ... // check for start < end }
   private Object readResolve()
                  throws ObjectStreamException
   {  return new Foo( start, end );
   }
}
```

Here we only use the de-serialized object to obtain start and end.

Serialization can be a problem for collections that use generics. Java uses a technique known as *type erasure* that removes generic type information from objects. Since you can't add a `readResolve` method to collection objects you must cast the resulting collection of Objects to the real type:

```
@SuppressWarnings( "unchecked" )
ArrayList<Foo> foos =
  (ArrayList<Foo>) ois.readObject();
```

(A more efficient approach would be to use Serialized arrays, which are more compact and do contain correct type information. You can convert your Lists to arrays before sending/storing, and back again when reading.)

`javadoc` provides three tags to document the serialized bits: **`@serial`**, **`@serialField`**, and **`@serialData`** (for `writeObject` or `writeExternal` methods).

`Externalizable` Interface

A related, lighter-weight interface is **`java.io.Externalizable`**. Classes that use this instead of `Serializable` must provide the methods to serialize/deserialize the object's data (including any inherited fields from parent classes); nothing happens automatically. When an `Externalizable` object is reconstructed, an instance is created using the public no-arg constructor and then your `readExternal` method called. Writing an object is similar. Any class versioning is your responsibility also.

Both interfaces can be used and mixed in read and writing a stream of objects. Each object to be stored is tested for the `Externalizable` interface. If the object supports `Externalizable`, the `writeExternal` method is called. If

the object does not support `Externalizable` and does implement `Serializable`, the object is saved using `ObjectOutputStream`.

> ***Summary:*** To serialize objects, the class must implement the `Serializable` interface (as must all parent classes). Serializable classes must have a version number. `javac` will add one automatially but issue a warning. Just add your own:
>
> ```
> private static final long serialVersionUID = 42L;
> ```

Learning advanced I/O never seems like much fun. But it is important for enterprise applications that Java is often used for, and is likely the sort of code Java programmers will encounter.

For many students, database technology is more fun. That is covered next.

Lecture 7 — Database Technology and Access from Java

A *database* is just a collection of related data. (The SQL standard calls databases *catalogs*, but the two terms are the same in practice.) A database ***schema*** is the design or plan of the database. A database can be as simple as a plain text file. But they can be complex as well. All databases need to support four basic operations: create, read, update, and delete (CRUD).

Two types, or models, of databases stand out as the most common and important:

Object Model – Uses **JNDI** (***Java Naming and Directory Interface***) to retrieve objects (or other things) by name. A *directory* is a type of hierarchical object database, and an organization may have several. JNDI permits you to search multiple directories. This sort of database is useful with Java, since it would permit you to store objects and later retrieve them, across a cluster of application servers. Such directories also serve a lookup function, allowing one to (for example) find a printer or specific database, knowing its name.

Some popular object databases in use today include MongoDB, Redis, Cassandra, and Amazon's S3 and Dynamo databases.

Relational Model – Uses *tables*, where each row is one record and each column is one attribute/property/field. Multiple tables can be linked on a common column (a ***join***) to produce a composite table. This is the most commonly used model, as it supports features needed for common business functions. Many companies may have invested many millions of dollars in proprietary relational databases, and you may need to use one from your Java program. Relational databases are designed from mathematical theory (*relations*) that offer guarantees about the behavior of such databases.

A set of tables and the constraints between them form a relational ***database***. The description of the tables (names, column names, types, indexes, and constraints) is called the database ***schema***. The term schema can also mean a group of table definitions that form a namespace. This provides a way to name a subset of tables and other objects within a database. To refer to a table *t* in a (different) schema *s,* use the notation *s.t*. (This is rarely needed for simple databases, which will use a default schema (namespace).

> Defining a proper schema for a given application takes some training. Small changes to the schema can result in such large performance changes as to render an entire application unusable.

A database can be something as simple as a single file with a standard format. However, all databases have many issues that must be addressed in an enterprise application. A DMS (Data management system), also called a DBMS or **RDBMS** (***Relational Database Management System***) provides support for creating and

accessing to databases, automatically handling such issues as concurrent access, caching, storage onto disk, transaction support, network access, security, etc. Examples include SQL Server, Oracle, DB2, MySQL (and MariaDB), SQLite, and PostgreSQL (commonly called Postgres). In fact, in 2025, Postgres is likely the most popular open-source relational database; with plugins, it also handles non-relational data!

All RDBMSs use a standard language known as **SQL** (***structured query language***) to perform CRUD operations known as DML (*data manipulation language*), operations on the schema (like creating the DB, specifying constraints, etc.) known as DDL (*data definition language*), and operations that control the DB server such as creating users, access control, etc., known as DCL (*data control language*).

SQL supports many useful features besides tables (but not every RDBMS supports all of these). ***Views*** are virtual tables formed by a query. ***Stored procedures*** are sets of SQL statements that can be stored in the server. Once this has been done clients can refer to the stored procedure, passing in parameters. (This is efficient as well as more secure.) A ***trigger*** defines a set of actions that are executed when a database event occurs (delete, insert, or update) on a specified table. A ***sequence*** is a counter used to generate unique ID; each time you use it you get the next number. An ***identity*** column is a *primary key* column that can use a *sequence* to generate the values automatically. A ***tablespace*** consists of one or more datafiles, used to specify a physical location of a database object.

Relational database are very common as they support the important business functions of OLTP and OLAP. Most businesses will have multiple databases in use.

[Taken from IBM's database articles at publib.boulder.ibm.com, no longer available]

Transaction-processing systems (OLTP) are designed to capture information and to be updated quickly. They are constantly changing and are often online 24 hours a day. Examples of transaction-processing systems include order entry systems, scanner-based point-of-sale registers, automatic teller machines, and airline reservation applications. These systems provide operational support to a business and are used to run a business.

Decision-support systems (OLAP, although no one knows what the *on-line* part is supposed to mean here) are designed to allow analysts to extract information quickly and easily. The data being analyzed is often historical: daily, weekly, and yearly results. Examples of decision-support systems include applications for analysis of sales revenue, marketing information, insurance claims, and catalog sales. A decision-support database within a

single business can include data from beginning to end: from receipt of raw material at the manufacturing site, entering orders, tracking invoices, and monitoring database inventory to final consumer purchase. These systems are used to manage a business. They provide the information needed for business analysis and planning.

Data for decision-support systems is often captured by online transaction-processing systems, extracted and transformed to a form suitable for analysis, and then loaded into a decision-support system (i.e., a separate DB). This process is called ETL (*extract, transform, load*). The resulting database is called a ***data warehouse***, giving us the term *data warehousing*.

I have a detailed set of relational database notes, *Database Overview for System Administrators* at wpollock.com. However, there are many excellent (that is, better) resources for learning SQL or specific DBMSs on the Internet. To learn about the internals of different systems and what a developer should know about them, I strongly recommend Designing Data-Intensive Applications, 2nd Edition by Martin Kleppmann and Chris Riccomini, due out in 12/2025. (I loved the first edition!)

NoSQL Databases

Generally, all non-relational databases are called *NoSQL*. But in fact there are many different types of non-relational databases available, with different use cases. For example, Cassandra is technically a *document database* and not an object DB. Some types of non-relational databases include graph databases (often used to manage social relationships such as in Facebook), spacial databases (for GIS (geographical) data), time-series databases (for metric data), and so on. As mentioned above, object-based (key-value) and relational are most commonly used.

Often other software sits atop a no-SQL DB to do queries and data analysis, such as Apache Spark or Apache Drill.

(See DB-Engines.com for current DBMS popularity and other news.)

JNDI (directory) and other non-relational databases are beyond the scope of this course. (Hey, I had to leave out some topics; these so-called lecture notes are already hundreds of pages long!)

Files or Database for Persistent Storage?

For read-only access to the data, a file (text, XML, or another format) can work well. But the reason most people use a database even for simple data is that any (web) application may be used by several users at the same time. If one user is reading a file while another writes to it, you will have problems of

one sort or another. Plain files also do not provide good access control, do not provide reports and audit logs, and cannot enforce data consistency rules.

In a cluster or grid of web servers, a ***load balancer*** will send requests to different web servers. So either you would need to keep a copy of your data file on each server and synchronize any updates, or your application will need network access to the file. Using a single back-end database solves this problem but can be a performance bottleneck; you end up with a cluster of replicated/sharded database servers.

Failover can be an issue for direct file access; what happens when an application crashes while updating the file? Efficiency can be a factor too — some filesystems are slow or can even lock a file while being accessed.

All in all, you are often better off using a database management system (DBMS). A DBMS is designed to efficiently handle all these issues so you don't have to.

Some web / presentation layer frameworks will handle the JDBC for you. Also, JDK 6 thru JDK 8 came bundled with *Derby*, an Apache RDBMS. Using a DBMS isn't much harder than using a file and can have many advantages. Of course, you must learn how to use that part of Java.

For simpler data persistence needs (such as the size and position of a window, or a user's font or theme choice), use the Preferences API, discussed later.

Embedded Database or Client / Server

A database may be ***embedded*** in an application. Such a DB is easy to set up; it is just a set of classes (that access a native library). The resulting DB is available to the one application only. A typical use would be a desktop or smart phone application. The most popular one is SQLite.

The data in an embedded DB can't be shared easily with other applications. An embedded DB cannot support concurrent writes and does not work well if the application and the database are separated by a network. For these (more common) situations, you need a single central DB that can be accessed by many applications/users. This is where you use a DBMS, or *DB server*. Applications must establish a *connection* to the server (usually across a network) and authenticate themselves (e.g., username and password). Also a single server may serve several different DBs, so the client application must also identify which DB it wants to connect with. Connecting to a server is thus slower than using an embedded DB.

The most common commercial DBMS is Oracle, which includes extra tools for creating complete DB-centric applications. However, Java SE used to include Apache Derby, an excellent DBMS that can be run either as a server or embedded.

> Derby was originally the commercial DBMS *Cloudscape*. It was written by a Californian startup, bought by Informix, acquired by IBM when it bought Informix, and renamed as *Derby* when IBM donated it to the Apache organization. When bundled in the JDK, it was also called *JavaDB*.

Even if eventually using an embedded DB, it may pay to use a client-server DB during development. Doing so will allow you to examine the DB during testing; with some embedded databases, that is hard or impossible. (With some, you can use Squirrel SQL (or DBeaver) to view them; see links in class resources.) If using Derby, you only need to change the JDBC driver used (and the connection code) to switch from client-server to embedded — the SQL stays the same.

> Derby, H2 (which is popular with Java developers), and HSQLDB are good for small to medium databases (up to a few tens of millions of rows each for dozens of tables). SQLite is good for small databases but only embedded ones; it lacks the features of the others.
>
> MySQL (now owned by Oracle) its successor, MariaDB, and PostgreSQL are good for Gigabyte sized databases that require fast connections such as for websites. (PostgreSQL is likely overall better but MariaDB is optimized for fast reads.)
>
> Go with Oracle or DB2 for Terabyte sized DBs. (Some DBs such as Google's main database are measured in Petabytes!)
>
> Note, even a small DB needs to be well designed (including proper indexes) or it will suffer serious performance issues.
>
> Derby, H2, and some others support **in-memory databases**, very useful when learning and during development! An in-memory DB is especially useful for testing.

Replication and Distribution (Sharding)

Databases are often replicated for load-balancing purposes or for high availability. (Sometimes replicas are stored around the world to improve performance by lowering latency.) However, distributing all reads and writes to all the replicas can cause problems, since it isn't possible to have a distributed system that is simultaneously always available, always consistent, and tolerant of failures (network or host). This principle is known as the CAP Theorem.

One way to deal with this is to have one server for writing and all other replicas are read-only. All SQL queries that modify data must go to the main server, while the more common lookup only queries can go anywhere. This design requires the application to use two database connections: one for read-write and one for read-only queries.

Often there is a lot of data in a database, and any given query only needs to look at a small part of the data. In such cases, the database can be split between multiple servers. There are many techniques for this. One is to group by tables, so queries that (for example) deal only with customer data are on one server, while inventory is on another.

Another common technique (especially with read-only data) is known as ***sharding***. Each shard is a subset of the rows of a table. For example, one can calculate the MD5 hash of the primary key and divide it by the number of shards. For two shards, all even hashes go to one shard and the odd hashes go to the other. Another possibility is to shard by some other table column, so (for example) all customers with the same zip code go into the same shard. To perform a lookup when sharded data must be combined, the query is sent to all shards which perform their lookups in parallel, and return the results to the client which must merge the results.

Each shard is stored in a separate but identical database, on separate servers. As with replication, the application code must be aware of sharding.

Another consideration is locally caching query results for a period of time. Doing so can greatly enhance performance, but you must be careful about using stale data.

For these reasons, enterprise applications usually have a separate *tier* (set of classes) that hide the details of persistent data behind a (hopefully) simpler interface. If your organization changes DB vendors or types, only the code in that tier is affected. (For simple web-based applications such as personal blogs, using sharding, geographically diverse replication, and caching would be overkill and a separate tier is not normally used.)

Transaction Processing

When updating a DB that is used by multiple clients simultaneously, the changes need to appear *atomic*. This is done by creating a *transaction*, which is a group of statements. A transaction is a unit of work that has "**ACID**" properties: ***Atomic*** (the group is done or undone completely), preserves ***Consistency*** of the data, ***Isolation*** (the transaction appears atomic to other transactions), and ***Durable*** (persistent). Transactions are generally supported only by RDBMS.

> Strict isolation is hard to implement well and expensive (takes a long time and other resources) to use. Most RDBMS provide a way to control the isolation level used, allowing one to trade strict isolation with better efficiency.
>
> The SQL standard defines four levels of transaction isolation. **The most strict is *Serializable***, which says that any concurrent execution of a set of Serializable transactions is guaranteed to produce the same effect as running them one at a time in any order. All four were standardized in the 1990s and

assumed that locks would be used to implement this. More recently, multi-version concurrency control (MVCC) supports lock-free concurrency, using *snapshot isolation*. Postgres supports a form of this, serializable snapshot isolation (SSI) which uses MVCC to improve performance, at the cost of more transactions failing to complete the first time.

Serializable isolation prevents the following potential issues from occurring: *dirty reads* (reading data modified by the transaction before the transaction is committed), *non-repeatable reads*, or *read skew* (a transaction often reads data more than once; if the data is different each time, that's a problem), *phantom reads* (similar to non-repeatable reads), and *write skew* (concurrent transactions each determine what they are writing based on reading some data, which overlaps what another concurrent transaction is writing).

Repeatable Read isolation keeps read and write locks (acquired on selected data) until the end of the transaction. This isolation does not prevent phantom reads.

Read Committed isolation keeps write locks (acquired on selected data) until the end of the transaction, but read locks are released as soon as the `SELECT` operation is performed. This may allow non-repeatable and phantom reads.

Read Uncommitted isolation is the lowest isolation level, allowing all the issues described above.

To make a transaction, start with "`START TRANSACTION;`". Now all the SQL statements you enter next will have no effect as far as other clients are concerned! After the last statement, you use the command "**`COMMIT`**;" which will make the changes visible to all. If you decide not to commit your changes, use "**`ROLLBACK`**;" instead of `COMMIT`. The transaction will also rollback automatically if there was an error that prevents the commit from completing.

Many DBMS servers can treat each SQL update/insert/delete statement as a transaction automatically. With MariaDB for example, just "**`set AutoCommit=1`**" and each statement executed is a single transaction. (Setting to zero instead turns off that feature.) With Derby, use "`AUTOCOMMIT ON|OFF;`". (This is not needed for embedded DBs since only one SQL statement can be running at once (from a single process/thread). But when multiple clients can execute SQL on the same data at the same time, unless you use transactions data can easily become corrupted.)

(DCL statements were the last part of SQL to be standardized, and even now (2025) many DBMSes uses propriety DCL statements. This is why changing vendors is often difficult.)

A DBMS records the commands for all changes in each transaction to a log file (the *journal* or *write-ahead log*). Then it makes the changes to the tables. Finally, the DBMS marks that log entry as complete. If the system crashes at any point, it can simply replay the journal entries to restore the DB, completing the transaction in progress. This works because such log entries are *idempotent*. That means running such commands more than once won't corrupt anything.

Most DBMSes also use a form of COW (copy on write), meaning the DB tables look unchanged to other users until the transaction is complete.

JDBC and SQL — A Brief Overview

An application's data may be updated, deleted, or added to the tables of a RDBMS using SQL. Note that while SQL is standardized, each different DBMS omits some SQL and/or provides proprietary SQL extensions that are useful enough to tempt one into using them. Even the standard parts don't always work the same way!

Typically, the application makes ***queries*** on the database and returns a **`ResultSet`**, a collection of the rows that matched the query. An example query might be "list all customer names, addresses, and phone numbers, who live in Tampa and who have made a purchase over $500 within the past year". The other typical operation is to add a new record to the database (which may mean adding rows to several tables in a single transaction). An example is adding a purchase order.

Most SQL statements are called *queries* or *updates*, and can be entered on one line or several lines. They end with a semicolon ("`;`"), although not all DBMSes will require this. The SQL keywords are not case sensitive; only data inside of quotes is case sensitive. SQL uses single quotes around literal text values (some systems also accept double quotes). Some examples of SQL DDL:

```
CREATE TABLE table_name (
  col1_name TYPE CONSTRAINTS,
  col2_name TYPE CONSTRAINTS,
  other_constraints
);
```

```
CREATE TABLE person (
  id INT NOT NULL,
  lastName VARCHAR(255),
  firstName VARCHAR(255),
  PRIMARY KEY (ID)
);
```

```
DROP TABLE table;
```
(There's also ALTER TABLE)

Some examples of SQL DML:

```
INSERT INTO table (col1, col2, ...)
  VALUES (val1, val2, ...);

    SELECT [DISTINCT] col1, col2, ...
          or use wildcard:  * instead of a column list
      FROM table [, table2, ...]
      WHERE condition (e.g. WHERE amount < 100)
      ORDER BY col;

    UPDATE table
      SET col2 = value2, col3 = value3, ...
      WHERE condition; (e.g. WHERE col1 = value1)

DELETE FROM table
  WHERE condition;     (e.g., WHERE col1=value1)
```

> The *value* used in UPDATE or WHERE clauses typically comes from a user, often a web form. **Remember to normalize, sanitize, and validate** such data before using it in an SQL statement! *Prepared statements* are safer.

Designing a good DB is a job for an expert. A bad design may be so slow as to be unusable, cause security vulnerabilities, or may even induce errors in the data! This holds true if either using SQL or the Java query API (JPQL).

> ***Normal forms*** are a way to prevent normal DML operations from destroying real data or creating false data. That can happen if the schema isn't designed for the types of queries and multi-user activity that is common. Creating normal forms is a straight-forward process of transforming a schema from normal form *n* to normal form *n+1*. Although there are many normal forms (at least 9, or over 300, depending on how you count them), normal forms 4 and higher cover obscure potential problems that very rarely ever manifest. Most DB schema designers are happy with third normal form.
>
> Not all databases should be normalized! Those used for OLAP rarely are and can have tables with hundreds of columns, often containing redundant information.
>
> But for OLTP and most other relation databases, normalizing the schemas makes good sense: databases get updated, and if not in normal form, that update logic must be in the application code, at every point where the database is updated. Errors are more likely in this case.

As an illustration on why DB design is hard, consider this example:

A common use for a DB is to allow customers to sign up for services from your website. Suppose you want to send a "Click here to confirm your registration" URL in an email to a new subscriber. This is commonly called an "invite".

Each user needs a unique ID to refer to their data in the DB, and it is "obvious" to use a sequence number. But then you have problems with exposing those numbers in URLs, or having attackers guess the next transaction number.

The correct solution is to use a sequence number, encrypted and the result encoded to base-64. When the user then clicks the link, your web server decodes the link, then decrypts it to recover the serial number. The user sees a large random-looking link only. Internally you can use simple sequence numbers in your DB.

But wait, there's more! To prevent attackers using random strings (that will decode into random IDs), you need to encrypt the sequence number plus a checksum of that number. Now most random URLs won't decode into valid sequence number-checksum pairs.

The DB table where you store these should include a timestamp as well (to time out invitations), a "responded" Boolean field (to prevent replay attacks), and the IP where the invitation was sent (to help prevent spoofing). Of course, the table includes other details of the invite, such as the username, etc.

When the reply is received you can validate the checksum, check the time against the timestamp, check if this has been replied to before, and check the source IP.

All this and we haven't designed the table for performance or privacy compliance yet! Clearly, DB design is a job for an expert.

JDBC Driver Types

Sending commands (queries, inserts, ...) to some DBMS requires sending commands in some way from the client or application code to the database server (that is, using a protocol). The commands and protocols are different for each different DBMS. (That is, to talk to an Oracle DBMS you would need to use the proprietary Oracle DBMS protocol.) These protocols are not part of SQL and are not standardized. Some of these protocols are ASCII text sent over TCP/IP. Others are optimized binary protocols. Most DBMSes support multiple protocols.

Usually, using the vendor's propriety protocol directly is the most effective. This requires the vendor provides a Java class for that. Few applications do that (it ties the DBMS vendor into your application logic), but that does provide the best performance and allows your application access to any proprietary features of your DBMS.

Instead, the common solution (not just for Java) is to use a *driver* that translates some standard DBMS protocol into the specific protocol needed for a particular DBMS. Of course, this approach has drawbacks: not all standard operations are

available on all DBMSes, and some DBMSes have extra features not part of a standard. Most languages support at least one standard (even if only standard in that one language).

In Java, this standard language is **JDBC (Java DataBase Connectivity)**. This is the lowest-level of access to a DB in Java. Later you will see how to use a higher level of abstraction to make your code easier to read and write. Your applications can use JDBC to connect to a DBMS, and if you change the DBMS, the code doesn't care.

To make this work, the JDBC part of the JRE must have a JDBC driver that can translate the standard JDBC access into the proprietary protocol actually used. There are several ways that can be done, each with different tradeoffs.

> ***ODBC*** is a standard connection protocol for talking to different DBMSes. (From Wikipedia: ODBC uses as its basis specifications from the SQL Access Group, X/Open (now part of The Open Group), and the ISO/IEC. Microsoft (in partnership with Simba Technologies) created ODBC by adapting the SQL Access Group CLI. It released ODBC 1.0 in September 1992. After ODBC 2.0, Microsoft decided to align ODBC 3.0 with the CLI specification making its way through X/Open and ISO. In 1995, SQL/CLI became part of the international SQL standard.
>
> Oracle dropped ODBC support from Java rather suddenly; see below.

There are ~~four~~ three types of JDBC drivers available:

1 **~~JDBC-ODBC Bridge~~** ~~- Since nearly all DBMSes provide ODBC drivers, you can use an JDBC-ODBC bridge to access them. This isn't really a type of driver, it is a "ODBC" driver that was standard with Java. It is commonly referred to as a "*type 1* driver" anyway. This solution is limited in features and performance (your queries go though JDBC driver, then the ODBC, driver, then the proprietary driver, and finally to the DBMS), but may be your only option if no JDBC drivers are provided by your DBMS vendor. This type of driver uses the native OS's ODBC driver, and thus can't be used in an Applet.~~

> ~~Note! There are two ODBC systems on Windows for 64-bit systems, ODBC32 and ODBC64. The type of ODBC must match your database, not your operating system. If you use 32-bit Java on a 64-bit Windows system, you must use ODBC32 drivers or your program won't work. Unfortunately, the default on 64-bit Windows (Under Administrative Tools) is ODBC64 (`%windir%\System32\odbcad32.exe`).~~ **~~To use ODBC32~~**~~, you must launch the 32-bit utility, not found in the Start menu: `%windir%\SysWOW64\odbcad32.exe`. Note the utility file name is the same in either case.~~

~~**As of Java 8, Oracle removed the JDBC-ODBC bridge from the JDK**, so you can't compile or run code anymore that uses this. (They didn't even mark it deprecated, they just removed it.) Since it was always included in the past, there are few if any alternatives; that may change over time. Meanwhile, use the other types of drivers.~~

2 **Native API** – This is the *type 2* driver that translates JDBC API into some propriety DBMS API. This solution requires the vendor's DBMS client be running locally, so this driver can talk to it. The DBMS client in turn talks to the DBMS server across a network, using a proprietary protocol. This solution is only a little better than a type 1 driver, as it provides slightly better performance. But this solution has deployment, manageability, and scalability issues.

3 **Net Protocol** – This is the *type 3* driver designed for use in Jakarta EE client-server applications. The client code uses this driver type to send requests to an application server (middle-tier server), which in turn talks to the actual DBMS using some other type of driver.

This solution is server-based, so there is no need for any vendor database library to be present on client machines. It provides opportunities to optimize portability, performance, and scalability. The net protocol can be designed to make the client JDBC driver very small and fast to load. Additionally, a type 3 driver typically provides support for features such as caching (connections, query results, and so on), load balancing, and advanced system administration such as logging and auditing.

On the other hand, Type 3 drivers require database-specific coding to be done in the middle tier, generally not a good idea. Also accessing a `ResultSet` may take longer since the data comes through the backend server.

4 **Native-Protocol** – This all-Java, *type 4* driver converts JDBC calls into the vendor-specific database management system (DBMS) protocol, so that client applications can communicate directly with the database server across a network. Level 4 drivers are provided by the vendor and completely implemented in Java to achieve platform independence and eliminate deployment administration issues. This provides the best performance, but can lead to deployment issues, since if you change your DBMS you must then update all clients with new drivers. **This type of driver is the most commonly used.**

Java DB drivers are loaded in a strange way. The driver class isn't directly accessed by your code so it won't normally be loaded by the class loader. To force the driver to load, you used to have to use some ugly code:

```
try {Class.forName( "name.of.jdbc.driver" );}
  catch ( classNotFoundException e ) {...}
```

Or the simpler: `Class c = name.of.jdbc.driver.class;`

Or even: `new name.of.jdbc.driver();`

In Java 6 and newer, this is not be needed (but should still work). **JDBC type 4 drivers are loaded automatically when you create a `Connection`. The JRE determines the driver class name from the connection URL and loads the correct driver for you.** You still use this older method to use non-type 4 drivers however.

As part of its initialization, the `DriverManager` class will attempt to load the driver classes referenced in the "`jdbc.drivers`" system property. This allows a user to customize the JDBC drivers used by their applications. For example, in a file (in your HOME directory) named `.hotjava/properties` file you might specify:

```
jdbc.drivers=foo.Driver:bar.sql.Driver:baz.Driver
```

The `DriverManager` will try to load as many drivers as it can find, and then for any given connection request it will ask each driver in turn to try to connect to the target URL.

Once this is done, you can create an SQL statement to be sent over a connection to the database. In typical Java fashion, you create a `Connection` object from the DB driver, then a `Statement` object from the `Connection`. Finally, you execute the statement, which (hopefully!) returns a `ResultSet` of the results of your query statement. Here's an outline of the code needed (using a type 4 driver):

```
Connection con = null; Statement stmt = null;
String user = getUserName(...); //Returns null on err
String pass = getUserName(...); // ditto
String url = getURL(...);  // ditto again
// Checks for credential errors go here...
try {
   con = DriverManager.getConnection(url, user, pass );
   stmt = con.createStatement();
} catch ( Exception e ) {
  System.err.println( "**** Cannot open connection to "
     + URL + "!" );  }
try {
 ResultSet results = stmt.executeQuery( "SQL_Query" );
 ...  }
```

The `ResultSet` contains *metadata* you can access: the number of rows returned, and the number, name, and type of each column. You access each row using a loop that causes the `ResultSet` object to internally "point" to the next row. (The pointer is called a *cursor*.) You can only advance the pointer; you can't jump around in the results. Also, note that some SQL statements don't return `ResultSets`, just a status.

> **Putting the DB connection username and password in the code is a bad idea!** Remember it can be read with any hex editor and other tools. It is also bad to store a password in an unencrypted and/or unprotected configuration file. Passing credentials to your program is also no good; often that password shows in the command line arguments of the process, which usually can be viewed by anyone with access to the (hardware) server. Using environment variables is not secure either.
>
> The correct technique is to use an encrypted password, and have your Java code decrypt it to use. See StackOverflow for details. Other methods for handling secrets is discussed below, in Passwords (and Other Secrets).

With JDBC, your Java program must access each column of each row, convert it to an appropriate Java data type, and (usually) build an object representing that row. The objects must be manually put in a collection.

Jakarta EE supports a more object-oriented approach, using query objects that return collections of objects directly. This will be discussed later.

Using Derby

The Apache Derby DB is easy to install and use, and used to be included with the JDK as "Java DB". (Which is why I have extensive notes about it.) You can get it and view the Derby documentation from db.apache.org/derby. Derby supports in-memory DBs as well as either embedded or server modes. (Installing and using any other RDBMSes is similar but the details vary quite a bit.)

Derby was well maintained until 2023. H2 until 2024. Both are still useful for development as most popular RDBMS don't support embedded or in-memory use, and setup is very simple. (Note, you can simulate Postgres or other DBMSes in-memory, by running a container on a RAMdisk. This is actually a good method for testing.)

To use Derby, make sure the environment variable **`JAVA_HOME`** is set to where you installed the JDK, and **`DERBY_HOME`** to where you installed Derby. Make sure that **`PATH`** includes `%JAVA_HOME%\bin;%DERBY_HOME%\bin`". (In `PATH`, don't use environment variables but only full, absolute pathnames.) Once set, open a fresh command line window, and test the settings by running "`ij NUL`". You should see the version number of Derby show.

Make sure **`CLASSPATH`** contains "`%DERBY_HOME%\lib\derbytools.jar`" and one or both of "`%DERBY_HOME%\lib\derby.jar`" (if using the embedded DB) and "`%DERBY_HOME%\lib\derbyclient.jar`" (if using the server). Or just add "`%DERBY_HOME%\lib\*`", which is what I do to use `javac` from the command line directly. If using Maven or another build system, you don't need to change the global `CLASSPATH`; Maven will add dependency jar files automatically.

All set! Now you can create and use databases. The easiest way is to use the embedded or in-memory database rather than setup the server. You should `cd` to the directory where you want to create the DB, then run the command line tool "**`ij`**".

> Using the embedded DB is easier, but if you want you can use the server.
> **To start the Derby server**, run: `start/b startNetworkServer.bat`
>
> **To shutdown the serve**r, run: `stopNetworkServer.bat`. (You can always hit ^C in the server's DOS window if you didn't use "`start/b`".)

Now create a DB named `AddressBook`, using the embedded driver:

```
C:\Temp> mkdir myfirstdb
C:\Temp> cd myfirstdb
C:\Temp\myfirstdb> ij
ij> CONNECT 'jdbc:derby:AddressBook;create=true';
```

> A Derby DB can be used from a zip or Jar file. (In this case, the DB is *read-only*.) To access a DB in a zip/Jar, use this connection URL: "`jdbc:derby:jar:(`*`pathToArchive`*`)`*`dbPathWithinArchive`*".

The next time you want to use this DB you can (but don't have to) omit the "`;create=true`" part.

> The ***system directory*** is where Derby keeps its databases (in a subdirectory for each). Derby uses the Java property **`derby.system.home`** to determine which directory is its system directory, and thus what databases are in its system, where to create new databases, and what configuration parameters to use. You need to define that on the command line when starting the Derby server, or from within your Java code. If you specify a system directory at startup that does not exist, Derby creates this new directory automatically (of course it won't contain any DBs yet). You can't specify this directory on the `ij` command line, but there are other ways.

If you do not specify the system directory when starting up Derby, the current directory becomes the system directory. To set this property from within your Java code (on Windows), use:

```
Properties p = System.getProperties();
p.setProperty("derby.system.home", "C:\\mydb");
```

(It's probably better to get the path name by using `File` or NIO2's `Path`.)

A Derby database is multiple files in a directory (or zip file) named for the database (`AddressBook` in the example above). Some DBMSes such as SQLite keep the database in a single file.

Now you can type in SQL statements to create tables (DDL). (E.g., "`CREATE TABLE addresses (name VARCHAR(30), phone VARCHAR(20), notes VARCHAR(255));`"). Now add some data to your DB.

If the SQL statements you used to create and initially populate your DB are saved in a text file, you can run that code to recreate a database easily with "`ij`" with the command "`run 'someFile.sql';`". (Be sure to delete the old DB first, by either first deleting the directory, or by using the SQL statement "`DROP DATABASE databasename;`" at the beginning of that file.)

To see a list of tables, use "`show tables;`". To see the description of some table, use "`describe tablename;`". Use DML to add, modify or delete rows and perform queries to that database.

When done with your `ij` session, use "`exit;`".

If you examine the directory (the folder), you will see Derby created a new sub-directory for the database (and named for it). You will also see a log file "`derby.log`" (in Unix text format) that is useful to troubleshoot connection problems. You can zip the whole `AddressBook` folder to backup your DB. Inside you will see a directory "`log`" that contains the DB transaction log files, a directory "`seg0`" that contains multiple files containing the actual DB data. You will also see a Java properties file (DOS text) named "`services.properties`". **Never edit any of the files in your DB directly!**

Derby doesn't support the SQL standard `INFORMATION_SCHEMA.*` views, so you can't use the following SQL standard method to list the tables in your DB:

```
SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_SCHEMA = 'AddressBook';
```

However this will work for Derby:

```
SELECT t.tablename FROM SYS.SYSTABLES AS t
WHERE t.schemaid = (SELECT s.schemaid FROM
  SYS.SYSSCHEMAS AS s WHERE s.schemaname = 'APP');
```

You can use "C:\>**`dblook -d 'jdbc:derby:AddressBook'`**" to see or save the DDL statements for your DB. (Derby includes `ij`, `dblook`, `sysinfo`, and other command line tools in the `bin` folder.)

You can set system-wide Derby DB properties in a text file called **`derby.properties`**, which must be placed in the directory specified by the `derby.system.home` Java property. Database-wide properties are stored within the database itself. You set and verify database-wide properties using system procedures within SQL statements. To set a property, use the `CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY` procedure, passing a property name and the value. To view the current value of a property, use the `VALUES SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY` function, passing in the name of the property.

To use a Derby embedded DB from your Java program, use code such as this:

```
import java.sql.*;
public class Foo {
  public static void main ( String [] args ) {
    String dbName="jdbcDemoDB";
    String connectionURL = "jdbc:derby:" + dbName;
       // + ";create=true";  // If creating the DB
    try {
      Connection conn =
         DriverManager.getConnection(connectionURL);
      Statement s = conn.createStatement();
      s.executeUpdate( SQL-string ); // insert or other DDL
      ResultSet rs = s.executeQuery( SQL-string );
      ResultSetMetaData meta = rs.getMetaData();
      ...
    } catch ( Exception e ) { ... )
```

> Unlike most interactive DB browser tools (including "`ij`"), **the SQL statements used with JDBC must <u>not</u> end with a semicolon!** Forgetting this is a common source of errors.

The `Connection` can be used to obtain a `DatabaseMetaData` object, which has methods you can use to query the DB in general (for example, get a list of the tables in that DB). See the `DerbyDemo.java` demo in the class resources.

The `ResultSet` is a table-like object, with methods to traverse the rows and columns (among others). But using a database `ResultSet` directly is tedious and error-prone! Modern Java has a new interface that automatically handles the conversion of objects to/from table rows, and can even generate the required SQL from simple method calls! This will be discussed next.

To use a DBMS server (such as Derby, MySQL, Oracle, Postgres, DB2, ...), you only need to make a few changes to the connection (and start the server before running your program):

```
String driver = "org.apache.derby.jdbc. ClientDriver";
String connectionURL = "jdbc:derby://localhost:1527/"
   + dbName;
```

There are many other methods and features included with JDBC, not discussed here. Also, keep in mind you can use JDBC with any relational database management system, as long as there is a driver available.

To create an in-memory Derby database, specify `memory` as the JDBC sub-sub-protocol. For example, to create an in-memory database named `myDB` using the embedded driver, use the following connection URL:

```
jdbc:derby:memory:myDB;create=true
```

For the network client driver, use the following connection URL. Because the client driver does not understand the memory subsubprotocol, you must include it in the database name:

```
jdbc:derby://myhost:1527/memory:myDB;create=true
```

Be careful to specify a colon (:) after `memory`. Database names are actually relative *pathnames* from the Derby *system directory*.

Finally, keep in mind an in-memory DB may require more memory then normal, and you may need to adjust the JVM's heap size.

ORM and the DAO Pattern: The Java Persistence API (JPA)

Since most programming code is designed to work with objects or collections of objects, to use a traditional DB requires ***object to relational DB mapping*** (ORM). For example:

```
class Person { String name; int age; ... }
List<Person> people = new ArrayList<Person>(); ...
Person p = new Person();
p.setName( "Hymie" );
people.add( p );
```

The problem with this code is that the `Person` object (the `people` list) isn't persisted anywhere. To store/retrieve people from a relational DB used to require SQL:

```
Create table person (...)
Select * from person where id=1234
...
```

Thus, you must manage people (`Person` objects) as a table of raw data, or convert each fetched (stored) `Person` object to/from the DB, converting objects to/from rows and changing Java types to SQL types. An additional problem is that the Java programmer needs to understand SQL, which isn't part of Java.

In production-quality code, the SQL queries are sometimes kept in text files and accessed by your Java program. (These files must be private for security reasons.) That way, a DB expert can write the SQL needed and it doesn't need to be embedded in your code. You can even change the SQL without recompiling. This is in fact a common technique.

One solution is to use an *object-based* database, but today most software still uses relational DBs. Using a non-relational DBMS is not often an option.

Wouldn't it be nice if any changes to `Person p` such as shown above automatically persisted the change to a relational database? This is the magic that JPA provides: you specify the persistence details either with annotations when you define class `Person`, or in a separate configuration (XML) file named `META_INF/persistence.xml`.

This type of software automatically converts the objects to/from the database `ResultSets` for you. This is **ORM (*object-relational mapping*)** software, and is common in Jakarta EE and other (e.g., Ruby on RAILS) solutions. ORM is also called the ***data access object*** (DAO) pattern. ORM provides an abstract interface to some type of database or persistence mechanism, which provides some specific CRUD operations without exposing details of the database. (That means that you can easily switch database details, even types, without changing or recompiling your Java code!)

Commonly used with Jakarta EE, the DAO pattern separates object persistence and data access logic from any particular persistence mechanism or API. Naturally all the work must still be done: to query the back-end DB (usually with SQL, but increasingly, other query languages), convert the results to Java data types, and store changed objects. But this can be done automatically!

The ORM API in both Java SE and Jakarta EE is called **JPA**, the ***Java Persistence API***. (The term DAO is rarely used anymore.)

> In fact, JPA includes its own query language, ***Java Persistence Query Language* (JPQL)**, and the results are automatically translated into the required SQL. However best performance *may* be obtained by an SQL expert *if* they use raw SQL instead. Note that the actual query strings should be stored in external property or text files, to make it simple to change the queries later, and such files must be kept private for security reasons.

The design of your data using objects is different than when using relational database tables. However, the fundamentals of data modeling are the same, known as ***entity-relationship* (E-R) *modeling***. The first question to ask while modeling data is: what ***entities*** are needed, and how do entities ***relate*** to other entities? The result is known as a *domain model*.

For example, if a retail order system model states that an "Order" has multiple "Line Items" for which reproducible order matters, you likely would have an object model that includes:

```
public class Order {
@Id
private final int ID;
```

```
@OneToMany(mappedBy = "orderId")
List<LineItem> lineItems;
...
}
```

If you further wish to guarantee that no `LineItem` appears more than once in a given order, you'd likely use a `Set` instead of a `List`. (The old JPA spec doesn't allow a `SortedSet`, but it seems that the "`@OrderBy`" annotation works around that limitation.)

> The term used to describe database objects is ***entity***. (Sometimes they are called *domain objects*.) In Java, entites are defined by regular Java classes, with extra annotations. Each entity class generally corresponds to a database table, and objects of that class correspond to rows of that table. However, an entity may be composed of data that came from several tables. (This is similar to an SQL virtual table, or view.)

With very little additional code, JPA can create the appropriate RDBMS tables and appropriate SQL queries from this class definition, when you do lookups or other CRUD operations. Someplace, the URLs, usernames, passwords, and database names must be made available to the JRE. But using JPA makes it easy to fetch a record as an object from a database, modify it, and update the database record.

With JPA, the entities are simple ***POJO***s (*plain old Java objects*). Once created, you add the object to a ***persistence unit*** ("PU"), which is just Java's name for a database. (Remember, the actual DB details for each PU are stored in a separate config file.) To use some PU, you use methods of `jakarta.persistence.entityManager`. Normally, you don't have to specify too many details as JPA uses sensible defaults for nearly everything. You only need to use a few required annotations in most cases, and use a minimal `persistence.xml` file. JPA replaces JDBC with much simpler, shorter code!

> JPA also includes a cache of recently read or written objects. Understanding that is important, since your data isn't "flushed" to the actual persistent DB immediately by default. Having such a cache improves performance, but just as with multi-threading, it can be a source of problems if you simply ignore it. Various annotations can configure caching.)

Once entities objects are added to some PU, the entities are ***managed***. That means any changes made will automatically be persisted (at some point, when the `entityManager` is "flushed"). Here's some examples:

```
Person person = new Person();
person.setID( 1000 );
person = entityManager.merge( person );
```

```
person.setName( "Hymie" );
Person p2 = entityManager.find(Person.class, 1234);
```

The change to `person` is magically persisted! (`em.merge` works much like `em.persist`.) Anyone forced to use JDBC in the past really appreciates how wonderfully simple JPA can be! And while the simple example above doesn't show it, JPA is very powerful if you need that functionality.

The **`EntityManager`** object is your program's contact point with JPA. You use its methods to persist entities, flush the cache, create transactions, and do everything else. Obtaining a reference to the JRE's entity manager is different when using Jakarta EE and Java SE; with EE, it is automatically *injected* into your code! You just declare a reference to it. When the Jakarta EE JRE (the "container") constructs an object of your class, it will populate that variable for you. With Java SE, you need to use a static method to obtain the reference.

(When learning Jakarta (and Java) EE, I always assumed the books I read had typos in all the code listings: they just declared an uninitialized variable with the proper annotation, then used it as if it were initialized. Guess I should have used better books!)

While JPA is built-into Jakarta EE, it is only supported by Java SE; **the standard JRE doesn't include an implementation of the `jakarta.persistence.*` classes!** That means your code will compile, but if you try to run it you'll see an error message similar to this:

```
C: \JPADemo>java com.wpollock.jpademo.Main
Exception in thread "main"
jakarta.persistence.PersistenceException: No
Persistence provider for EntityManager named MyDB
  at jakarta.persistence.Persistence.createEntityMa
nagerFactory(Persistence.java:85)
  at jakarta.persistence.Persistence.createEntityMa
nagerFactory(Persistence.java:54)
  at com.wpollock.jpademo.Main.main(Main.java:24)
```

With Java SE, you will need to include a third-party implementation of JPA known as a *persistence provider*. Some you can use include the popular free/open source Hibernate (which predates the JPA, and was the inspiration for it), OpenJPA (a minimally compliant provider, but can be fast for that reason), and EclipseLink (the one I use for class demos; they claim to be the reference implementation and come with sensible defaults). If not using Maven or Gradle, simply download the one zip file, extract the folder `eclipselink` someplace, and make sure the two Jar files **`eclipselink.jar`** and **`jakarta.persistence-api.jar`** are

found in the CLASSPATH. (Look in `eclipselink\jlib` and `eclipselink\jlib\jpa` for these two Jars.) (See the Java Classroom Setup resource for details.)

> As of 2017, Hibernate has about 72% of the marketshare, Eclipselink has 13%, and OpenJPA has 2%, and the others have less. (I cannot find any more recent data on marketshare.) Personally I like EclipseLink best; Hibernate has better support (a larger community) and more features. None of these are perfect, perhaps because the JPA standard is still imprecise or incomplete in some areas.

Note that most of these JPA providers include additional features beyond what JPA requires. Using these extra features can greatly ease development and/or improve performance. But (just as with SQL) using non-standard extensions limits portability — you won't be able to easily switch JPA providers if you use extra features only provided by the first one. These features usually require extra configuration files to use; for EclipseLink, you can configure the extra features in a file named `eclipselink-orm.xml`, or in the standard `persistence.xml` file, or use non-standard annotations in your code.

> Until 2019, the JPA API was part of Java and the package named "`javax.persistence`". When Oracle gave control of JPA to the Eclipse Foundation in 2019, there was a legal requirement to change the name from "`javax`". The new package name is "`jakarta.persistence`". If you have old code (such as an older tutorial found on the Internet) you will need to change the name in your Java files and your `persistence.xml` file.

Note that one drawback of using JPA is that you must learn the JPA persistence commands, annotations, config file syntax, and the query language. If you already know SQL, using JDBC and SQL may be easier than learning all that new stuff. Also, if your organization already has SQL expertise available, you can't use that if they don't also know JPA too.

JPA is better though; if you're a Java developer who doesn't already know SQL very well, you are well advised to learn and use JPA and JPQL.

> Persisting complex classes with fields referring to collections can be done in SQL or JPA. However, getting all the JPA annotations exactly correct can be tricky. An alternative is to convert collections to JSON strings and persist the strings. (For example, a `Person` has-a collection of friends.) This solution doesn't scale well; it's better to learn JPA.

Show and review JPAdemo.java.

(Note, *lombok* can generate all the entity required methods (get/set for all fields, no-arg constructor, toString, equals, and hashCode) for you automatically by adding a one annotation! Lombok is not officially supported however. Since Java 17, you can use `records` instead of plain classes.)

> Ultimately your data is still stored in a relational database. A JPA programmer cannot ignore that and must still understand RDBMS concepts. You must include proper annotations to identify primary keys, constraints, multiplicity of relationships, columns to be used in joins, etc., or an incorrect schema will be generated. For performance reasons, you may also need to provide SQL hints and use *stored procedures* for security, using sharding, etc. You will also need to understand JPA caching. The learning effort to master JPA is about the same as for mastering SQL (including sharding, replication, caching, etc.)

Preferences API

There are times when you want to persist configuration information for an application. In the past you had to use `java.util.Properties` to save such data as the preferred font, theme, or language, the last window size and location, and other application preferences (often seen by the user as *options* or *customizations*), as well as configuration data such as database connection details, including usernames and passwords. (I don't recommend using properties or the properties API for sensitive information, as discussed elsewhere.)\

Using the `Properties` class (and the associated files) requires some effort spent managing implementation details and permission to read/write the file(s) on the client host. But that is better than the alternatives of managing and parsing files directly or working with complex databases.

The Preferences API (`java.util.prefs`) introduced in Java 1.4 provides a convenient framework for storing and recalling data about user-specific preferences or system-wide preferences. **It manages preferences for each *user*, plus a *system-wide* set of preferences, in a hierarchical collection of *nodes*.** In essence, the JRE includes a small NoSQL database of the key-value type. It is similar to the Windows registry but local to your JRE.

Each node has a *pathname* that identifies it. The package and class names are useful to identify nodes, but you can name them anything you like. Each node contains a collection of name (key) – value pairs (in other words, a Map). Typical information stored in system preferences might include installation configuration data for an application, window locations (in general, the state of the application), or user preferences. The JRE takes care of all the implementation details for the programmer and requires no extra permissions.

Warning! Don't use Preferences to store plaintext passwords or other sensitive data! Such **data must be securely encrypted first**.

Here's an example showing a `JFrame` that remembers its window location and restores it when next shown: (*From* java.sun.com/developer/JDCTechTips/2003/tt0715.html#1)

```
   import java.util.prefs.Preferences;
   import java.awt.Dimension;
   import javax.swing.*;  /javax is correct!
   public class MainWindow extends JFrame {
      private Preferences prefs;
      public MainWindow() {
        prefs =
Preferences.userNodeForPackage(MainWindow.class);
        setToPreferredSize();
      }
      public void setToPreferredSize() {
        int width = prefs.getInt("width", 100);
        int height = prefs.getInt("height", 200);
        setSize(width, height);
      }
      public Dimension getDimensions() {
        int width = prefs.getInt("width", 100);
        int height = prefs.getInt("height", 200);
        return new Dimension(width, height);
      }
      public void putDimensions( Dimension dimension) {
        prefs.putInt("width",
(int)dimension.getWidth());
        prefs.putInt("height",
(int)dimension.getHeight());
   }  }
```

Note that with the get* methods, you must specify a default value to be returned in case the key wasn't set previously, or if the DB is unavailable. Some JVMs will create a key with that default value if the key isn't already defined, but that is optional for the implementation and you cannot count on it.

The node's pathname is limited to 80 characters, as are the *keys*. The *values* are limited to 8K (8192) characters each. (Total space avail. depends on implementation.) See the **`java.util.prefs.Preferences`** Java docs for more information.

For Windows systems, the system and user preferences are stored in the registry, in `HKEY_LOCAL_MACHINE\SOFTWARE\JavaSoft\Prefs` and `HKEY_CURRENT_USER\Software\JavaSoft\Prefs` respectively. On Linux and Unix, the user preferences are kept in the hidden folder `~/.java/.userPrefs/`, and the system-wide ones in `/etc/.java/`.

> Since you can't tell how much space is available, you can't count on this persistent data store. Make sure your application will work fine even if the JVM is full or loses data (say, because of a JRE upgrade). On Windows, running registry cleaner utilities may delete saved preferences.

UUIDs

Universally Unique IDentifiers have many uses: For database transaction IDs, for disk (storage volume) IDs, for User IDs, and indeed, for any Internet-accessible resource. Here we are concerned only with database uses. UUID are partly or totally random numbers. Using UUIDs, you don't have to worry about name collisions when multiple nodes (in our case, database replicas) simultaneously generate IDs; the chance of duplicate IDs is low enough to safely ignore. (Note this is only an issue if you allow multiple replicas to write to the DB.)

To help developers use such IDs, there are actually standards for UUIDs. Most languages and databases support these standard UUIDs, making your code portable and simpler. The official standard is RFC 9762 (actually still just a draft).

Confusingly, the different types of UUIDs are known by *version numbers* rather than type numbers or names. Different versions are actually just different UUID types, optimized for different use cases. (Some older UUID standard versions have been replace with newer ones, but don't automatically use the version with the highest number! Currently (2024) there are ten versions defined (two with names, not version numbers).

Use v4 when you just want a random ID. This is a good default choice as it is very secure.

Use the newer v7 (if supported on your system) if you're using the ID in a context where you want to be able to sort them; they contain (start with) a Unix timestamp, so newer UUIDs will always have a bigger number than older ones. For example, consider using v7 if you are using UUIDs as database keys.

> Note! The creation date is part of the UUIDv7, so if that is sensitive information in your application, don't use this. Look at the other versions shown in the standard RFC for a good one to use for your use case

Java through version 24 has support for a small number of versions in the `java.util.UUID` class, supporting versions 1, 2, 3, and 4 but not any of the newer versions. And Java 25 isn't expected until 2030. Instead, use the popular library UUID Creator, which can be used from Maven as the UUID-creator library.

Lecture 8 — Networking

xxx

Lecture 9 — Collections

Collections Overview

A collection is nothing more than a single object that contains many others. A collection can also be called a container. The collection classes (except for arrays, which are built into the language) are in the **`java.util`** package. Collections and their permitted operations are the subject of ***data structures*** courses at the university level. Your textbook covers some of this. (Some people find data structures very interesting; ask me if you want a book reference for this topic.)

Java has a feature called *generics* that allows you to specify the type of objects you permit in some collection. If you don't use this feature, you can add any type of object to any collection and Java won't check the type. You also get a warning message even when there's no error. Generics will be discussed after collections.

There are many different types of collections. **The choice of which to use in a given design depends on what you plan to do with the objects in the collection**. The differences are the ease of use of the collection: adding, finding, removing, and iterating over the objects in the collection can be done different ways.

Collections can only store objects (references), never primitives. (However, it is rarely useful to have a collection of plain `int`s.) You can use wrapper classes to create collections of primitives if needed. (Java 5 added *auto-boxing* that does that automatically as needed.)

> With Java, it is important to remember that objects can't really contain other objects. They contain references to objects. So the same object might appear to be in several collections at once. Also, forgetting that you have some collection containing a reference to some object is a common error; if you do not remove it, the referenced object can never be garbage collected.

Java collections are specified by interfaces. (Show `Collection` interface methods.) All collection classes implement one of these interfaces. The collection interfaces extend a common parent interface, `Collection`. (The only exception is the `Map` interface, and the interfaces which extend `Map`.)

In addition to arrays, historical Java versions only provided **`Vector`**s, **`Hashtable`**s, and `Dictionary`s. These are quite ugly and rarely used anymore; however, they have been redone to implement the `Collections` interface.

Arrays are still very useful and are undoubtedly the most efficient and safest choice in many situations. **Arrays can be of primitive types (no other collection can)**, are space efficient, and allow the fastest access to the elements via an index,

of any collection. Arrays have limitations as well. For example, they cannot grow or shrink (you can't add/remove elements from them).

Unlike arrays, you cannot use the square brackets to access the elements of collections. Common operations for collections include add, remove, search, compare, toArray, and size methods. Additional methods are defined for particular types of collections; for example only Maps have a `Map.put(` *`key`*`,` *`value`*`)` method.

Additionally, most collections are `Cloneable` (since collection objects are clearly mutable) and `Serializable` if the contained objects are. Keep in mind, when you make a copy of a collection, you have made a copy of the references it contains, not the objects themselves; this is a *shallow* copy.

Also common is the need to look over all the elements in a collection. This is sometimes called *traversing*, or *visiting*, the elements in the collection. The preferred Java term is to ***iterate*** over the collection. Java supports that using the enhanced "for-each" loop and *iterators*. (These are discussed below.)

As mentioned previously, some collections do not implement the `Collection` interface. Instead, they implement an interface called **`Map`** (**show**). `Map` collections are used differently than most other collections (explained below), and it made sense to the Java designers to provide two simpler interfaces rather than one ugly interface. Java 21 (the current LTS version as of 2024) includes `Collection` and nine other interfaces that extend it, `Map` and four other interfaces that extend that.

Collection Types (Interfaces)

Each collection is of some *type*. The type determines what operations and features are allowed. When designing a program that uses collections, you must first determine the type of each. Java has many, but the most important are:

- **List –** A **`List`** is an ordered collection of elements. The order can simply be the order they were added but could be some sorted collection too. Lists support using an *index* to access the elements, much like an array. Use `Arrays.asList` to convert an array to a `List`; collections have a `toArray` method to go the other way. New in Java 8 is a `sort` method for `Lists`.
- **Queue –** A **`Queue`** is a collection that removes elements, not by name or position but in a specific order. Typical is ***first in, first out (FIFO)*** order. A FIFO Queue thus defines a list of elements, in the order they were added. Queues are often used to hold elements until they can be processed (and thus act as a buffer). We typically talk about queues as adding elements to the ***tail*** of the list, and removing them from the ***head***.

- **Stack –** A **`Stack`** is a ***last in, first out*** **(LIFO)** queue. Stacks are surprisingly useful. They are used in runtimes for function calls (the arguments), in numerical expression evaluation (Java's virtual machine uses a stack), in the implementation of various tree algorithms (such as a *depth-first search*), and in many other applications.
- **Priority Queue –** A **`PriorityQueue`** is a queue in which elements are removed in priority order, regardless of the order they were added. The constructor takes a Comparator argument that sorts added items by some priority ordering. Note you cannot add null to a PriorityQueue.
- **Set –** A **`Set`** is a collection without any duplicate elements. That is, if you try to add the same element twice to a `Set` nothing happens. `Set`s are great to ensure you have a single instance of each element. There is also in interface for a **`SortedSet`**. (Set operations are described below.)
- **Map – A Map is a collection of *pairs of elements*.** One element is called the ***key*** and the other is an associated ***value*** (which may be a collection object). With Maps, you typically need to look up a value given its key. **Duplicate keys are not allowed, and the objects used for the keys should be immutable (or subsequent lookups may fail).** Maps are very commonly used:

  userID–>Student object, URL–>HTML_page or graphic (in a cache), sessionID->EJB, etc.

  The values can be any object type, and duplicate values are allowed. (Keep in mind, if the values are mutable objects, duplicate references to the same object would be a bad idea.) It is common to have other collection objects as the values. For example, the section number for some HCC course could be a key, and the value might be a List of the students enrolled in that section.

  Maps provide three views: a `Set` of the keys, a `Collection` of the values, and a `Set` of key–value pairs (`Map.entry`). This allows you to easily pull out the current set of keys, sort those, and loop over them.

  Historical Java provided an abstract class `Dictionary` you could extend to create maps. It is now marked "Obsolete". (Apparently you cannot mark a whole class as deprecated.)

  There is also a `SortedMap` interface, which is nearly identical to `Map`, but you get a sorted set of keys. With a regular `Map`, to iterate over the keys in sorted order, you'd create a `List` from the set of keys, sort that, and iterate over the sorted list. Using `SortedMap` is easier.

There are `NavigableMap` and `NavigableSet` interfaces, similar to their sorted cousins but allow returning the closest match.

> There are no interfaces for TreeNode or LinkedList in Java collections. (How they are used in Java collections is considered an implementation detail.) If you want a Tree of objects, you have to create your own collection interface (and class). Use the existing collection interfaces for inspiration, also `org.w3c.dom.Node` and `jakarta.swing.tree.{TreeNode,TreeModel}`.

Qu: Which type of collection is most appropriate for: (a) a collection of students in our class, (b) a web browser's cache of files downloaded from URLs, (c) the "undo" list in a word processor, (d) print jobs waiting to be printed, and (e) a web browser's history list? What about for the search engine project?

> In Java 21, a new interface was added: **`SequencedCollection`**. This was done because List, SortedSet, and several other collection types had a notion of an ordered sequence of elements, but the methods on each were different. Now, they all have the same methods. See JEP 431 for details.

Collection Implementations

There are many classes that implement a given collection interface such as `List`. The general naming scheme used in Java is *ImplementationNameTypeName*. **The choice of implementation comes down to expected uses and efficiency**. For a given collection, maybe finding elements occurs much more frequently (or is less tolerant of delay) than adding and removing elements. In programming, there is often a trade-off between speed and memory use. Once you've chosen the type of collection you need (an interface), you choose an implementation of that interface that matches your needs. Note, a class that fulfills an interface or an abstract class in known as a ***concrete class***.

> **The choice of collection type (the interface to use) is far more important than the choice of implementation.** It is easy to change later the implementation, if you think you made the wrong choice; it's a one-line change.

Some implementations are useful regardless of the type of the collection. However, there are no interfaces for collection implementations. Java simply provides some concrete classes of some collection type with a particular implementation. For many collection types, the same implementation choices are available:

- **Linked(List)** – Fastest for inserting and removing elements from anywhere in the collection and can also preserve the elements in some order. Slow for searching or iterating.

Linked lists were once common data structures. However, modern hardware makes linked lists a poor choice today. Linked lists are slow because:

- The "linked" part makes CPU vectorization impossible, so operations on the elements are slower;
- The "list" part defeats caching and prefetching, so getting to the next element may stall the CPU pipeline for a relatively long time.

If you want a fast linked list, the first thing you have to do is to choose another data structure.

On top of that, implementing graph structures (such as linked lists or trees) with pointers can be dangerous; problems include memory leaks and dangling pointers. (Java supports linked lists in its standard library, and it should be safe (if slow) to use them.)

There are many alternative implementations of graphs, but all have their own trade-offs. For example, one possibility is to have an ArrayList holding nodes, another holding edges, and use Integer indexes to refer to them.

(This is already deeper into data structures than I intended to go into. Sorry.)

- **Tree** – Keeps data in some sort of order, efficiently, when adding/removing elements.
- **Hash** – The fastest for lookups but shouldn't be used if you want to maintain the elements in sorted order. Note, Java collections use a fast but insecure hash function. **Be careful when using HashMaps with user input as the keys; malicious users can perform an attack called hashDOS if you do!** Finally, Java doesn't use a cache-friendly hash algorithm currently (2024), so performance may be less than you expect. (Usually, it won't matter.)
- **CopyOnWrite** – (in `java.util.concurrent`) This makes a copy of the collection for every add (or *mutative*) operation, but is therefore *thread-safe*. This type is very slow when updating the collection but can be very fast for concurrent iteration (as no locking is required).
- ***Others*** – There are specialized implementation available too, such as *queues* implemented with arrays, sets specialized to store `enum` values, etc. Not all of these are in the `java.util` package. And of course, Java allows you to "roll your own" collection; just make a class that implements the correct interface.

Java doesn't provide all possible collection types. And for the types it does have, it provides one or two implementations at most; the standard library does not include concrete implementations of all available types. Apache Commons provides an

expanded set of collection types and implementations, each with additional methods for convenience.

> Common computer science knowledge is often tested by employers in job interviews, rather than language-specific knowledge (unless they have an immediate need for a programmer in that language). It is common to be asked to sketch or explain (or write pseudo-code that uses) common data structures such as a linked-lists, queues, stacks, trees, and hash tables. A short but good tutorial can be found at www.programiz.com.

Concrete Collection Classes

Java has concrete versions of ten types+implementations. However, there are many more available, created by methods of collections that return a wrapped version with names such as `UnmodifiableList`, `SynchronizedSet`, and `CheckedMap`. On top of that, there are five concurrent-aware types (such as `BlockingQueue` and `ConcurrentMap`) and twelve concrete implementations of those.

So the number of collections supported in Java is 20 basic types, three variations (wrapped) of the 15 basic types with some 10 concrete implementations of those, and 12 implementations of the concurrent collection types. So depending on how you count them, there are over 40 collections available in Java! And don't forget arrays.

(Apache Commons has a package (not part of standard Java) that includes even more!)

Using Collections

Best practice is to declare your collections by the type (interface), and not the implementation (concrete class):

```
List l = new LinkedList();
```

rather than: `LinkedList l = new LinkedList();`

(This example doesn't use generics and will result in a compiler warning. Generics are discussed next.)

In general, you should **always declare the type of a collection with the least-specific type that meets specifications**. This allows you to change the implementation easily without breaking any code.

Class **`Collections`** contains many useful static utility methods that can be used on any collection: max, min, reverse, shuffle, sort, copy, fill, and binary search (**show**). This is similar (but not identical) to the class **Arrays**.

To search a collection to see if it contains some element, the elements in the collection must implement a useful `equals` (***and*** `hashCode`) method. (This is discussed below.)

If you want to sort a collection, the elements must implement the **`Comparable`** interface; all primitives and standard classes (enums, strings, ...) are comparable. (**Show CollectionsDemo.java for examples.**)

(Since Generics were added to Java, many interfaces were updated with generics. This includes `Comparable` which is now `Comparable<T>`. What this means is you should declare a class as "`class Foo implements Comparable<Foo>`" and define your `compareTo` method to use an argument of type `Foo` instead of type `Object`. If you don't, you will get compiler warnings.)

A `Comparable` (or `Comparator`) must reflect a *total ordering* on the objects it compares. (This is another error the compiler can't catch, like not defining `hashCode` when required.) So what the heck is a "total ordering"? A total ordering satisfies this transitive relationship:

If $X < Y$ and $Y < Z$, then $X < Z$, for all X,Y,Z

A total ordering isn't always possible. In Poker for example, one can have:

Ace < *Five* and *Five* < *Ten*, but *Ten* < *Ace*

This violates transitivity, so the ranks do not follow a total order.

Qu: Why does this matter? Ans: because none of the `Comparable`- or `Comparator`-based utility classes and methods behave properly, including `Collections.sort()`, `Arrays.binarySearch()`, `SortedMap()`, and others, unless the element types have a total order. So for Poker, an array or `List` of `Card`s cannot be sorted using the standard library.

In such cases, it is usually better to define your own sort and search methods, than to make your class (`Card` in this example) `Comparable`. (For Poker, an Ace is only low if it would complete a straight. So it is possible to define a total order for whole hands, just not for individual cards.)

Iterating Over the Objects in a Collection

In Java 5, a new for-loop was added to the language to iterate easily over a collection, known as the *for-each* loop (or "enhanced for loop"):

```
for ( final Type element : someCollection )
{  do something with element ...
}
```

You can use this to examine the elements, but not to safely add or remove any elements while the loop is in progress. For greater control and safety, you should use the **`Iterator`** method.

Every collection has a method: `Iterator` **`iterator()`** that returns an `Iterator` object for this particular collection. An iterator has three methods: **`hasNext()`**, **`next()`**, and **`remove()`**, and can be used in a `while` loop:

```
Collection<String> c = ...;
Iterator<String> iter = c.iterator();
while ( iter.hasNext() )
{  String element = iter.next();
   do something with element ...
   if ( something )
      iter.remove();
}
```

(**Also show List.ListIterator()**, which provides additional methods such as `add()`**.**)

Note that while it is safe to add and remove elements using an iterator, that doesn't protect you from changes made to the collection by other threads while the iteration is in progress.

> Any class that can be iterated over, even if not a collection, should implement the `Iterable` interface and provide an appropriate `iterator()` method.

Collections have a ***capacity***; the number of elements for which memory has been reserved. It is fast to add an element to a collection if the capacity is more than the current size, but slow if the system needs to expand the capacity for a new element. Not all collections support viewing and changing the capacity, but if they do, the method is called `ensureCapacity` (and to view, just `capacity`).

Initially, there was no way to create a collection with some initial values (recall that Java arrays do allow that). You can come close by using the `Arrays.asList` method:

```
List<String> names = Arrays.asList(
    "Moe", "Larry", "Curly"
);
System.out.println( names );
```

At some point, most collection interfaces added a static `of(...)` method for this:

```
List<String> names = List.of("Moe", "Larry", "Curly");
```

Using the of method creates an unmodifiable collection (one that cannot grow or shrink), so sometimes you still need to use the first method.

Many collection types support a `toArray` method. This allows a quick way to "deep" print a collection (the `toString` method of collections provides a "shallow" print):

```
System.out.println(
   Arrays.deepToString( myCollection.toArray() )
);
```

Set Operations

Sets are used heavily in mathematics and in programming. There are common operations on sets that come up all the time.

If you have a `Set` A containing the elements {1, 2, 3} and a `Set` B containing {2, 3, 4}, then the *union* is {1, 2, 3, 4} and the *intersection* is {2, 3}. Java supports `Sets` (`java.util` package), but those classes don't have any `union` or `intersection` methods. How, using the `Set` methods that do exist, can you compute the union and intersection of two `Sets`?

The *union* is easy using `Set.addAll()`:

```
Set union = new HashSet(setA); union.addAll(setB);
```

The *intersection* is almost as simple:

```
Set intersect = new HashSet(setA); intersect.retainAll(setB);
```

(Other set operations are also possible, such as *set difference*. Java may not name these methods as you might want, but they are there.)

> Sets are used in Boolean searching. An OR-search is the union of the results of each word's result. An AND-search is an intersection.

One clever use of Sets is to de-duplicate a `List` while preserving order:

```
Set<String> dedupNames = new LinkedHashSet<>( names );
names.clear();
names.addAll( dedupNames );
```

Java 8 Additions for Collections

In Java 8, *Lambda* notation, method references, functional interfaces, and new interface methods, combine to make working with collections much simpler, or at least require less coding. These features are primarily of value when working with collections. Consider a `List<Student>` named `students`. To sort this list by any property of class `Student` is now trivial. For example, to sort by `StudentID`:

```
students.sort( comparing(Student::getstudentID) );
```

The method `comparing()` is a new static method in the interface `Comparator`. The method `sort()` is a new default method in the `List` interface. `Student::getStudentID` is a method reference, it provides shorthand for the corresponding lambda `s -> s.getStudentID()`.

Compare this one liner with some of the older solutions, such as using an anonymous class to define a `Comparator`:

```
Collections.sort(students,new Comparator<Student>() {
   @Override
   public int compare(Student student1,
     Student student2) {
      return student1.getStudentID()
        .compareTo(student2.getStudentID());
     }
 });
```

Or even using lambda notation (anonymous functions):

```
Comparator<Student> StuIDComparator =
(student1, student2) -> student1.getStudentID()
  .compareTo(student2.getStudentID());
Collections.sort(students,StuIDComparator);
```

(*Show CollectionSortingDemo.java*.)

Since Java 8, there are about a half-dozen different ways to iterate over a collection! For small-ish collections, any code that is clear and easy to read is fine. For large-ish collections the Java 8 **`forEach`** method for collections is often best:

```
collection.forEach( (item)->{doSomethingForEachItem} )
```

Note the `forEach` method is defined in the `Iterable` interface that collections implement. (*Show CollectionsDemo.java again.*)

For very large collections, you may need to use a different way to iterate for maximum performance.

As discussed above, Java's `Set` collection supports oft-needed set operations such as *union* (`set1.addAll(set2)`), intersection (`set1.retainAll(set2)`), and even set subtraction / set difference (`set1.removeAll(set2)`). However, all these methods are marked in the Java SE API as "optional"! Also, they alter (mutate) `set1`, requiring extra coding. Since Java 8, there are other ways to accomplish these operations using streams (*aggregate operations*, described below). These are not optional and thus always work:

```
result = Stream.of(set1, set2).flatMap(Set::stream)
         .collect(Collectors.toSet());  // Union

result = set1.stream()
         .filter(item-> set2.contains(item))
         .collect(Collectors.toSet()); //Intersection

result = set1.stream()
         .filter(item-> !set2.contains(item))
         .collect(Collectors.toSet()); //Subtraction
```

Importance of the `hashCode` and `equals` Methods

There is a required practice but one that is commonly forgotten: When comparing collections, sorting them, or even just searching them, no matter what the collection type or implementation the collection will use the element's **`hashCode`** method and **`equals`** method.

The *gotcha* is forgetting to provide a compatible `hashCode` when you add `equals`. The default `hashCode` inherited from class `Object` will work, but it renders most implementations very inefficient! **You should provide an `equals` method for classes that represent things, and anytime you do that you should also provide a `hashCode` method. The compiler can't check if you forgot, but it is a likely error nonetheless!**

Another *gotcha* is forgetting that arrays use the stupid `Object.equals` method. Use `java.util.Arrays.equals` instead, and do not forget the difference between shallow and deep equals.

> ***What is a hash code?*** (Discussed earlier in the course under *security*; you should have paid attention!) Also known as a *hash*, *message digest*, *checksum*, or *fingerprint*, a hash code is an Integer (generally, an `int`) computed from some of the fields of an object. (In general, any data can have a hash code computed for it (*demo on a file*). Different field values should result in different hash code values.
>
> Hash codes are fast to compute and generally unique; although it is possible for two different objects to have the same hash code, it is very unlikely and nothing to worry about. (There are simple techniques to deal with such hash *collisions*.)
>
> Hash codes are used everywhere: networks use them, file downloads use them, digital signatures use them, databases use them, and Java uses them in many of the collection implementations.
>
> **So if you mess up your `equals` or `hashCode` methods, collections of such objects will not work correctly!**

"Objects are equal" implies "hash codes are equal", but not necessarily the other way around. A hash code is an `int` value, and there are "only" four billion different `int`s.

The fields that you use to define equality for objects of your classes are the ones that describe the identity of an object. <u>These must be immutable</u> (read-only) fields. If they are mutable, an object might change its identity unexpectedly!

Once you've picked the fields for `equals()`, in order to satisfy the Java contract (which is not checked by the compiler!), you pick a subset of those fields to implement `hashCode()`. (Note in some cases such as objects using an `int` for an ID, you'd use the one field for both.)

> ***Never* use mutable (writable) fields to calculate hash codes** (or in `equals` comparisons). If an object is added to a collection and a mutable field is later updated that is used in the hash code calculation, the collection may "lose" the object! This happens because a collection uses the hash code to limit where to look for an object in a collection.
>
> If a class is completely mutable (i.e., has no immutable fields usable for identity), then hashCode should throw UnsupportedOperationException.

The `hashCode` method is designed to execute quickly; the benefit is that collections then can quickly decide whether two objects are definitely not equal. Only then will the more time-consuming `equals()` check be performed.

It is rare today for programmers to have to devise hashCode or equals methods. (See below.) Still, if you ever need to make your own, it is easy:

(**Show *creating a hashCode method* resource.**) There are two steps **to construct a hash code** using same fields as used for your `equals` method (Adopted from "*Effective Java*" by Joshua Bloch, (c)2001 Addison-Wesley, pp. 36–41):

1. **For each field used in `equals`, calculate hashcode depending on type of field `f`** (note the use of Java's bit-wise operators here)**:**

   boolean: `fieldHash = f ? 0 : 1`
   some integer type except long: `fieldHash = (int) f`
   long: `fieldHash = (int) (f ^ (f >>> 32))`
   float: `fieldHash = Float.floatToIntBits( f )`
   double:
   ```
   int v=Double.doubleToLongBits(f);
       fieldHash = (int) (v ^ (v >>> 32))
   ```
   Object reference: `fieldHash = f.hashCode()`
   collection or array: *treat each item as a separate field, or use `java.util.Arrays.hashCode`*

2. Combine each `fieldHashes` into the result so far (which is initialized as: `int result=17;`):

```
result = 37 * result + fieldHash
```

Generating `equals` and `hashCode` methods is simple but tedious to do for each class. Many IDEs can generate these (and other) methods for you, as can Lombok. If not using such tools, you still don't have to implement a `hashCode` method since Java version 7! It has a static **`java.util.Objects.hash()`** method, which takes as arguments the fields to use and returns a suitable hash:

```
public MyClass {
    ... // Declare final fields X, Y, and Z
    public int hashCode () {
        return java.util.Objects.hash( X, Y, Z );
    }
}
```

Finally, Java 16 added a new type of class, a *record*. Record objects are immutable, easy to declare, and have all standard methods predefined on them. Many of the objects you would store in collections or databases can (and should be) defined as records.

(Records and Lombok are discussed below.)

Caching and Lazy Initialization of Complex Hashcode (*Skip*):

Once computed, it makes sense to save the hash value so next time you don't have to compute it again. This works best for immutable objects (calculate the hash in the constructor).

If some of the object's fields used for `equals` or `hashcode` are mutable (**a very bad idea!**) and have changed, you will need to computer the hash code again. You will need to know if the hash code is valid or needs to be calculated, so use a special value (e.g., zero or some negative value) to mean "uninitialized". Or use a `boolean`. You then reset the hash to uninitialized in any set methods. Here's an example using zero to mean *not yet computed*:

```
private volatile int hashCode = 0; // uninitialized
public int hashCode()
{  if ( hashCode == 0 )
   {  int result = 23;  // 23 is arbitrary value
      result = 37 * result + field1hash;
      result = 37 * result + field2hash;
      ...
      hashCode = result;
   }
   return hashCode;
```

```
}
```

(This is listed for the sake of completeness, not because it's a good idea.)

Techniques for Dealing with Expensive `equals` Methods (*Skip*)

Usually the obvious implementation of `equals` is fast enough. If not, a possibly faster `equals` method can first check for equality of hash codes of two objects (of the same type). If the hash codes are different, you know right away the two objects can't be equal. You then compare the addresses using "==". If equal, you know both references refer to the same object (which is always equal to itself). These are fast checks.

Failing that, you must do a field for field comparison, for each of the fields chosen to indicate identity. Here's an example equals method (which doesn't use hash codes):

```
class Foo {
  ...
  @Override
  public boolean equals ( Object other ) {
    if ( this == other ) return true;
    if ( other == null ) return false;
    if ( getClass() != other.getClass() )
      return false;

    Foo otherFoo = (Foo) other;
    return
      primitiveField1 == otherFoo.primitiveField1
      && Object.equals( field2, otherFoo.field2 )
      && ...;
  }
}
```

Notice how object types are compared. If you don't use `Object.equals`, you must worry about the case where either `this.field2` or `otherFoo.field2` is `null`. (The `java.util.Objects` class has a `deepEquals` you can use if necessary.)

Another point in the above example is the comparison of classes. This means a `Manager` object won't ever compare `equal` to an `Employee`. Depending on the semantics needed (that is, all subclasses have the exact same meaning for equals), you could replace that with an `instanceof` comparison instead. For example, the `Set` interface has two different implementations, but `equals` returns true if two `Sets` contain the same elements, even if the `Sets` are of different types.

If you do redefine `equals` in a sub-class, you should include this comparison at the top of `equals` (in the sub-class):

```
if ( ! super.equals( other ) ) return false;
```

Since Java 7, you can use the static utility methods in **`java.util.Objects`** class to simplify working with possibly `null` object references, when working on `equals`, `hashCode`, and similar methods.

Since Java 16, you can use `instanceof` *patterns* to simplify the code. For example, instead of this:

```
public boolean equals(Object o) {
   if (!(o instanceof Point))
      return false;
   Point other = (Point) o;
   return x == other.x && y == other.y;
}
```

You can use this (notice the addition of `other`):

```
public boolean equals(Object o) {
  return (o instanceof Point other)
  && x == other.x && y == other.y;
}
```

If the type matches, it will be declared as the variable `other`, eliminating the conversion step of the original code. The variable has block scope.

Providing a `toString` Method

You should always add a `toString` method to your classes. It can always be used for debugging and logging, even if you can't think of any other use. More commonly, the `toString` is used when displaying a collection (as a String) or saving one to a file.

Several interfaces were mentioned here and should be used appropriately by you in production-quality code, especially if there is any possibility of your objects being used in a collection: `Comparable` is the most important, but there are others such as `Iterable` and `Cloneable`.

Records and Lombok

One problem with Java is all the boilerplate code needed. It is tedious to write and worse to read. Modern IDEs can automatically add much of this, but it still bloats your source code.

Project Lombok allows you to define simple classes simply; add the right annotation and you will automatically get constructors, get/set methods, `toString`, `hashCode`, and others. **The problem with Lombok** is that is uses an undocumented feature of common Java compilers to modify the generated bytecode (actually, it modifies the *abstract syntax tree* used internally, from which byte code is generated) to include all that. Popular IDEs have a "deLombok" function to explicitly write all the boilerplate Lombok generates, in case someday Lombok no longer works.

I don't worry that will ever happen, as other officially supported tools also use that interface. Also, Java 23 includes a preview feature of a new class API for parsing, generating, and transforming Java class files. This would allow Lombok-style transformations officially, eventually. See JEP 466 for more information.

Records — In Java 14, a new (preview) feature was added to address some of what Lombok does in a supported way. The feature was made official in Java SE 16, with JEP 395. (Records were enhanced in Java 21 with JEP 440.)

You can create a restricted class called a ***record*** that provides most boilerplate methods for you, so you don't have to write them (or read them when reviewing code). For example,

```
record Point(int x, int y) {}
```

defines a class `Point` that will automatically have two `private, final`, and non-static `int` fields, accessor (getter) methods for each (named for the field and without the word "get"), A *canonical* constructor (whose signature is the same as the header), and `equals`, `hashCode`, and `toString` methods.

You can override some of these methods (`@Override` works for these generated methods) but not other inherited methods (such as clone). For example (modified example from the *Java Language Specification §8.10)*:

```
record PointWithBounds(int x, int y) {
   // Override accessor methods:
   public int x() { return this.x < 100 ? this.x : 100; }
   public int y() { return this.y < 100 ? this.y : 100; }
}
```

It can be used this way:

```
   PointWithBounds p1 = new PointWithBounds (200,300);
   PointWithBounds p2 = new PointWithBounds (200,300);
   System.out.println(p1.equals(p2));  // true
   System.out.println("x: " + p1.x() + ", p2:" + p2);
```

Records implicitly extend class `java.lang.Record`, not `Object`, so **you cannot extend a record**. However, records can implement interfaces.

Objects of records are immutable. A record class is implicitly `final` (thus preventing another class from extending a record). You can add additional static members and instance methods, but not additional instance fields.

A nested record is implicitly `static`. Records can support varargs for the last type defined:

```
record Manager(Person mgr, Employee... employees) {}
```

You can declare additional constructors for records. In a record, a new *compact-form* constructor is allowed for the canonical version (all args are passed in):

```
record Point (int x, int y) {
   Point { ... }  // Compact constructor; note the
missing params
   Point(int x, int y, int scaleFactor) {...}
}
```

Records are a great addition to Java, especially for Jakarta EE (when defining *data transfer objects* or *DTOs*). Records have similar uses to tuples in Python. For more information on records, see Java 16 language updates: records and JEP-395.

Recall (from Java I) about enhanced `switch` and `instanceof`. You can use them with records. The pattern matching used can match components of records and assign each to a local variable. (In other languages, this feature is called *destructuring*.) Here are some examples:

```
record Rectangle(double a, double b) { }
record Square(double a) { }

static double getArea(Object shape) {
  if (shape instanceof Rectangle(double a, double b)) {
    return a * b;
  } else throw
    new IllegalArgumentException("Unknown shape");
}

static double switchWithRecordPattern(Object shape) {
  return switch (shape) {
    case Rectangle(double a, double b) -> a + b;
    case Square(double a) -> a * a;
    default -> throw new
IllegalStateException("Unexpected value: " + shape);
  };
}
```

Threading Issues Affecting Collections

Most collections are used from a single thread, or from a single threaded program. To make the collections more efficient, **they are not *thread-safe***. This means that if one Thread is iterating over some collection and another is modifying the collection (by adding or worse, removing elements) the results are not guaranteed to make sense. However, a thread-safe version of any collection object can be easily generated using `Collections.synchronizedCollection()` and other more specific wrappers. Such classes essentially *lock* the collection while one thread is accessing it, causing other threads trying to use it to wait.

Collection Wrappers

Given some collection *c*, you can use various `Collections` methods to provide additional features by *wrapping* the collection. (We've seen this before, in I/O and elsewhere. It is the ***Decorator Design Pattern***.) The wrappers include:

- **Synchronization Wrappers** – The synchronization wrappers add automatic synchronization (thread-safety) to an arbitrary collection. (`Vector` and `HashTable` have this feature built in.) Use the methods such as `Collections.synchronizedCollection()`.
- **Unmodifiable Wrappers** – These take away the ability to modify the collection, by intercepting all the operations that would modify the collection and throwing an `UnsupportedOperationException`. Use `Collections.unmodifiableCollection()` to create these. The unmodifiable wrappers have two main uses:
  - To make a collection immutable once it has been built. In this case, it's good practice not to maintain a reference to the backing collection. This absolutely guarantees immutability.
  - To allow certain clients read-only access to your data structures. You keep a reference to the backing collection but hand out a reference to the wrapper. In this way, the clients can look but not modify; you maintain full access.

Warning!! An *unmodifiable* collection means the collection object can't be changed; it does not mean the objects within the collection can't be changed!

If objects in a collection are updated then the collection may break, as elements may no longer be in correct sorted order (and as the old hashcode may not equal the new one, contains() may not work!) You should be sure to re-sort any collection whose elements may have changed. Using *immutable* object types in your collections avoids this problem (e.g. Strings — to change a String in a collection or anywhere means deleting the old String object and creating a new one). **(Show CollectionsDemo.java again, as example of *immutable* class.**

- **Checked Interface Wrappers** – wrappers are provided for use with generic collections. These implementations return a dynamically type-safe view of the specified collection, which throws a `ClassCastException` if a client attempts to add an element of the wrong type. The generics mechanism in the language provides compile-time (static) type checking, but it is possible to defeat this mechanism. Dynamically type-safe views eliminate this possibility entirely.

Lecture 9 — Generics

This feature of Java 5 took about 5 years of planning and testing to add to the language! It was possibly the most requested feature, and the largest single change since Java 1.0.

Generics add stability to your code by making more of your bugs detectable at compile time. Generics also allows you to eliminate many annoying `Object` references and casts. Finally, generics allows you to write less code (by eliminating *mostly the same* methods and classes). This is accomplished by allowing classes (and methods) to have "type" parameters: variables that stand for some type (class). When invoking the method (or using the class), the programmer specifies an actual type. These type parameters can be used (almost) anywhere within a generic or *parameterized* class (or method) that a class could. The main use of generics is for collections. (*Show API for List.add()*.)

Generic classes are (relatively) easy to use:

```
List<String> names = new ArrayList<>();
names.add( "Piffl, Hymie" );
String myName = names.get( 0 );
```

Notice the lack of annoying casts operations? Pre-Java 5, all collections were simply collections of `Objects`, no matter what you intended to use it for. The programmer had to be careful to typecast the results of `get` and other methods, and to make sure the wrong sort of object was never added—the compiler couldn't check to see if your collection contained a `Date` object by accident.

Generics only have meaning to the compiler and are only checked by the compiler. **The JVM doesn't know about generics!** (That's a lie: some reflection can show generic info, but it is limited.)

> The stripping of generic information at runtime is called ***type erasure***. Most people consider this a serious mistake that complicates development and use of generics. Type erasure was apparently done to avoid major changes to the JVM and the class file format.

Note the common error here:

```
List names = new ArrayList<String>();
names.add( "Piffl, Hymie" );
String myName = names.get( 0 );
```

This won't compile because a plain `List` is really a `List` of "anything", not a `List` of `Objects`. **Thus there is no *is-a* relationship**: A `List` of `Strings` *is-**not**-a* `List` of anythings. This won't work either:

```
List<Object> names = new ArrayList<String>();
```

This fails because, while a String is-a Object, a List<String> is-not-a List<Object>. With generic collections, the generic types must match exactly by default.

A ***raw type*** is the name of a generic class or interface, but without any type arguments. This is legal, but in most cases it will generate compiler warnings. For example, suppose we create a generic wrapper class as follows:

```
/** Generic Wrapper class.
 * @param <T> the type of the value being wrapped
 */
public class Wrapper<T> { // T stands for "Type"
   private T t;
   public void set(T t) { this.t = t; }
   public T get() { return t; }
}
```

We could use it and mix it with raw types like so:

```
Wrapper<String> stringWrap = new Wrapper<>();
Wrapper rawWrapper = stringWrap; // OK, not a good idea

// Create a raw type from Wrapper<T>:
Wrapper rawWrap = new Wrapper(); // OK, not a good idea

// This generates a warning: unchecked conversion:
Wrapper<Integer> intWrap = rawWrap;
```

(Use the `@SuppressWarnings("unchecked")` annotation to suppress unchecked warnings.) Instead of using raw types (and suppressing warnings) such as `Wrapper`, use `Wrapper<Object>` instead.

> Later, we will see how to use generics with an *is-a* match but not an exact match.

Defining Generic Methods and Classes

Declaring a generic method is easy:

modifiers ***typeParam*** *returnType name* (*parameters*) *throwsClause body*

Here's an example (from tutorialsPoint.com):

```
public static <E> void printArray( E[] inputArray ) {
 // Display array elements:
 for ( E element : inputArray ){
   System.out.printf( "%s ", element );
 }
 System.out.println();
}
```

A point worth noting is that ***method signatures*** **do not include any generic information**. The following code causes a compiler error, since there are two methods with the same signature:

```
class Foo {
 int  aMethod(List<String> list)  { return 0; }
 long aMethod(List<Integer> list) { return 1; }
}
```

By convention, type parameter names are single, uppercase letters. Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name. The most commonly used type parameter names are:

- `E` – Element (used extensively by the Java Collections Framework)
- `K` – Key
- `V` – Value
- `N` – Number
- `T` – Type (when a more specific name isn't suitable)
- `S`, `U`, `V`, etc. – 2nd, 3rd, 4th, etc., type parameters

Creating a generic class is much harder than using one, but less often needed. There are restrictions (*bounds*) and wildcard type parameters to worry about. Read Generics Tutorial (PDF) on this for detailed information. However here is one example:

Suppose you wanted a `static` utility method that given an array returned the middle element. Without generics, you must provide overloaded versions of that method, one for an array of `Object` plus several for arrays of primitive types. Of course, you'd need to cast the return value appropriately as well, a common headache with overloaded methods that just return `Object`.

Here is a generic solution: a method that takes an array of any type `T` and returns an object of type `T` (on-line as *GenericDemo.java*):

```
class ArrayKit {
  public static <E> E getMiddleElement(E[] array) {
    E middleElement = array[array.length/2];
    return middleElement;
} }

public class Test {
  public static void main ( String [] args ) {
    String[] names = { "Abe", "Bob", "Chuck" };
    String s = ArrayKit.getMiddleElement(names);
    Integer [] ages = { 18, 29, 43 };
```

```
        int i = ArrayKit.getMiddleElement( ages );
        System.out.println( "name: " + s + ", age: "
          + i );
        List list = Arrays.asList(
          "Moe", "Larry", "Curly" );
        String stooge = ArrayKit.getMiddleElement(
          list.toArray() );
} }
```

There are some restrictions on the use of generics:

- Primitive types can't be substituted for type parameters. (But you can use wrapper classes such as `Integer`, and use autoboxing.)
- You can't throw or catch generic classes.
- You can't declare arrays of parameterized types.
- `static` fields and methods belong (at least logically) to the `Class` object representing some class, and not your class's objects. So static generic methods like the following just won't work:

  ```
  Class Foo<T> { public static T aMethod () { ... } ... }
  ```

- There is no such thing as a generic constructor ("`T x = new T();`"). However, you can use reflection to create new instances of a generic class.
- Class `Class` is generic, so `Foo.class` and `fooObj.getClass()` each return an object of type `Class<Foo>`.

You can define a generic class with: `class Foo <T> { ... }`

(*Show GenericRefDemo.java, shows a generic Cache class using Reference objects, discussed below.)*

With Java 7, a simplification was added to make working with generics easier and less verbose. You can use (in some places) "<>" (known as the *diamond operator*) to tell the compiler to infer (guess) the generic type information. Here's a before and after example:

```
static class Library<E> {
  Set<Method> methods = new HashSet<Method>();
  ...
}
```

In Java 7, this becomes:

```
static class Library<E> {
  Set<Method> methods = new HashSet<>();
  ...
}
```

Generic Wildcard and Bounded Types

By default, there's no is-a relationship between generic types. However, you can do this using *wildcards* to say the type must match T or be a subtype (is-a T). You can also specify a *bound* to say the type must match T or be a supertype (parent class) of T.

To illustrate use of wildcards and bounds, here's an example of a generic Stack class with methods `pushAll` which pushes a sequence of elements onto the `stack`, and `popAll` which pops all the elements into a given `Collection`. It has some issues with the use of generics, which we will address with wildcards and bounds:

```
public class Stack <E> {
  private List<E> list;
  ...
  public void push ( E e ) { ... }
  public E pop () { ... }
  public boolean isEmpty () { ... }
  public void pushAll ( Iterable<E> items ) {
    for ( E item : items )
      push( item );
  }
  public void popAll ( Collection<E> c ) {
    while ( ! isEmpty() )
      c.add( pop() );
  }
}

public class Demo {
  public static void main ( String[] args ) {
    Stack<Number> stack = new Stack<>();
    stack.push( 1 ); // auto-boxed as Integer
    Collection<Object> coll = new Collection<>();
    stack.popAll( coll );
```

```
    }
}
```

With standard generic type parameters, types must match exactly. Thus, passing a sub-type (i.e., an `Integer` object added to a stack of type `Number`) will cause an error. In the stack example above, I would like to `push` any object that is-a `Number`. How can that be done?

You can use what is called a ***bounded wildcard type*** in such cases. Instead of "<T>", use "**`<? extends T>`**" to mean "a T or any type that extends T" (also called a "subtype of T"). Here, T is Number.

A similar problem occurs in `popAll` method. If the caller passes a collection type that exactly matches the `Stack` type (Number), no problem. But in the code above we try to put `Numbers` (a Number stack) into a collection of `Objects`. Logically, it should work since a `Number` is-a `Object`. But without an exact match, an error is issued.

This problem is the opposite of the previous one; we want to say the generic parameter is a collection of type `Number` or any super type (parent type). The wildcard looks like "**`<? super T>`**".

> Both `extends` and `super` in generics include that class itself and any implemented interfaces.

To fix the above `Stack` class, the correct method declarations should then be:

```
public void pushAll (Iterable<? extends E>... items)
public void popAll (Collection<? super E> c)
```

A rule that might help you to remember when to use which (wildcard or bound) is "methods that set something use `extends`; those that put something in an arg, use `super`". Note you should never (probably) return a wildcard type.

Copy Constructors for Generic Classes

While `clone` works for some simple cases, it won't work with generic classes. But a copy constructor will work fine! In this example you cannot assume "value" is immutable, so we need to deep-copy it:

```
public class GenericDemo<T> {
   public T value;
   public GenericDemo () { }  // no-arg constructor
   public GenericDemo (
         GenericDemo<? extends T> other) {
      value = other.value.clone();
   }
}
```

This will work fine as long as class T is cloneable (a try-block is needed here but not shown). If not cloneable, there's a complex solution involving reflection, but that only works if class T has a (deep) copy constructor! (There is no perfect answer for making deep copies in Java; every solution will fail in some cases.)

Aggregate Operations [adopted from Oracle's Java SE tutorial]

Java 8 added significant support for Map-Reduce using streams. What is Map-Reduce? The idea is to process a collection by specifying methods to be applied (mapped) to each element, with the results reduced to a single value at the end.

Streams offer two major benefits. The first benefit is (usually) succinct code. The second is streams can easily take advantage of multiple CPU cores when available. For example, consider using an iterator to filter a list (without any parallelism):

```
List<Person> filteredPeople = new ArrayList<>();
Iterator<Person> iter = people.iterator();
while (iter.hasNext()){
  Person p = iter.next();
  if( p.getAge() > 21 && p.getWeight() > 100 &&
    "Chris".equals(p.getName()) ){
    filteredPeople.add(p);
  }
}
```

Because of the explicit iteration, the system cannot automatically take advantage of additional CPUs (cores). A programmer would need to create threads, ensure no corruption of shared data occurs, and collect the results of all threads at the end. This is known to be a difficult task!

Using a parallel stream (all available cores) to filter the same list:

```
List<Person> filterPeople = people.parallelStream()
   .filter((Person p) -> p.getAge() > 21 &&
       p.getWeight() > 100 &&
       "Chris".equals(p.getName()))
   .collect(Collectors.toList());
```

(Since Java 16, that last line can be replaced with the simpler "`.toList()`".)

This works by creating a `Stream` object from a collection, which then has ***chained methods*** attached to process each item (the chained methods form a *pipeline*). Streams can make use of multiple cores if available (and the operations in the pipeline permit) to process collections in parallel; this is called *parallel mode*: the JRK splits the collection into pieces via a *spliterator*, starts each pipeline on a different core with a different piece, and combines the results of the pipelines into a single result at the end.

> The MapReduce design pattern was popularized by research at Google and used in many products. Besides streams, Java has additional support for this technology (e.g., fork-join); we will discuss it with multi-threading.

Another way to think about streams is that they provide automatic iteration over a collection (or other data source). Traditionally, the programmer needed to use a `for`-loop or an `iterator` (with a `while` loop) and have the processing steps in the loop body. With streams, the programmer states the steps to be done for each item; the JRE decides if and how to break up the iteration into multiple threads, and for each, when to fetch the next value.

(This is another use of the method chaining design pattern discussed previously, when we discussed advanced object construction techniques.)

The class `Stream` contains methods that operate on a stream and can be chained together, forming a ***pipeline***. There are two types of stream methods available: *intermediary* and *terminal*. Intermediary methods return a stream. Multiple intermediary methods, such as `filter` or `map` are combined with a single terminal method to create a stream pipeline.

A terminal method such as `collect` or `reduce` ends the pipeline, consuming the stream and returning a result. Once a stream has been traversed or consumed, it cannot be used again; you just create a new one from the collection. (Note that if using parallel streams, each core processes a part of the data, and reduces that part to a value. The JRE automatically combines the values from each core into a single value.)

The stream is able to reduce the number of lines of code by chaining method calls together. In essence, the programmer is specifying *what* to do, not *how* to do it. The compiler and JVM determine the how. This style of programming is called *declarative*, similar to SQL or HTML+CSS.

Although this chaining mechanism does save space, long chains should be avoided if possible if they make code hard to read and maintain.

Some more examples: To print the name of all members of a `List` of `Persons`, you could use a `for` loop or `while` loop with an iterator, or this code:

```
myList.stream()
 .forEach( e -> System.out.println(e.getName()) );
```

Or consider this code to iterate over the codepoints (remember a Unicode codepoint is composed of one or two `chars`) in a `String` `s`:

```
s.codePoints().forEach(codepoint ->
   System.out.printf("%c ", codepoint));
```

The alternative with a loop would look similar to this:

```
for ( int i = 0; i < s.length(); ) {
   int positionInString = s.offsetByCodePoints(0,i);
   int codepoint = s.codePointAt(positionInString);
```

```
        System.out.printf( "%c ", codepoint );
        i += Character.charCount(codepoint);
    }
```

In the code above, the standard terminal `forEach` Stream method is used to apply a method to each element "`e`". Here, a Lambda expression is used as the method (method references are also common). While functionally the same as using a `for` loop, the new stream way allows a simpler syntax and a chance to optimize execution (taking advantage of multiple cores, for example).

> Notice how filter, map, reduce, and the rest take functions (usually lambdas) as arguments. This is a classic use of *high-order functions*.

Warning: real-world text handling requires the use of the ICU libraries, not Java's standard library functions shown here.

All collections, arrays, and NIO I/O channels support a method **`stream()`** to allow you to apply an arbitrary method to selected elements. Also new with Java 8 are ***generator functions***, which produce a stream of elements from their arguments rather than from a collection or an external data source. Generators are useful when you have an infinitely long sequence (for example, a sequence of prime numbers), or live data (say you want a stream of stock quotes). In such cases you do not want to try to create a collection first! Instead, you "generate" the stream dynamically, returning elements on-demand. See the `java.util.stream.Stream.generate` method; point out it takes a `Supplier` object (or lambda). Show `FibonacciStream.java`.

A pipeline starts with a ***source*** of a stream, zero or more ***intermediate*** methods that apply to streams and produce an output stream, and finally a ***terminal*** method that produces something other than a stream (or nothing); the terminal method is the ***reduction*** operation.

> "**Intermediate operations are always *lazy***; executing an intermediate operation such as `filter()` does not actually perform any filtering, but instead creates a new stream that, when traversed, contains the elements of the initial stream that match the given predicate. **Traversal of the pipeline source does not begin until the terminal operation of the pipeline is executed."**

The various stream operations are defined in `java.util.stream`. The standard intermediate and terminal methods are defined in the class `Stream`. There are also special classes for operations on streams of `int`s, `long`s, or `double`s, which are commonly needed.

Let's modify a previous example, by only showing the names of people over 18 from a `List` of `Person`:

```
myList.stream()
 .filter( e -> e.getAge() >= 18 )
 .forEach( e -> System.out.println(e.getName()) );
```

The `filter` method takes a function that returns a `boolean` value, and simply filters out any elements from the stream for which the function returns `false`. In this example, `myList` is the source, `filter` is an intermediate operation, and `forEach` is the terminal operation.

You can use the standard methods for more complex operations. For example, suppose you wish to not only print the names of those over 18, but also calculate the average age of everyone in the list. You can't use `filter`, since that discards from the stream those under 18, so the average won't be correct. Also, you can't directly apply `average` at the end since the `forEach` method doesn't produce a stream.

Here's one way to do it, using the `map` intermediary method:

```
double average = myList.stream()
  .map( e -> { if (e.getAge() >= 18 )
     System.out.println( e.getName() );
     return e;
  } )
  .mapToInt( Person::getAge )
  .average().getAsDouble();
System.out.printf( "Average age: %4.2f%n", average );
```

In this example, the lambda provided to `map` is applied to every element, printing the name if the person is 18 or older. But unlike with `filter`, all elements are returned regardless of age. The stream of `Person` elements is then mapped to a stream of `int`s, using the `getAge` method of class `Person`. (Notice how using a method reference was simpler than using a lambda expression.) Finally, the `average` of the stream of ages is returned as a `double`.

> There are other approaches to this task, but all of them are ugly. The problem is trying to do two distinct tasks in a single pass through a stream. Generally, such tasks will prevent any performance gain from using streams as they won't automatically parallelize. In such a case, it is okay to use explicit iteration; the code would likely be clearer.

Null values in Streams — Optionals

An intermediate stream operation that might return a `null` is a problem. Because of method chaining, the `null` will cause a `NullPointerException` by the next operation in the pipeline. One possible way to deal with an intermediate operation that might return `null` is to follow that operation with a `filter`, removing any `null` values from the stream:

```
list.stream()
    .filter(Objects::nonNull)
    ...  // rest of stream;
```

But that may not be appropriate depending on the application; if you need to count the values, or have the same number of output values as input values, you cannot simply discard `null`s. Also, you need that filter after every operation that might result in a `null` value.

> The usual purpose of allowing a `null` value is to model the absence of any value. This seems logical, but the inventor of the concept later referred to `null` as his billion dollar mistake. (And that was when a billion dollars was considered a lot of money!)

Even without streams, **production-quality code must deal with possible `null` values**. For example, suppose you have a `list` of students and want the name of the academic advisor for the student named `Piffl`. You might have code such as:

```
public String getAdvisor ( String name ) {
   Student s = null;
   if ( name != null ) s = list.getStudent(name);
   Person advisor = null;
   if ( s != null ) advisor = s.getAdvisor();
   if ( advisor != null )
      return advisor.getName();
   return null; //Or a default or throw Exception
}
```

(Try this with the `:?` operator for fun.) When what you really wanted to write was just:

```
public String getAdvisor ( String name ) {
   return list.getStudent( name )
      .getAdvisor()
      .getName();
}
```

The trouble is that many programmers write the second (dangerous) version and only add `null` checks after a problem manifests.

Streams provide implicit iteration. Maybe there is a way to deal with missing values implicitly as well? That was the idea behind **`Optional`**, a generic class that is simply a wrapper around an object reference that might be `null`. An object of class `Optional` either contains a (wrapped) reference to something or is empty. By always producing an `Optional`, there will never be a `null` value.

> For Java 7 there was discussion about adding a new "safe navigation" operator, similar to the one in the Groovy language (or PHP 8):
>
> ```
> list.getStudent(name).?getAdvisor().?getName()
> ```
>
> But such a drastic change (adding new operator and bytecodes) to the JVM, changes throughout the standard library, and changes to other tools proved too much. Instead, for Java 8 the class `java.util.Optional` was added. Class **Optional** provides similar support for safe method chaining but is more verbose. However, it required no drastic changes.

The pattern for working with `Optional`s is to apply one method or function at a time in a pipeline (method chaining), similar to a stream. Consider this pseudocode:

```
return list.get( item )
   .applyIfNotNull(function1)
   .applyIfNotNull(function2)
      ...
   .applyIfNotNull(functionLast);
```

(I made up "`applyIfNotNull`"; the real Java name is either `Optional.map` or `Optional.flatMap`. I like my name better!) The idea is to not throw a `NullPointerException` at any point, but if any of the intermediate values were `null`, just return `null` at the end.

To make this work, you need to wrap the return of `list.getStudent` in an `Optional`, then chain together a bunch of `Optional.map` calls:

```
Optional<Student> optStudent =
   Optional.ofNullable( list.getStudent(name) );
return optStudent.map(Student::getAdvisor)
   .map(Advisor::getName);
```

The **`Optional.map`** method returns an empty `Optional` if `optStudent` is empty (contains `null`). If an `Optional` is not empty, the value of the function given as an argument to `map` is applied to the wrapped value. That result is then wrapped in an `Optional` (for the next stage in the chain).

If you use a method or function that already returns an `Optional<Foo>`, the `map` method would return an `Optional<Optional<Foo>>`, which likely won't compile. For such cases, use the **`flatMap`** method instead of `map`; this avoids the extra wrap operation.

As of Java 16, you can use `Stream.mapMulti` instead of `flatMap`, which is similar but better in some situations (see the Java Docs for details).

Class `Optional` has other useful methods such as **`ifPresent(`*`function`*`)`** (at the end of a chain, either applies the function to the value in the `Optional` or does nothing), **`orElse(`*`value`*`)`** (at the end of the chain, returns the contents of the `Optional` if one is there, or a default *`value`* if the `Optional` is empty), and others. (*Show API.*)

Serializable objects must not have fields of type `Optional`. If that isn't an issue, you may (once you get used to the style) use `Optional` wrapped fields.

Otherwise, you should still allow others to use your field's objects directly. You can add methods to work both ways and let the user of your code decide, for example:

```
public class Student {
   private Advisor;
   ...
   public Advisor getAdvisor() {...}
   public Optional<Advisor> getOptAdvisor() {
      return Optional.ofNullable( Advisor );
   }
}
```

There are also `Optional` wrappers for primitives, such as `OptionalInt`. These don't work exactly the same way and don't have all the same methods. I prefer to avoid using them, using autoboxing instead: `Optional<Integer>`.

One thing (of many :-) that I find confusing is that `Optionals` have methods with the same names as Streams, such as "`map`". When reading a pipeline, be sure you understand which operations are on an `Optional` and which are on a `Stream` of `Optionals`. Remember a `Stream` method returns a `Stream` (of `Optionals`), but the lambda function inside of those methods is passed an `Optional`. So inside the Lambdas, `map` refers to `Optional.map`.

Keep pipelines as short as possible and use whitespace (new lines and indenting) to help the reader understand your code.

(I wish they had chosen to add the "safe navigation" operator instead!)

Parallel Streams

So far, we have only shown sequential processing of collections. In the ***sequential mode***, each item in the stream is read and processed one at a time.

In ***parallel mode***, the data is split into multiple segments using a `Spliterator`. Each of the segments is then processed individually (possibly different threads, thus taking advantage of multicore hardware). For example, given a `List<Integer> intList`, both of these will sum the numbers in the list:

```
int sum = intList.stream()
   .reduce(0, Integer::sum);

int sum = intList.parallelStream()
   .reduce(0, Integer::sum);
```

To take advantage of multiple CPUs (cores), the only change you make is from `.stream()` to `.parallelStream()`. Internally, the JRE may distribute the work. (Warning! Don't use a parallel stream if the processing order matters!)

Here's an example showing another use of the `reduce` terminal method:

```
public class Main {
  public static void main( String[] args ) {
    final List<Integer> list = List.of( 1, 2, 3 );
    System.out.println(
      list.stream().reduce( 0,( subTotal, num )->
subTotal + num )
    );
  }
}
```

The reduce method takes two arguments. The first is the "identity" value, which is used if the stream has no elements. It is also used as the initial value of the reduction ("subTotal"). Next is a method that takes two arguments: the first is the current value of the reduction, and the second is the next element from the stream. Here, the stream of `int`s is simply summed. (There is a `sum()` method you could use instead.)

If you want to return a filtered or processed collection as a new collection, you can use the `collect` terminal method like so:

```
myList = someStream .collect(Collectors::toList());
```

There are many predefined collectors to use.

Note! A parallel stream is not guaranteed to perform better than a serial one. If you don't know the gory details of the JVM, the real hardware, and your code,

you must guess if using a parallel stream helps or hurts performance. (Even if you do know all those details, you still must guess but not as often.)

The answer in cases such as this is to measure the performance both ways. During development, add *scaffolding* code similar to this:

```
long bestTime = Long.MAX_VALUE;
for ( int i = 0; i < 1_000; ++i ) {
   long startTime = System.nanoTime();
   { code under test goes here; }
   long duration = // duration in milliseconds
      (System.nanoTime() - startTime) / 1_000_000;
   if ( duration < bestTime )
      bestTime = duration;
}
System.out.println( bestTime );
```

> Actually, the above code is a bit simplistic. Server-side Java will perform code optimization after code has been executed some tens of thousands of times. External caches take time to fill as well.
>
> To measure true performance, you need to run the code 50,000 times (say) first, then start your measurements. Furthermore, a single "best time" is not a good measure. It might be best to collect all the times in a list and either plot them or display the best, worst, and average times.

Using streams is preferred when processing very large data sets of the sort that don't easily fit into memory. Examples of such data sets include weather data, stock-market data, seismological data, voting data, geographic data, image data, and many others. These sorts of applications today are known as "big data". There exist special frameworks for processing big data, such as Apache Spark (in 2018 this is the most popular big data framework) and many others.

Lecture 11 — Weak References, Reference Objects, the Memory Model (review), and Garbage Collection

In production quality code for high memory applications (a web browser for example), Java's all-or-nothing approach to object references is too inflexible. Sometimes you would like object references that *may* be GC'd (i.e., have the GC (*garbage collector*) delete the object). Think of the memory cache of a web browser. The `Image` objects held in memory are also on the disk. So if the browser runs low on memory, it should be okay to GC such objects. But this is hard to manage in Java; you'd have to write code that periodically checked for available memory and if running low, deleted some of the `Image` objects from the collection. (If that image is needed later, it can be reloaded into memory.)

In another example, suppose you have a class that can't (or shouldn't) be extended. This is common if the class is marked `final`, or when using the static factory pattern (in this case you may not even know the type of object returned, only that it implements some interface). Now suppose you need to add some functionality to the objects of that class. For example, suppose you have a final class `Widget` and need to add a manufactured date to each `Widget` object. You can use a `HashMap` to do this:

```
mfgDateMap.put(widget, widgetMfgDate);
```

1. This causes a problem when you try to get rid of a Widget object: the `HashMap` holds a reference to it, so it never gets GC'd. (This is an example of a *memory leak*.)

To support these and similar tasks, Java provides a set of *wrapper* classes in the **`java.lang.ref`** package. These classes extend the **`Reference`** class and are used to create object wrappers that contain an `Object` reference. So rather than store a reference to an `Image`, you store a reference to a `Reference` object that in turn contains the reference to an `Image`.

So what's the point of this? Simply that when the only reference to some object is through one of these `Reference` objects, it can be GC'd anyway! (Also, `Reference` objects give your program a way to be notified when some object gets GC'd using **`ReferenceQueue`**s, but this is only needed for sophisticated memory caches.) Such reference objects have a method `get()`; this returns the reference to the real object, or `null`.

In the first example above, when the system runs low on memory (or if you manually run the GC), the system is free to delete the `Image` object even though there is a reference to it (the reference inside the `Reference` object should be the only reference to the `Image` or it won't be GC'd). Note that a `Reference`

object is not normally GC'd, but that is usually very small compared with whatever it, in turn, referenced.

Normal object references are called *strong references*. An `Object` with a strong reference cannot be deleted by the GC. Including strongly referenced objects and dead objects (those with no references), there are six levels of references known as *reachability states*. They are:

- **Strongly reachable** — an object can be reached through at least one chain of strong references.
- **Softly reachable** — not strongly reachable but reachable by at least one chain containing a soft reference. Use the **`SoftReference`** class to create such a reference. Such objects may be deleted at the discretion of the GC. It is not guaranteed but it is likely that the GC will clear older references before newer ones, so soft references provide a useful cache behavior. Use them when you want to increase the efficiency of code by caching in RAM some data that is expensive to compute or load (say across the Internet). Should the object containing the data be GC'd, you can merely reload the data, but meanwhile the data is available. This is common in "thick" clients.
- **Weakly reachable** — not softly reachable but reachable by at least one chain containing a weak reference. Use the **`WeakReference`** class to create these. Using a weak reference holds a reference to an object but says "hey GC, reclaim this object if this is the only type of reference to it.". One example use might be registering an `Observer` (for say button click events). An event source component holds a collection of observer object references so they can be notified when some event occurs. If a regular collection were used, the Observer would have to de-register itself or it will never be GC'd. Using a `Collection` of weak references solves this problem; when the Observer object goes away, so will the weak reference in the List. (Note! The Java specs don't say how the collection is implemented. If they had listened to me however, ...)
- **Finalizer reachable** — not weakly reachable (and thus in essence dead), but the finalizer hasn't been run. (Note that finalizer has been deprecated in Java.)
- **Phantom reachable** — An object has been finalized but can be reached by a phantom reference. Use the **`PhantomReference`** class to create these. Such references are only useful to see when an object has been finalized; you can never get the object through such a reference. (This is for highly advanced and rare uses only, involving *reference queues*, which allow the GC to communicate with your code about the status of object references.)
- **unreachable** — an object can't be reached by references of any type.

Using Weak and Soft References

Real world applications use many collections. Normally the reference to some object in a collection prevents the GC from removing it. This means it is up to the programmer to determine when to "expunge" objects from collections (or from the fields of other objects).

Weak references allow the application to create additional references to objects that do not interfere with garbage collection of those objects. Soft references allow the application to enlist the aid of the garbage collector by designating some objects as "expendable", which can be GC'd if needed. Put another way, **use Soft references to keep objects alive even after clients have removed their references, in case clients start asking for them again by key. Use Weak references for collections that automatically delete objects no longer in use.**

You can create and use a weak reference like this:

```
WeakReference<Widget> weakWidget =
    new WeakReference<Widget>(widget);
mfgDateMap.put(weakWidget, widgetMfgDate);
```

Elsewhere in the code you can use `weakWidget.get()` to get the actual `Widget` object. If the `Widget` object has been GC'd in the meantime, this will return `null`.

> This won't actually work, since the map would still retain the reference to the `widgetMfgDate`, not to mention the `WeakReference` object itself. Java provides a `WeakHashMap` that works the way you want: if the `widget` is GC'd, the map will notice that and delete the `weakWidget-widgetMfgDate` entry completely.

Weak references are good for providing *canonical mappings*. These are mappings from some unique identifier (say a serial number or student ID number) to an object (a Widget or Student); so a lookup table or list. Using such a `Map` you can quickly obtain a reference for some object, if it is loaded into memory already. If not, you would load it then, and add a reference to it to the `Map`. This approach uses memory as a ***cache*** and allows your program to use much less memory than if all objects were loaded all the time. (Of course, if you need to later load back some object, this will slow down your program, so there is a space versus time trade-off here.)

For example, imagine we have a database containing a large number of Employee (or Invoice, or ...) records, each uniquely identified by an employee number, and that we only work with a few of them at any given time (perhaps there are too many to fit in a running JVM). If an Employee objects we need isn't in memory

we must fetch the data from the database and create the object. This is expensive so if the object is already in memory you don't want to reload it.

You can **use weak references to refer to all employee objects currently loaded in the JVM** and provide a `HashMap` which maps from the employee number (the key) to a weak reference (the value) to the corresponding Employee object. This is a *canonical* mapping between the employee number and the Employee object, meaning there is a one-to-one correspondence between employee numbers as keys in the map and Employee objects which have that employee number. If the Employee object is already in memory a quick lookup gets you a reference. If not, you load it and add a weak reference to the HashMap.

Before rolling your own cache, note that JPA already has one built-in.

We would not want to prevent an Employee object from being garbage collected if no other part of the program is using that object since we could always retrieve the object from the DB. You want to avoid using a soft reference, since this makes the GC wait until running out of memory to reclaim the objects. Weak references give the garbage collector greater freedom to collect objects more quickly.

This approach allows one to work with a subset of a very large collection of objects with much less memory than would be needed if you tried to read in all the objects at once. If you don't use weak references you have to work around the garbage collector to determine when to "free" a given object, removing the benefits of the garbage collector and automatic memory management.

This is common enough that Java provides a class **`WeakHashMap`** in `java.util`. Note in a `WeakHashMap` it's the keys that are weak references, not the values. If a `WeakHashMap` key becomes GC'd, its entry (both the weak reference key and the strongly referenced value) is automatically removed from the `Map`. Using a `WeakHashMap` rather than a regular `HashMap` requires no changes other than the switch from `HashMap` to a `WeakHashMap`. If you follow the standard convention of referring to your maps via the `Map` interface, no other code needs to even be aware of the change.

A `WeakHashMap` is a great way to associate extra data with items of some class, without having to edit or extend the class. That's very handy if the class in question comes from a third party. For example, if some class Foo is wonderful but you need to have an additional Date stored for each Foo object, you can do so by:

```
WeakHashMap<Foo,Date> fooExtraData =
   new WeakHashMap<>();
...
Foo f = new Foo();
fooExtraData.put( f, new Date );
```

And if you have more that one item to associate? Then you can use a `Map` of `Foos` to `Maps` of key-value pairs:

```
Map<Foo, Map<String,Object>> fooExtraData =
  new WeakHashMap<>();
```

You can use a similar technique to add data to `Enum` values, only you would use an `EnumMap` instead of a `WeakHashMap`.

You can build a cache from a normal `HashMap` by mapping normal (key) objects (e.g., a `String` or an `Integer` or common) to `SoftReferences` (e.g., to `Image` objects).

ReferenceQueue

Once a weak or soft reference starts returning null, the object it pointed to has become garbage and the reference object is pretty much useless. This generally means that some sort of cleanup should be done to remove useless reference objects from collections. `WeakHashMap`, for example, removes such defunct entries to avoid holding onto an ever-increasing number of dead `WeakReferences`. But other collections do not automatically remove such entries.

The `ReferenceQueue` class makes it easy to keep track of dead references. If you pass a `ReferenceQueue` into a weak or soft reference's constructor, the reference object will be automatically inserted into the reference queue when the object to which it pointed becomes garbage. You can then, at some regular interval, process the `ReferenceQueue` and perform whatever cleanup is needed for dead references. For example, you can check the queue for a soft (or weak) reference object that is now useless, and remove it from collections:

```
ReferenceQueue rq = new ReferenceQueue();
Foo f = new Foo();
SoftReference<Foo> sr = new SoftReference<>(f, rq);
f = null;  // remove only strong reference
List<SoftReference<Foo>> l = new ArrayList<>();
l.add( sr );
...
sr = rq.poll();
if ( sr != null ) l.remove( sr );  // Cleanup list
```

Note that reference objects are small, so it is common to not bother with cleanup. However that is a mistake for long-running programs; over months or years lots of useless objects will consume significant memory and slow operations on collections.

Discussion/project/demo: Create a cache using soft references of (`Images` or web pages or customer records or ...)

Java's Memory Model (review)

All data is either local method variables, which are stored on a fixed-size stack, or objects, which are stored in the ***heap*** (a.k.a., the *free store*). (This is probably true of every programming language.) Java only allows primitive and object references to use the stack. The underlying operating system has limits on these for every process, including the Java virtual machine (which is also a process).

Using stack-based memory is very efficient, especially since Java doesn't bother to initialize items on the stack. But the stack cannot be used for any item whose size is not known at compile time. Also, stack-based data is copied when making method calls; so large items, even if the size is fixed (for example, an `Integer` object), would hurt performance and use up memory.

> Java SE and Jakarta EE (so Oracle-based Java) contain another stack, the *operand stack*. This is used instead of registers; the Java virtual machine has no data registers. The Android Java (Dalvik) VM does have registers and lacks this stack.

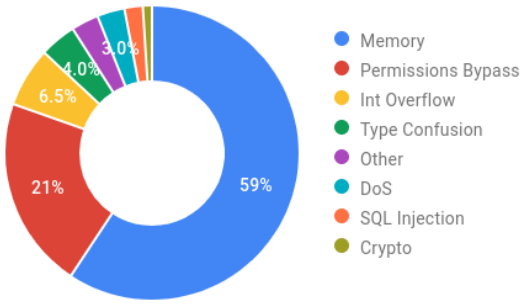
In some languages, any fixed-size item can be placed on the stack; **in Java, all objects go on the heap**, fixed size or not. Items on the heap require more effort to create and to use. Even in a simple case, the program must first read the object's reference from some variable (on the stack), then access the data in the heap.

> The heap works like a fancy restaurant: When you show up, you say "party of 4" and the host needs to locate an unused table of that size. If 2 more friends show up unexpectedly, you need to ask the host for a larger table. If the next table over is empty, it can be moved next to your table to make room. Otherwise, a larger table must be found (if there is one) and your party is moved there. The old table is cleaned so it can be used by another party.
>
> The heap works the same way: when your program creates an object, the JVM must find an unused place in the heap that is large enough. If the object grows, the JVM must add space to that object (if there's any free space right after it) or find a new location in the heap that is large enough and move the data there.

Garbage Collection

Garbage collection was added to Java to help with ***dangling references*** (where you have a reference to an object that was already deleted), ***memory leaks*** (where the programmer allocates memory with *new* and forgets to free it up with *delete*), and

other memory-related errors. Java doesn't even have a "delete" operator. How common are memory related bugs in software? Take a look:



Types of Critical and High severity vulnerabilities fixed in Android Security Bulletins in 2019

This chart shows the sources of security issues in Android in 2019, which is written mostly in C. **Memory-related errors are the largest issue by far**, followed by permission errors and integer overflow issues. These errors occur despite all the testing, code review, and other tools and techniques Google used to avoid them.

Using Garbage collection in the language automates much of the memory management your code must otherwise do, and thus eliminates the possibility of many memory related bugs (sadly, not all).

The basic idea is simple: when the system runs low on memory, the garbage collector (GC) thread runs and frees up the memory for all objects that are no longer referenced (*unreachable objects*). So in Java, to delete an object, all you need do is set its reference to `null` (assuming it has only one reference, the common case). The next time the GC runs, it will delete the object and free up the memory it used.

> The JRE keeps track of all "live" threads. Any object reachable from any live thread is considered *reachable*. Note Thread objects themselves are reachable as long as they are "live", whether or not you save a reference to them.

Using a GC really does make the language simpler to use and safer too. Unfortunately, there are a number of related issues not solved by the GC:

- Running the GC takes many CPU cycles and can lead to user-noticeable delays. This can be a huge issue in performance-sensitive code.
- There is no guarantee that the GC will free up enough memory for your program (although running the GC several times in a row *may* free up a little more memory on each pass). If too many large objects remain reachable, you can still run out of memory. It can be difficult to "null out" all references to some object at the right time. So a *memory leak* is still possible.
- There is no guarantee the GC will be run at all, which makes calls to objects' *finalizers* (discussed below) unpredictable.
- Many resources are scarce but take little memory, so you can easily run out of them while the GC doesn't run, thinking there is plenty of free memory.

Examples of such resources include open file references, graphic contexts, disk buffers, sockets, and others.

Solving these problems requires careful techniques, the use of some special classes for memory management, and an understanding of the issues involved.

In languages without garbage collection, the programmer must manually free every object from the heap when it is no longer used. Forgetting to do so leads to *memory leaks*. Other issues of manual memory management include double-free some heap object and use-after-free.

Java helps with this issue by using a *garbage collector* ("GC"). The GC runs periodically and attempts to locate objects that are no longer usable (*reachable*), and free up their memory. Note the Java language does not guarantee any particular GC method; that's up to the JVM vendor to choose. GC can hurt performance (in some cases, stopping the JVM completely) whenever it runs. And it usually runs at unpredictable times.

> There are other interesting ways to deal with dynamically sized objects, such as with Rust's ownership system. Rust is slowly becoming (2021) more and more popular and is inspiring new programming languages. Rust is considered *memory-safe*.

One way to implement GC is to use ***reference counting***, which tracks how many references exists to each object. This model fails if two or more objects have references to each other: the ref count never drops to zero. It has other problems too.

Modern JREs uses more complex algorithms, but they are understandable in terms of a simpler model called ***mark-and-sweep*** (or *mark-and-compact*); this should be sufficient for a Java programmer to understand. In this model, the GC uses two phases. First all *live* objects are identified by starting with a set of *roots*, the directly accessible references (e.g., local object reference variables on each live thread), and seeing what references can be found from them. This process is recursive, and marks all objects in memory as *live* that can be reached (*reachable*). To do this safely. the JRE must pause all other running threads. When this is done, all remaining objects are *dead* and are finalized and then deleted. (This *sweep* phase can be done concurrently with other threads running.)

> In practice, Java GC works differently (the exact behavior depends on which JVM you have). In Sun's Java 6 JVM, an algorithm called ***concurrent mark and sweep*** is used. The heap gets split into three areas ("generations") and those areas are subdivided into smaller areas. Each area can use a different GC algorithm. The *permanent area* is for the JRE classes and doesn't use any GC since these are always needed. The *old* area uses a mark-and-sweep method called *mark-compact* in what is called a *major* collection. The *new*

area uses a method called *copy-compaction*; this area is split into three spaces (*Eden, survivor space 1 and survivor space 2*). All new objects get created in Eden, and in a *minor* collection the live objects in Eden are copied into SS1 (and the dead ones are finalized and reclaimed). In the next minor collection, the live objects in Eden and SS1 are copied into SS2, in the next Eden plus SS2 live objects are copied into SS1. This continues until an object has been copied a certain (tweakable) number of times, then it gets copied into the *old* area instead.

Ever since this concurrent mark and sweep GC was used, newer garbage collections also use this ***generational* approach**. So a run of the GC may not examine all objects in the heap when run! This is why running the GC several times in a row may recover addition memory each time.

Most GCs in Java include parameters you can adjust, to control how much CPU time is used, how many threads are used, how long (maximum) pauses should be, the sizes of the different generational areas, etc. While you shouldn't worry about performance-tuning the GC at first, or in most cases, it might matter for some uses. You will need to read about the current version of Java and the extra options available for this! For Java 24,see the HotSpot VM Garbage Collector Tuning Guide.

Demos: Start some Java app, then run `jps` (to get its *PID*), then `jmap -heap` *`PID`*. Also: `jmap -histo:live` *`PID`*. To browse a snapshot of the heap, use `jmap -dump:live,format=b,file=dump.dat` *`PID`*, and then open the GUI tool VisualVM and load the file `dump.dat`. Also show `jconsole` *`PID`*.

The Java HotSpot JVM supports many (five at least as of Java 24; for some reason the included *Shenandoah* experimental GC is not mentioned in most official documentation, but its been present for many years) different GC algorithms; you can select one (and tweak it) using command-line args.

Selecting a GC and proper parameters for it was such a burden for programmers that since Java 5 the choice is made automatically by the JVM (See the HotSpot JVM GC Tuning Guide). Still, you can over-ride those choices if you want, using extra "-XX" options. Some of the (non-standard and may vanish or change) GC parameters are shown here. (These links are in our class' resources, but you should look for a more recent guide that covers the newer GC algorithms.)

I cannot find a recent official guide for these options! However, you can list them out with the command:

```
java -XX:+UnlockDiagnosticVMOptions -XX:+PrintFlagsFinal -version
```

There's over 700 of them! (The test will only cover the first 600. Just kidding!)

Different Garbage Collection Algorithms Supported in Java 24

(See the HotSpot JVM GC Tuning Guide for more details.)

The **serial collector** uses a single thread to perform all garbage collection work, which makes it relatively efficient because there is no communication overhead between threads. Useful for small-ish sized heaps or when running on single core systems; with large heaps the pauses can get annoyingly long for user applications. The serial collector can be explicitly enabled with the java option `-XX:+UseSerialGC`.

The **parallel collector** is also known as *throughput collector*. It is a multithreaded "generational" algorithm are optimized for machines with multiple CPUs or cores. Using multiple threads allows garbage collection to proceed in parallel with the application thread so the application can proceed without perceptible pauses. By default, the parallel collectors allocate one thread per processor. This collector is optimized for very large (gigabyte or larger sized) heaps. It should provide very fast throughput with minimal pauses. It requires using a mark-and-compact algorithm for the *old* generation. This might mean pauses of >1 second, so this is more suitable for server applications.

The Garbage-First Garbage Collector (or G1 GC) was introduced in Java7 (and JRE6u14). G1 GC uses all available CPUs (cores, hardware threads, etc.) to speed up its "stop-the-world" pauses (when an application's Java threads are stopped to enable GC). It also works concurrently with running Java threads to minimize whole-heap operations during stop-the-world pauses. G1 is *generational* like nearly all the GC algorithms. It treats newly-allocated (a.k.a. *young*) objects and objects that have lived for some time (a.k.a. *old*) differently. It concentrates garbage collection activity on young objects since they are the ones most likely to be reclaimable. G1 also performs heap compaction over time. Compaction eliminates potential fragmentation problems (e.g., occasionally very long pauses). With G1, stop-the-world pauses should be <40mSec. G1 is the default GC and is well suited for client and server applications.

> Unlike other HotSpot *generational* GCs, in G1 there is no physical separation between the young and old generations. Instead, there is a single contiguous heap which is split into same-sized regions. Each region holds either young or old objects (but not both). With G1 only regions that appear to contain mostly reclaimable ("dead") objects have the live ones (the *survivors*) copied (*evacuated*) to a new region, then the whole region is reclaimed. The other regions aren't touched. The concurrent marking phase is only to determine which regions should be cleaned (*swept*) of dead objects.

The **Z garbage collector** (ZGC) was introduced for Java 11. It features the speed of the parallel collection but with stop-the-word pauses guaranteed to be under 4 ms (compared to > 200 ms for some of the others). (So the best of all worlds. See opsian.com for a nice overview.)

The **Shenandoah garbage collector** is a low-pause time garbage collector, introduced in Java 12 (and back-ported to older versions later, such as Java 7), that reduces GC pause times by performing more garbage collection work concurrently with the running Java program. Shenandoah does the bulk of GC work concurrently, including the concurrent compaction, which means its pause times are no longer directly proportional to the size of the heap. Garbage collecting either a 200 GB heap or a 2 GB heap should have the similar low-pause behavior!

Note the Java GC documentation does not list Shenandoah, but it is there. See blogs.oracle.com for a discussion. Shenandoah is still considered experimental, but it has be usable for over a decade!

How to pick the best GC algorithm: Java 17 includes five different GCs. There were others as well in historical versions of Java but those are not deprecated or completely removed in modern Java.

> While the many GCs were available since JVM 11, they have been greatly tweaked in more recent versions. So older advice you find on the Internet is not necessarily any good! The modern JVM can auto-select a GC for you, and that will be fine most of the time. Also, note these garbage collectors are only guaranteed to be available in HotSpot Java VMs. Use IBM's JVM and you have entirely different GCs available.

You can select which one you want with "`-XX:`" options (unfortunately, not well documented). Start with the *default* (which is currently G1). If pauses are too long (or GC isn't run often enough), try *a newer GC* if available in your JVM. Each choice comes with various "`-XX:`" options you can use to fine-tune the behavior. **However, *you really should know what you are doing before fiddling with any of the "`-XX:`" options!***

(See https://blogs.oracle.com/javamagazine/understanding-garbage-collectors and https://blogs.oracle.com/javamagazine/understanding-the-jdks-new-superfast-garbage-collectors for details of the GCs included with Java 15.)

Finalizers (*Skip; Deprecated in Java 9 and marked for removal since Java 18*)

A *finalizer* is a special method (`protected void finalize()`) that is called automatically by the GC just before deleting the memory associated with that object. If the finalizer adds a reference to an object to a list someplace (or a static reference), the object is *resurrected* and may be GC'd again later. However, it is guaranteed the finalizer will never run more than once for any object. You use finalizers to reclaim non-memory resources, such as open files, sockets (internet

sessions), database connections, file locks, etc. You rarely use them, since there is no guarantee they ever get called, so you must provide other methods to provide such cleanup (such as `Graphics.dispose()` and `File.close()`) and you must remember to call them. The finalize method is more of a safety net in case objects are deleted unexpectedly (say in response to an exception).

Here's an example ~~stolen~~ adopted from "The Java Programming Language", 3rd Ed., by Ken Arnold, James Gosling, and David Holmes, (C)2000 by Sun Microsystems Inc., pub. by Addison-Wesley. Page 317:

```
public class DBFile
{  private FileReader file;
   public DBFile ( String fileName ) throws
     FileNotFoundException
   {  file = new FileReader( fileName );  }
   public synchronized void closed () throws
                    IOException
   {  if ( file != null )
      {  file.close();
         file = null;
   }  }
   protected void finalize () throws Throwable
   {  try
      {  close();
      }
      finally
      {  super.finalize();  // always use this
}  }  }
```

The programmer is expected to call close directly, but just in case the `finalizer` will close it. Note too the use of the `finally` clause of a `try` block, to ensure the superclass's `finalizer` is invoked. (Even classes that don't extend anything still extend `Object`.)

Finalizers are impossible to implement correctly or use safely, and were deprecated in Java 9. You should replace finalizers with `java.ref.Cleaner` "actions" and/or try-with-resources, or possibly use a shutdown hook (if you are okay with the cleanup not happening until the JVM shuts down.

Interacting with the Garbage Collector

In early Java, the `System` class had two static methods: `gc()` to run the GC and `runFinalization()` to finalize unreachable (dead) objects that haven't had their `finalize` methods run yet. Since Java 2, these methods as well as the methods `freeMemory()` and `totalMemory()` can be called from a runtime

object:

```
Runtime rt = Runtime.getRuntime();
long f = rt.freeMemory(); ...;  rt.gc();
```

> The details of Java's memory model (JMM) require some understanding of modern computer hardware: memory, virtual memory, multiple CPUs, and multiple caches. This can be important when understanding multi-threading issues in Java. For more details, see gvsmirnov.ru/.../jmm-under-the-hood. (It's dated 2004 but is still a good resource.)

Java 24 (available as a preview feature since Java 17) includes a feature called the foreign function and memory API. This will allow the use of code and data (memory) outside of the JVM, allowing one for example to store data in memory so you can invoke a C program on it. See JEP 454 for details.)

Lecture 12 — Testing (with JUnit) and Using Assertions

[*Adapted in part from "expert one-on-one J2EE Design and Development", Ch. 3, by Rod Johnson, Wrox pub. (c)2003 Wiley Pub.*]

> *Failure isn't an option. It's mandatory.*

Here we will discuss testing, starting with what it is and why it is important. Then we will go over the types of testing and their purpose. We will cover related concepts as well, then go into implementing tests using JUnit and other tools.

Testing is one of the most important areas in software development, yet one which students (and unfortunately others) ignore completely or at best pay lip-service to. Modern development methods such as *Extreme Programming* (XP) and *Agile Software Development* advocate a ***test first*** method of programming: First you write the tests from the specifications, requirements, or design (depends on the type of test), and then you write the actual code that makes the tests pass. Testing can do a lot more than find bugs in your code!

Recall that software is developed in phases: requirements, design, implementation, etc. There is lots of research and general agreement that finding bugs in the early phases of a software project is much cheaper than finding them later. At least one study found the cost of finding and fixing bugs goes up 10 times with each passing phase of a project! A bug spotted in the requirements phase can often be corrected with a 5-minute phone call to the client, but that same bug spotted during acceptance testing may require a complete re-design and re-implementation of the project!

> Testing was originally one of the final phases, but clearly that didn't work, and testing activities begin in the first (requirements) phase in modern software development.

The bottom-line is that even if it doubles the time it would otherwise take during coding to develop tests, it will still save time and money overall.

It is common to have most of the code devoted to testing in a serious software project. For example, SQLite is written with about 155,800 lines of C code (not counting blank lines or comments). But it also has over 92 million lines of test code (nearly 600 lines of test code for each line of production code)! (Reference: www.sqlite.org/testing.html) This is not unusual for databases, servers, operating systems, flight systems (on airplanes), and other serious projects. Small apps such as flashlight (on a cell phone), a calculator, etc., have much less testing code, but still should have some; maybe half the code is for tests. The importance of testing to ensure software quality and reliability cannot be overstated.

Tests have other uses too. In *test-driven development* (TDD), you write tests when designing the code's API. Tests can ensure compliance with various standards and

laws. Tests can be used to show the code is done (acceptance tests). You can document various requirements with tests. Of course, finding bugs in the code is a primary use too.

> Testing can help in unexpected ways. One problem with code is ***accidental correctness***: Your code happens to work some of the time on some specific data, but not in the general case. As a trivial example, suppose you check this code with "`x=2`": "`result=x+x`". You get "`4`" and are happy, right? As it happens, you were supposed to have "`result=x*x`" and the incorrect code accidentally produces the expected result.
>
> "I am often surprised that assertions [and tests] that I expect to fail actually go through just fine. This usually means one of two things: Either the functionality is already there by some fluke (cf. accidental correctness), or my test is incorrect. In either case I can then fine-tune and adapt the assertions to make sure that they fail and thus test some missing functionality." — Manuel Woelker from unit testing revelations.

In addition to finding bugs, tests can be used to specify the required behavior of methods and classes. In some situations, test cases become part of the requirements. This is because a method signature may tell you something about how a method can be called, but it doesn't say everything about how it should be used. For example:

```
public Color pickColor ( float r, float g, float b )
```

This doesn't really tell you what the range of expected values for the three arguments are, or what happens if negative values are sent. The full specification of a method needs to include the ***semantic*** information too. Commonly this is done with comments (Java docs), with formal specifications of ***preconditions***, ***postconditions***, and ***invariants***, or a combination. That sort of information can be expressed as a series of tests too.

> **It is not possible to completely test any non-trivial program**, to the point where you can prove no bugs remain. (*Show "**Test case self-assessment web resource***; do in class if time permits.) That doesn't mean you shouldn't use testing! Rather, initially you will need to "cherry-pick" a few dozen tests that you have the time and budget for. As bugs are discovered, or new features proposed, you add more test cases to the initial set.

Test code must be simple and clear. If too complex, you can't be certain the test isn't flawed. Even simple, short test methods should be "tested" (or at least reviewed by others). I usually manage that by inserting a debugging line of code into the method under test, altering the result. In this way, I can force the test method to fail, so I can examine the resulting messages. Then I comment-out or remove such debugging lines. (A better way is *mutation testing*, discussed below.)

A ***test case*** or just "test" is a test of a single item: a feature for some tests, a method/class/package (i.e., a module) for unit tests. In Java, each test is usually implemented as a separate (test case) method in a test suite class. **A test case must include all the inputs (and possibly a sequence of actions) to be done, along with the expected result.** The test *passes* if the *SUT* is invoked with the inputs and produces the expected output.

A ***test suite*** is a collection of test cases that are related. A project can have many test suites or just one. Often there is one or more for unit tests, system testing, and acceptance tests. Often each test suite is implemented as a separate class.

> When testing, the code being tested is referred to as the ***system under test***, or the ***SUT***.

Methods of Testing

Tests can be classified by either how they work or their purpose (type). All tests work in one of two ways:

- **Black-Box Testing** — This means to **test to the public interface (or specifications)**. Such tests should be built from the design documents and *not* by inspecting the code. In essence these tests make sure the code does what it is supposed to do, without testing the implementation details.
- **White-Box Testing** — Also called ***glass-box testing*** or ***structural testing***, such tests are designed to thoroughly cover all the code. This requires you to inspect the code. For example: make sure all `if` statements are tested in both branches, every method is called, every loop body is executed at least once, etc.

Types of Testing

There are many different types of tests done. Each type has a different goal and is often preformed differently. The types of tests (in no particular order) include:

- ***Ad-hoc*** **testing** is a commonly used term for software testing performed without planning or documentation. These tests are intended to be run only once, unless a defect is discovered. Such an approach is better than nothing, but is only useful for small ("toy") problems.

  > Ad hoc testing is sometimes called *exploratory testing*. Ad hoc testing has been criticized because it isn't structured, but this is also the main strength of exploratory testing: important things can be found quickly. It is performed with improvisation; the tester seeks to find bugs by any means that seem appropriate. (This takes experience, talent, and luck. So not a technique suitable for novices.)

- **Unit Testing** — These are tests that cover the functionality of a single unit of code: in object-oriented languages, this is usually a class, but could be individual methods or whole packages. (In other languages, a unit is a function

or a module.) Unit tests are sometimes black-box tests and don't usually test the implementation; *assertions* (see below) can be used for implementation testing (and fuzz testing). However, unit tests can be white-box tests which many be necessary to ensure good *test coverage* (discussed below). Generally, a framework is used to simplify unit testing. For Java, JUnit framework is very popular (and discussed later).

Typically, each "unit" requires multiple test cases to thoroughly test; to keep test methods simple, they should only test a single aspect of the unit.

Unit tests are based on the design documents, not the implementation or the requirements/specifications. The point is to ensure the unit implements the design correctly.

> Many students don't get this point at first. You often write tests *before* there is any code to test, once you have at least a partial design. The idea is that later, when you want to know if the code (that is eventually implemented) does what the design says it should do, the tests should pass. Until then, all the tests are expected to fail. (Indeed, without skeleton classes with the proper stub methods, your tests won't even compile!)
>
> Sometimes you don't have a formal design when starting. That's okay! It is far easier and faster to scrap a faulty design before you've written anything but a few tests. Tests might let you know if the design is bad.
>
> By writing tests first, you can design the API of your software at the same time; this is a good way to ensure a well-designed API for reusable code. (See *test-driven development*, below.)

Unit tests often use both legal and illegal sample data. Production-quality code must not crash servers or corrupt data. So you need to test with illegal data too.

It is important to pick test data that doesn't have special properties that might mask errors (accidental correctness). Often you duplicate test case methods with a few different sets of values. Each separate item tested has its own ***test case***. The set of test cases for some unit comprise the unit test and is called a ***test suite***. (Often you have multiple units tested, commonly all of them, in a single test suite.)

A test case using only legal input values that shouldn't cause any exceptions or errors, is called ***happy path testing***. You should make sure your test suite includes a one (or a few) such tests, *in addition to* (numerous) tests for failure conditions.

Unit Testing void Methods or Methods with Side Effects

Testing `main` (or any methods with side-effects or that return `void`) requires special handling. The problem with `main` is when it prints to the console (or shows a GUI). While that can be tested directly with some (okay, a lot of) effort, a better way is to design methods for *testability*. Don't have `main` (or other side-effect producing methods) also do any computation. Instead, have `main` pass arguments to another method to do the computation, which returns the data to be printed, saved to a file, or whatever. Then you can easily test those other methods. (This is an application of the *single responsibility principle* (SRP). This is also known as *designing for testability*.

- **Boundary-value Unit Testing** — These are unit tests that cover **what happens when you use extreme but legal data**. Examples include passing in negative, zero, or huge values for primitive types; Strings, arrays, and collections of zero length (or very large), etc. Often simple tests using normal values won't catch the "off by one" errors in loops, or division by zero or overflow bugs.

  Most methods you need to test take one or more arguments. Generally, unit test methods don't take arguments, so to test some method with many sets of arguments you would need to duplicate your test method with different values in each test method. Running the same test cases with different parameters is known as ***parameter (value) coverage*** or ***parameterized tests***.

  Duplication of code is bad, but creating complex tests with loops or that read data from external sources is also bad. Modern test frameworks such as JUnit have some support for parameterized testing: test cases that take arguments where you specify a source for the parameters, and the framework will invoke the test method for each set of arguments. (JUnit features are described later.)

- **Integration Testing** — Also known as ***system testing***, this is the testing done after unit testing, to ensure a sub-system or the system as a whole correctly implements the design. (That the design correctly implements all the requirements is a different question, usually not done by code testing.) System testing need not test an entire project; individual software modules are combined into sub-systems that can be system tested as a group. This testing is needed since even if individual units are perfect, they still must interact with each other and the results of that have yet to be tested. Integration testing is easier when modules are loosely coupled, but still needed.

  In essence, you set up sample persistent data, sample users, etc., test some *use case* (one "feature" of the requirements), then examine the resulting data (and log files, if any) to see if the (sub)system did what it was supposed to do.

There are several approaches you can take to integration testing. Most make use of *stubs* (sub-systems not yet written but invoked by the code you are testing) and *drivers* (test code that invokes the methods of the sub-system you are testing). A **top-down approach** tests the top-level modules, which will need to invoke stubs (or sometimes *mock objects,* discussed later) for any modules not yet written and unit-tested. A **bottom-up approach** uses drivers to invoke the methods of lower-level modules, when the upper-level modules are not yet written. A big-bang approach waits until all modules are written, and tests them without stubs or drivers.

Cross-cutting concerns such as logging, metric reporting, and security, are also part of integration testing. It is possible to test logging and metric reporting as part of unit tests instead, however.

- **Regression Testing** — These are tests (both unit and system) designed to make sure that what was working before is still working now. After adding new features or bug fixes to code, you need to **run regression tests to make sure your changes haven't broken anything**. If you saved your tests and they are of the black-box type, they can be used for regressing tests.

  > As bugs are found, a good practice is to write a new test case to reproduce that bug and add that test case to your regression test suite. Only then go and fix the bug.

- **Acceptance Testing** — Sometimes called *validation testing*, these **test the system as a whole lives up to expectations**. Unlike other black-box tests (unit and system), these test the code against the requirements and specifications, not against the design. Often a contract will specify payment only after successful acceptance testing.

  Such tests usually involve the customer playing the role of a user and putting the system through various ***use cases*** or ***scenarios***. Such manual tests are more common with acceptance testing than with unit testing. (In many *agile* methodologies, acceptance tests are written in the requirements phase, for each use-case or user story.)

- **Load and Stress Testing** — Part of acceptance tests (and sometimes systems testing), these test the whole system (or major part, such as the web server application) under increasing load and for long periods of time. When you only have loads in the range required it is called ***load testing***. Such testing can show stability and possibly uncover concurrency (multi-threading and multi-processing) problems. Besides slowly increasing the load, you can test what happens if the load increases suddenly. This is called ***spike testing***.

  When the load is increased beyond the design point it is called ***stress testing***. Such tests can often uncover weak spots in the design or implementation, and

also show if the application ***degrades gracefully*** (that is, it doesn't crash and/or corrupt the valuable data).

- **Quality of Service (QoS) Testing** — This is similar to load testing. It is sometimes referred to as *Quality Assurance (QA) Testing* (Technically, QA is not a type of testing). You need to make sure the application loads quickly and is responsive when running, even when network connections go down or bandwidth is limited. Testing response times and other quality metrics is testing against the *non-functional* requirements. (QA starts with the requirements to ensure the system always does something sensible and is repeatedly checked while the product is under development.)

  The Boeing 737 MAX 8 and its *Maneuvering Characteristics Augmentation System* (MCAS) software failed in the Ethiopian Airlines crash in March, killing 157 people. Or did it? According to Trevor Sumner, CEO of Perch Interactive, in this Twitter thread, "Nowhere in here is there a software problem. The computers & software performed their jobs according to spec without error. The specification was just shitty."

  The crash was caused by a lack of QA testing. For various reasons (such as cost or the fact that auditing other's code is not rewarding in most cases), much of these new 737 Max jets were 'self-certified' by Boeing. In the case of the 737 MAX, Bob Tapscott, chief designer and architect of the Jeppesen Aviation Database (JAD) that supplies flight directions to flight management systems, said "One decision Boeing made, that seems very difficult to understand is why some safety features were an option. In my experience I did not ever meet anyone in the industry, that felt that life-saving features might be optional."

  Aja Hammerly, developer relations advocate at Google Cloud, worked in software QA for seven years before moving into other roles, and describes the processes required for effective software QA this way:

  "First thing you want to do is you want to figure out, what does operational look like, and figure out what the risks of things going wrong are so that you can prioritize your testing based on high-risk areas first, usually, or at least ensuring that your high-risk areas are sufficiently covered," she said. "And those risks can depend on the system, which also means the fundamental QA is looking in looking deeply at the system, not just the spec but of actual implementation."

  In her experience, most of the "interesting and potentially troublesome" issues are found in integration testing, where systems come together. Those systems may work perfectly on their own, but

> when they're put together bad things happen. Consider, for example, the $125 million Mars Climate Orbiter that was lost in space in 1999 because Lockheed Martin's engineers used Imperial measures (miles) when programming navigational commands for the probe, while NASA used the metric system (kilometers) to issue those commands from its systems, and nobody noticed. The JPL review panel that looked at the loss concluded that it was an integration QA failure in which the process that should have caught the issue went wrong.
>
> Most QA time is not spent on what Hammerly calls the "happy path" – the cases in which everything works properly. Instead, she said, 80 – 90 per cent of her time was spent on edge cases.
>
> [From an article in www.itworldcanada.com.]

- **Fuzz Testing** — Testing with random data, including valid but unusual values as well as illegal values, is known as fuzz testing, or simply "fuzzing". This is discussed in more detail, below. While not commonly used for small projects, it is increasingly used for larger projects.
- **Smoke testing** — Also known as a ***build verification testing***, this is testing performed on a system prior to being accepted for further testing. This is a "shallow and wide" black-box test that looks for answers to questions such as "Does it crash when launched?", "Does the UI show?", or "Do the buttons on the windows do anything?" If you get a "no" answer to basic questions like these then the application is so badly broken there's no point in more detailed testing. When automated tools are used the tests are often initiated by the same process that generates the build itself.

  In the context of software engineering, a smoke test generally consists of a collection of tests that can be applied to a newly created or updated computer program. In this sense **a smoke test is the process of validating code changes *before* the changes are checked into the larger product's official source code collection.** Such tests are often done automatically by the build software or (source code) *check-in* software.

  > The term *smoke testing* come from the plumbing trade. It refers to the first tests made after repairs or first assembly, to provide some assurance that the system under test will not catastrophically fail. After a smoke test proves that the gas pipes will not leak or an electrical circuit will not burn, the assembly is ready for more stressful testing.

There are even more types of functional tests, but they are rarely used:

- **Concurrency testing**, testing the interactions of multiple threads, can be important too, but is very hard to arrange. Testing Jakarta EE (multiple clients

accessing a cluster of servers) is even more difficult and requires special tools and techniques (briefly mentioned later).

- **GUI testing**, or more generally the testing of user interfaces, is also difficult (but not impossible) to automate. Mostly, a human tester interacts with the GUI. Automated tests use a script to simulate a human. There are nice tools for this for testing web-based user interfaces. Some such tools also have a "monkey test" mode, where the test system automatically "clicks" and "types" at the GUI until it crashes.

Non-functional Testing and Auditing

Any of the above test types that check the functionality of the code are ***functional tests***. Functional tests include unit, integration, acceptance and the other types of testing discussed above.

There are other tests that often require humans and cannot be easily automated. These include **compliance auditing/testing** with various user interface (UI) standards; testing licensing agreements; HPI, PII, and other sensitive data handling complies with various laws and regulations; and security compliance. In the absence of SLAs (*service level agreements*), such testing can include adequate response times, pleasing user interface, etc. Sometimes a company attorney and marketing person must also check the system and documentation for compliance. Sometimes external audits are done (often for security and to guarantee compliance.)

Non-functional tests and audits are for these other aspects of the system. Non-functional tests include the usability of the user interface(s) (e.g., for sight-impaired users), performance SLAs, load limits, graceful degradation (stress testing), compliance testing, documentation accuracy (and completeness) verification, security testing, recovery testing (system comes back online with no data loss after unexpected reboot), I18N testing, operational procedure testing, etc.

(Note there is some overlap in the definitions of functional and non-functional testing. There shouldn't be any, but items such as graceful degradation, user interface (UI) acceptability, reliability, and response times may not be considered functional aspects if there is no SLAs for them in the project requirements.)

> ***Rubber ducky testing*** is an informal term that refers to a method of debugging code, not testing at all. Also known as *rubber ducking* or *rubber duck debugging*. The idea is that you can debug your code by forcing yourself to explain it, line-by-line, to an inanimate object such as a rubber duck. (Yes, the first programmer to do this did have a rubber duck on his desk!) The expectation is that upon reaching a piece of incorrect code and trying to explain it, a programmer will notice the error.

Static Analysis of Code

When writing code, it pays to use tools that can detect as many issues as possible early. Such tools examine source code files looking for bad style, common logic errors (e.g., incorrect format strings for `printf`, illegal regular expressions, etc.), and anti-patterns (poor code). Such tools are called *static analysis tools*.

> One of the earliest such tools was developed in the 1970s for the C language and was called "`lint`", likely because the program caught things similar to lint in wool. Regardless, the name stuck and such tools are also known as *linters*.
>
> The linter included with `javac` ("`-Xlint:all`") is good, but there are many others that are often considered better.
>
> I have used PMD, SpotBugs and CheckStyle. SonarQube is also popular. (I like SpotBugs and CheckStyle, but YMMV.) All these linters are available as plugins for all major IDEs.
>
> A popular linter for Java and many other languages is the commercial tool Coverity (although it has an online version that can be used for free in some cases).

(While most compilers now include many checks, the various lint tools have evolved as well and are usually well worth using. For C and C++ you can use *splint* or the simpler *RATS* (looks for security issues only). For JavaScript, try *jslint*. For Rust, you have *clippy*.)

> These static analysis tools can also be invoked automatically from *continuous integration* (CI) tools, and some test frameworks.

Source code style compliance **testing** is part of static analysis and often part of unit testing. It either happens during code review, or increasingly commonly, automatically. Most IDEs have a setting where you can have your code reformatted as you type (or when you save). (I find this leads to a sloppy coding style and I don't use it. The IDE built-in tools generally won't allow exceptions either.)

Instead, you can run a tool such as CheckStyle and have any violations cause a build failure. Or you can have style checking (and other static analysis) produce reports that can be examined during code review (so violations do not fail the build). CheckStyle is discussed further, below.

> I find one reason some programmers, especially students and novices, don't like using such tools it they feel they take too much time. Nonsense! Finding errors early reduces the time to produce software.
>
> Another reason is that they are often sloppy coders, and static analysis produces an overwhelming number of warnings and errors. That is not the

case if you compile frequently, linting as you do, and fixing the problems each time.

Coverage — Choose Test Cases Carefully!

Not every possible set of inputs and conditions can be tested. You want to have the smallest number of tests that as fully as possible test ("cover") all of the features in the requirements and/or specifications. For each feature tested (or for each unit tested), you want to thoroughly test it. For example, if one input in a test is a filename, you should test with: very long names, short names, null names, names with strange characters, and names with illegal byte sequences.

You can use random generators to pick values such as numbers or to generate text strings or valid filenames, in a given range (except for boundary-value testing). This removes a tester's bias which might mask errors.

Code coverage is another meaning of the term coverage that mostly applies to unit tests: the test suite runs every path through your code. If every line of code (that you are testing) is executed during tests, you have 100% code coverage; this metric is known as *lines of code covered*. (This is a good reason to use white-box unit tests.) This is related to implementation testing, which uses *assertions* within the SUT rather than separate test code.

While 100% feature coverage in acceptance and integration tests is a good goal to shoot for, it is a poor idea to aim for 100% code coverage in unit tests. Testing getter and setter (and other trivial) methods is rarely useful, as you end up testing if Java assignments and return statements are working. You need to test that your code is bug-free, not that the Java runtime (JRE) is bug-free!

Code coverage is not best measured by lines of code covered, although this is common. Instead, ***Branch coverage*** tests the possibilities (each branch) of every conditional (if-statements, loops, switch, etc.). The number is not dependent on how the branch is implemented.

Why **branch coverage is better** can be shown with a contrived example:

```
public int getNameLength(bool isManager){
   User user = null;
   if (isManager) {
       user = new User();
   }
   return user.getName.length();
}
```

If you have a single test case for this that passes `true` to the method, you will see 100% line coverage. But you only have 50% branch coverage! You need a second test case that passes `false` to see the error in this code.

High coverage does not guarantee high code quality nor even through testing. Low test coverage (say 70% or less) does indicate more testing is needed. 70% to 90% code (or branch) coverage is typical for most personal and business applications. High risk systems, including health care, avionics, and military systems, often have 100% coverage requirements using specific metrics.

Code coverage tools can tell you what parts of your code were not included in any test. An example tool is jcoverage (see also other code coverage tools). A popular coverage tool to use with Maven (and/or Jenkins) is JaCoCo.

Demo: Eclipse Marketplace, install EclEmma. (This is based on the JaCoCo library.) Open Triangles project in Eclipse, right-click-->Coverage As-->Junit test. Show source code coverage markers. Show results in Project-->Properties-->Coverage too.

There are other types of coverage, but they are less common and not as important. (Software Engineers love metrics!) Some are *function coverage* (percent of functions/methods invoked at least once during testing), *statement coverage* (better than counting lines as you may have multiple statements on one line), *condition* (or *predicate*) *coverage* (test with each part of every boolean expression set to true and false, not just the whole boolean expression as in branch coverage), *modified condition/decision coverage* (MC/DC), which is a combination of branch and condition coverage. (This is the metric often mandated for high risk systems.) Don't confuse function coverage with *functional coverage*, a term that applies to acceptance tests (percent of the functionality of the system that was tested).

In unit testing, with methods that take arguments, you need to invoke the method with lots of different values for each parameter (or you wind up only testing the happy path). For example, an `int` argument should be tested with negative, zero, positive common values, plus boundary values and illegal values. `Strings` should be tested with the string set to null, zero-length, one character, typical values, long values, plus boundary values and illegal values. So each test case method needs to be duplicated for each set of parameters, or a single test case needs to be run multiple times with different parameters. See *parameter testing*, below.

Microsoft now (5/2020) ships Windows with a kernel feature "InitAll" that initializes all stack memory prior to use. This causes a performance hit but helps with the uninitialized memory bugs. This step is necessary according to Joe Bialek, a security engineer in the Microsoft Security Response Center (MSRC). Bialek argues (also in this blog post) that static analysis can't find

> everything, [Testing] can't identity all bugs of this type either, and manual code review doesn't scale to very large code bases, and is prone to error. So the solution is to have the operating system initialize all memory all the time.

Mutation Testing: Measuring the Quality of Test Suites [*From* Bell-sw.com]

Even with 100% branch/condition coverage in your unit tests, you may have undiscovered bugs. This can happen with poor quality tests, or if your tests themselves have logic errors in them. So simply having lots of test cases (high coverage) is not a guarantee of through testing. **How can one know if their test suites are of high quality?** You should test them, of course!

One way is to make errors in the code deliberately and then see if your tests will catch the problem. This is what *mutation testing* does.

A *mutation* is a slight change to your code (that you are testing) that introduces a bug. The altered version of the code is called a *mutant*. After creating one or more mutants in your code, the test suite is run. What should happen?

If your tests fail, we say the mutant was killed. If tests pass, the mutant has survived. **A surviving mutant shows a flawed or missing test case.** If any mutants survive, the test suite should be improved to kill them. (The test report shows the surviving mutants to the programmer; they must determine the issue.) In most cases, it means you have to add the missing tests or rework some of your test code.

Mutations can be as simple as removing a statement, changing a statement (e.g., change `++x;` to `+x;` or `--x;`), changing a value (e.g., change `5.0/9.0*(f-32)` to `5.0/9.0*(f-31)` or `5/9*(f-32)`, or `int count = 0` to `int count = 1`), or changing an operator (e.g., change > to <). There are so many possible mutants, a tool is needed to generate and test them.

A *mutation percent score* for your tests is calculated as:

*killed_mutants / total_mutants_tested * 100*

100% means you can have high confidence that your testing is high quality, if you tested billions of mutants. Mutation testing however can take a very long time to run; you often simply have to let it run in the background for however long you can afford. The more testing done, the higher your confidence in your test suite quality.

For Java, PiTest is a good mutation testing tool. It works with Maven and generates the mutants in the byte code, making this testing much faster.

Pairwise and Combinatorial Testing

Even if you have a high-quality and high-coverage test suite, you still haven't thoroughly tested the system! (Thorough testing is impossible.) This is because a large number of **errors may only appear when certain conditions occur together:** for example, very long filenames that also contain weird characters, or the user interface language is set to Spanish and negative numbers are used. It is quite possible that testing each condition separately will not reveal such errors. (These are also the hardest bugs to find and fix.)

It is in general not possible to test all combinations of all features. For unit testing, it is rarely feasible to include 100% branch coverage or conditional coverage, plus through parameter coverage (different values in all possible combinations). There would simply be too many test cases and take far too long (and if run on the cloud, cost far too much money). Trying to test every boundary value test in combination with every other is even more impossible.

However, all is not lost! A technique known as ***pairwise testing*** can generate a small number of test case inputs that test every possible pair of features! Often such tools can generate unit test cases for pairwise parameter value coverage as well. Usually, commercial software is used to generate the test inputs from a list of features (or method inputs/parameters) and the number of values/conditions of each.

> This technique is mostly used for integration, whole system, and acceptance testing. It is rarely applied to unit testing, but can be.

Three-way and *n*-way testing is also possible and useful for high risk systems. This type of testing is also known as *combinatorial interface testing* (CIT). Such tests can take a very long time on a dedicated cluster of fast computers (days to tens of weeks)! However, these tests are hard for humans to generate because there could be thousands/millions of them. So a test-case (actually, test case parameter lists) generator tool is needed.

As a trivial example, suppose I needed to (acceptance) test software with three features, each of which contains two types of test values. Then testing each feature by itself takes six test cases (and likely would be done by manually creating test cases when there are so few). Testing all possible combinations would take nine test cases. But just four(!) pairwise tests will test each feature in combination with each other feature. Since **an error caused by a single feature is likely (but not guaranteed!) to manifest when tested in combination with some other feature(s)**, just the four test cases will provide very good functional coverage. For example:

Test Case	OS Type	CPU	Protocol
1	Windows	Intel	IPv4
2	Windows	AMD	IPv6
3	Linux	Intel	IPv6
4	Linux	AMD	IPv4

Usually of course there are many more than three features (how many features does a word processor or web browser have?), and each feature may have several different values or conditions to test. For just 10 binary features, only 13 test cases are needed for all three-way feature interactions, as opposed to 120 if test cases aren't chosen carefully (a total of 2^10=1,024 test cases is possible). For real-world software with dozens of features with many possible values each, there could be (many) millions of combinations. Using pairwise test case generators can reduce that to thousands or even hundreds. Clearly, the benefits are enormous!

> Pairwise coverage won't generally find errors that only occur when 3 or more features have specific values (conditions), but such errors are very rare anyway. Still, high risk systems may require 3-way or *n*-way (where $n > 3$) testing.

See pairwise.org (or other sites such as www.softwaretestinghelp.com) for a list of tools, many free, which can generate the coverage table. One popular tool is the freely available PICT tool from Microsoft (sadly no longer maintained). Note, you still need to turn that into JUnit or other tests to run. (*Demo ACTS; show PDF tutorial p18 (Ch. 4), for a demo of generating tests from method inputs, not features*.)

There are other considerations to using CIT. If your system has configuration options, you need to run all the tests for each combination of those as well. Also, if an *n*-way CIT test fails, you can't be certain without additional testing (or code examination) which factor or pair of factors caused the failure.

Certain systems require slightly different approaches to CIT, such as event-driven system where you need to test different sequences of events. Coming up with the system model (the list of parameters and their possible values) is still a manual process, so if you miss something, it won't be tested.

> On the other hand, if you include too much, testing gets too expensive and you may not be able to compete all scheduled tests. For example, the popular DBMS "MySQL" was tested using CIT; it took a large cluster of computers months to complete the test suite just once!

CIT testing is an active area of computer science research.

Fuzz Testing

As mentioned earlier, testing with random data that includes valid but unusual values as well as illegal values, is known as fuzz testing or simply "fuzzing". Although such testing was probably done with punch cards in the 1950s, fuzz testing as known today was invented in 1988 by the university of Wisconsin professor Barton Miller. Apparently, he was working from home, connected by a dial-up line during a stormy night. The storm induced random noise on the line and messed up some commands. So he turned the issue into an assignment for his students, challenging them to feed random input which he termed *fuzz* into various Unix programs to see which ones would break. Somewhat surprisingly, about a third of the programs broke! Prof. Miller published a paper about it two years later and fuzz testing was born.

Unlike other forms of testing, the fuzz tests cannot know what the results of any test case should be. In addition to crashing, hanging, and detection of illegal memory access at runtime, such testing relies on *assertions* (discussed later) in your code to detect any illegal states. If the input causes an assertion to fail, the test shows a flaw in your code. Fuzz testing in general cannot actually test that your logic is correct or even pinpoint the problem. **To use fuzz testing fully, the programmer should use lots of assertions.** Without assertions, the fuzz test can only provide a stack trace to help you figure out the issue.

Over time, fuzz testing became more sophisticated. For example, by compiling a program to validate memory access, fuzz testing could find failures when programs would attempt to access illegal memory, or uninitialized memory. (Java doesn't suffer from such problems!) Runtime checks for overflow or array bounds checking can be enabled. The checks available depend on the language, compiler, and OS used. (Java doesn't have runtime over or under flow checks but it does array bounds checking.)

When fuzz testing, a program could fail in two ways: it might crash or it might hang. It may crash due to memory issues, assertion failures, or for other reasons. To catch the latter, you need to run the program with a timeout. Generally, a human must look at all the failures to see exactly what happened.

The software that generates fuzz tests and data and runs the tests is known as a ***fuzzer***. (Note how this is similar to tools that generate mutation tests.)

Today, there are different types of fuzzers:

Generational fuzzing uses random number generator to come up with tests. Unfortunately, the vast majority of those are not legal and tell you nothing. The few that are legal generally end up testing your input validation code only. To test interesting logic in your software, your test inputs must be valid and (mostly) pass input validation. So modern generational fuzzers need some guidance to generate

inputs. Fuzzers that know the structure of the input are called *smart*; pure random input fuzzers are called *dumb*. Smart fuzzers use a model or grammar to help generate inputs that will make it past your validation code.

Mutational fuzzing takes existing tests and generates new tests from those, taking (for example) test values from one test and using them in an unrelated test, or mutating the data in random ways (but that still make it past input validation). This technique avoids the problems with generation fuzz testing but results in "blind spots", since all values came from a human (the original test data).

Finally, a fuzzer can be aware of the program structure and use that knowledge to generate more useful tests (that test more of the program's code). Thus fuzzers can be black-box, white-box, or gray-box.

All fuzzing techniques can be combined, to provide better fuzz tests.

In fuzz testing, a ***runner*** is a program that takes the generated input from a fuzzer and runs the software (with the right options enabled) and under a timeout. There are many fuzz testing frameworks you can use for different languages.

The advantage of fuzz testing is that it can be automated and can generate and run many, many tests reasonable quickly. Using the cloud, billions (or trillions if you can afford it) of tests can be run in a few days. If any fuzz tests result in a crash or assertion error, a human must examine those cases. Note, however, that fuzz testing cannot prove your program is correct! It's just that the more tests run, the higher your confidence will be.

> Automation is generally necessary since so many tests are run. Reportedly, in 2016 the Google OSS-fuzz project produced around 4 trillion test cases in a week. Running them and collecting the interesting (failure) results requires automation.

Fuzz testing is good at finding security bugs, all sorts of memory abuses (including buffer overflows), unhandled integer over- and under-flow, faulty data validation and sanitation, off-by-one errors, concurrency issues, and more. Some serious issues were found with fuzzing. One famous example was the heartbleed OpenSSL Bug.

Some currently (2025) popular fuzzing tools for Java include Google's OSS-Fuzz and Jazzer, both of which are powered by LLVM's libFuzzer (and in fact share some code).

JUnit Testing Framework

While you can perform unit tests without any frameworks, that is tedious and error-prone. Test frameworks make selecting and running tests much simpler, and generally show test output in a graphical way that's easy to interpret.

JUnit is a very powerful unit testing framework for Java and includes many features we won't discuss here. It is ubiquitous today, although other testing frameworks exist. The first versions of JUnit had incremental improvements, until JUnit 4. JUnit 5, known as JUnit Jupiter, is a complete rewrite with many new features and improvements. However, it is different from (incompatible with) prior JUnit versions. Development of JUnit 4 has stopped.

You can **download the newest JUnit 4** from junit.org and put the `junit*.jar` file someplace on your classpath. Note, an additional downloadable jar file includes the API docs for JUnit, which can be handy to have. (Unzip the jar, open the top level `index.html` file, and bookmark it in your favorite browser). For examples of using JUnit 4, see the old cookbook at the JUnit home, or look at the examples and other docs at the JUnit 4 Wiki site. Check with Maven Central for the latest version of JUnit 4. JUnit 4 include both a GUI and non-GUI test runner you can invoke from the command line.

JUnit 5

The current version of JUnit is 5, a.k.a. JUnit Jupiter, released in 2017. JUnit 5 is both leaner and more powerful than JUnit 4; it has an extensible design. To work with JUnit 5 from the command line, a stand-alone Jar file is available that contains all dependent Jars and includes a non-GUI test runner. (Since all IDEs include GUI JUnit test runners, the developers felt it would be a waste of time to have one.)

Some of the features include *parameterized tests* (you can pass values into test methods from external sources), *repeatable tests* (for testing for timing errors a.k.a. race conditions, or performance issues), and a whole lot more. The demo in the class resources shows a trivial class with two tests, one is repeated ten times.

JUnit 5 also includes a "vintage" test engine, so your project with old JUnit 4 (or 3) tests can still be used with JUnit 5.

JUnit 5 is highly modular; a given project only needs some of the Jar files. (There are about 16 Jar files altogether! I think the assumption made by the developers is that you would be using Maven or another build tool, and only need to list just a few artifacts in your POM.xml file or whatever; the rest get pulled in automatically by following dependencies.) Previous versions of JUnit were a single Jar file.

To use JUnit 5 since version 5.4, only a single dependency is needed. (Older versions need two or three dependencies.) Just add the latest version of `org.junit.jupiter:junit-jupiter` as a dependency. All JUnit 5 release versions of the Jars are found on Maven Central, so even if not using Maven, you need to download the Jar file(s) from there. Note the modular nature of the code means several developers could work on different parts concurrently, so updates are frequent.

JUnit 5 is a much better testing environment, but it is completely different from older JUnit versions. There is currently (2019) little documentation on using JUnit 5, but I expect that to change. Not all IDEs and build tools support JUnit 5 (2018), but that should change soon as well. For now, we will use JUnit 4 in most of this course. See also the JUnit 5 User Guide.

Unit Testing with JUnit 4

Update `CLASSPATH` to include the `junit-4.13.2.jar` file if necessary. Don't forget to also add "`.`" to `CLASSPATH` too. (JUnit 4 uses `CLASSPATH` to locate all *your* class and all *your* test code, and **not** extension directories or the Java 9 module path. So if testing your own Jar files, make sure to list them on CLASSPATH.)

Create a class to hold your test suite (set of unit tests, which become regression tests; note you can have multiple test suites if you wish):

```
import org.junit.*;  // The various annotations
import org.junit.runner.JUnitCore; // Test runner

import static org.junit.Assert.*; //AssertX methods

public class MyAppTestSuite {
    test case methods go here
}
```

Now all you do is add test cases. Each JUnit 4 test case is a `public void` method that takes no arguments and that is annotated with "`@Test`". The body of test methods is discussed below (it's the same for versions 4 and 5).

Unit Testing with JUnit 5

If using JUnit 5, there are many Jar files. But you can you use a single Maven dependency:

```
<!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter -->
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.4.2</version>
    <scope>test</scope>
</dependency>
```

Or you can use junit-platform-console-standalone, a single Jar that contains everything needed.

> For my preferred setup, I install the JDK someplace without version numbers, `C:\Java` (or `C:\Program Files\JDK`, if you prefer). That directory should be defined as the value of the environment variable

`JAVA_HOME`. Next, I create a folder to contain all my extensions that I download as Jar files: `C:\Java\MyLibs`. I can place any Jar file in there, such as `junit-`*`version`*`.jar`. Finally, I use the `CLASSPATH` wildcard mechanism: `CLASSPATH=.;C:\Java\MyLibs\*`.

With this setup, I can update the JDK (to the same location) or replace old jars with new ones, and not have to change any environment variables. (The JDK installer may not remove previous JDK versions; do that manually before installing a new version of the JDK.)

Create a class to hold your test suite:

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.*;  // Annotations

public class MyAppTestSuite {
    test case methods go here
}
```

Since unit tests shouldn't need access to private members of your classes, they don't need to go into the same package. Nonetheless, it is common to do so, since you can then use the JUnit testing framework for more than black-box unit tests.

By that same token, to make testing easier it is common to make all testable class members package-private rather than private, so they can be accessed by your test class and methods.

Now all you do is add test cases. Each JUnit 5 test case is a `public void` method that takes no arguments, and that is annotated with "`@Test`". For JUnit 5, other kinds of methods can be used too, but they still need this annotation.

The test cases can be in any order. (They don't necessarily run in the order you list them.)

A good naming scheme for such methods is test*DescriptionOfTest*. (You don't get extra points for short names!) (JUnit 3 required method names to start with "`test`", but not v4 or v5, and many people think it is a waste to add "test".)

A nice feature of JUnit 5 is the ability to set a *display name* for both test suites and test cases. Just add "`@DisplayName("Happy Path test for method foo")`" annotation to any test method (or to the test suite class).

Structure of a Unit Test Case

These unit tests all have the same pattern of code: *setup - invoke a method(s) to test something - examine result*. With behavior-driven development, these three steps are known as *given-when-then*. Here's an example unit test case for TextKit.pad():

```
@Test
@DisplayName("TextKit.pad() happy path test")
public void PadWithLegalInputs () {
   // Given:
   final int num = 512, minWidth = 6;

   // When:
   final String result = pad( num, minWidth );

   // Then:
   assertFalse( "null String returned from: pad("
   + num + ", " + minWidth + ").", result == null );
}
```

The test case methods contain any code you need, and one or more "`Assertxyz`" statements (See JUnit Java docs) to check the results. For example: **`assertTrue`**("message", cond), **`assertFalse`**("message", cond), `assertSame`("message", objRef1, objRef2), or **`assertEqual`**("message", obj1, obj2). There are many others too (*show* JUnit 4 Java docs *for class Assert or* JUnit 5 API docs *for class Assertions*).

(For JUnit 5, the message goes at the end instead.)

> It is not uncommon to have one assert call per test method. Thus, if testing legal inputs with three different sets of data, you could have three different (but very similar) methods. The advantage is that when a test fails, no further code in that test method is run. So if you had additional assert calls in the same method, the additional tests won't be run.
>
> However, I think it is okay to group multiple test data sets into a single test method, as long as the method is testing just one aspect. This approach has benefits too: a data-driven test method (using JUnit 5) is simple and clear and easy to add many data sets (including the expected result for each). And if one data set fails, it is usually pointless to run the others until that problem has been fixed. For example, a boundary value test fails on the first of ten test data sets; the other tests for the same boundary value test are not needed since you know what the problem is already. (Note that reporting is by test methods, not individual assertions.)
>
> (Junit 5 has support for *parameterized tests*, where you can specify the source of the test data and it will automatically loop over your method.)

Generally, I use `assertFalse` in JUnit 4 which causes the test to fail (and displays the message) if *cond* evaluates to `true`. (I've always found that confusing!) For example, this tests that `num` <= `max`:

```
assertFalse("num greater than max!", num > max);
```

With `assertFalse`, the expression matches the failure message in this style. The other way works fine too, so it's a matter of personal or team style:

```
assertTrue("num should be less than or equal "
   + "to max!", num <= max);
```

(Remember that with JUnit 5, the message is the second argument and the boolean expression is the first.)

What if a test should throw an `Exception`, or return within a specified amount of time? You can pass optional parameters to the JUnit 4 `@Test` annotation. This test *should* throw an `Exception`:

```
@Test(expected=IndexOutOfBoundsException.class)
public void testFooIndexOutOfBoundsException() {
  List emptyList = new ArrayList();
  Object o = emptyList.get(0);  //this in foo
}
```

Note that uncaught exceptions (other than the expected one) cause a test to fail, but without any helpful error message.

JUnit 5 does this differently; the `@Test` annotation takes no arguments. Instead, you need to write code like this:

```
@Test void someTest () {
    Exception e =
        assertThrows(ArithmeticException.class,
        () -> calculator.divide(1, 0));
  assertEquals("/ by zero", exception.getMessage());
}
```

The optional JUnit 4 `@Test` parameter **`timeout`** causes a test to fail if it takes longer than a specified amount of clock time (in milliseconds). The following test fails:

```
@Test(timeout=100)
public void testFooTimeOut() {
  for(;;) ;
}
```

JUnit 5 uses `assertTimeout(` *`duration, methodCall`*`)` instead:

```
@Test
void someTest () {
    Interger result = assertTimeout(ofMillis(200),
        () -> {...; return result;});
    assertEqual( result.equals(expected) );
}
```

This assertion uses the main thread and won't abort if the time is exceeded. Using `assertTimeoutPreemptively` instead runs the test in a different thread which will be aborted after the time limit. The time limit is a `java.time.Duration` object.

> Generally, unit tests cover public interfaces of classes or packages, so you can simply import your packages and classes under test into your test suite. You can create a new package `foo_test`, for the tests for package `foo`.
>
> But you may wish to unit test some package-private or private methods too, or use the JUnit test framework for other sorts of testing. Your test suite class(es) will need access to your code to use any non-public methods or fields.
>
> There are a number of techniques for this situation. The simplest way is to put the testing classes into the same package as the code being tested, and make fields and methods package-private instead of private. Another possibility is to nest the class you wish to test inside your unit test class. That gives your test code special access but can cause deployment issues. A good discussion of this is found on StackOverflow. (One technique discussed there, using reflection in the tests to access non-public class members, is not possible in modern Java; it was considered a security hole!)

Additional JUnit 5 Features

With JUnit 5, you can group tests together and have groups and subgroups using `assertAll`:

```
@Test void someGroupedTest () {
    assertAll( "group heading",
        () -> assertNotNull(result),
        () -> assertTrue(result == expected)
}
```

Code blocks (() -> {....}) can contain other `assertAll` groups; if one assert fails, remaining nested ones won't be run.

Tests not ready to use can be marked with `@Disabled` annotation. For example:

```
@Test
@Disabled("Disabled until issue #17 is closed")
void someFeatureTest () { ... }
```

When doing boundary value tests (or sometimes other tests), you need to run the exact same test code but with different data each time. *Parameterized tests* make it possible to run a test multiple times with different arguments. They are declared just like regular `@Test` methods but use the **`@ParameterizedTest`** annotation instead. You also need to specify the source of the data with another annotation;

the annotation used depends on the source, which can be: an array of values, the output of some method, or an external file. These test methods take arguments (the test data and sometimes, the expected value for each test case); normal tests methods take no arguments. Here's a simple example using an array of values:

```
@ParameterizedTest
@ValueSource(ints = {0, 1, 22, 1_000_000, -1, -10})
void someFeatureTest (int testValue) { ... }
```

You can use **`@csvSource`** instead to pass in sets of values, for example the test case data and the expected result. With this, the data is an array of Strings. Each string is comma-separated values (using single quotes if a value contains a comma). The delimiter can be changed too. For example:

```
@ParameterizedTest
@CsvSource({"#Data, ExpectedResult:",
            "1,     true",
            "3,     true",
            "4,     false"})
void someFeatureTest (int testValue, boolean result)
{   assertEquals( isPrime(testValue), result );
}
```

(Note how you can include a comment.) The number of values in each string and their types should match the parameter profile of the test method.

To repeat a given test some number of times (say 25), just use this annotation:

```
@RepeatedTest(25)
```

This is useful with multi-threaded code or tests that depend on external factors that might fail.

Hamcrest and AssertJ

Test code must be short and readable. (Have I said that already?) Lengthy or complex code may have bugs! You never want to have to create tests to verify your tests are correct!!

To help keep test code short and readable, alternatives to using the low-level *assertX* methods that are provided in the JUnit framework exist. (JUnit 5 does include some of these alternatives built-in.) Two popular extensions include Hamcrest and AssertJ. These provide support for the given-when-then style of tests:

```
// Given
 preconditions go here, e.g., create objects used in test
// When
execute the code under test
```

```
// Then
verify the results
```

Hamcrest uses *matchers* with an `assertThat` method, that makes your tests read like English (sort-of):

```
assertThat(result, is(not(equalTo(expected))));
```

AssertJ is a *fluent* API (uses method chaining). Here's what it looks like:

```
assertThat(frodo.getName())
        .startsWith("Fro")
        .endsWith("do")
        .isEqualToIgnoringCase("frodo");
```

(I prefer AssertJ to Hamcrest.)

(See also Cucumber for a framework that allows you to write tests in English!)

A quick example for AssertJ: First, install AssertJ by adding the Jar to your CLASSPATH or by adding the Maven dependency. Then:

```
package com.wpollock.foo;
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;
import static org.assertj.core.api.Assertions.*;

public class FooTests {
  public void testPadWithLegalParams() {
    // Given:
    final int num=512, minWidth=5;
    final String expected = "  512";
    // When:
    final String result=TextKit.pad(num, minWidth);
    // Then:
    assertThat(result).isEqualTo(expected);
  }
}
```

Test Fixtures

You can add "helper" methods to a test suite if you want (or even additional classes). Such methods are handy when you need to do the same steps in several different test cases. For example, in complex situations you may need to create a bunch of objects, create files, setup a database, or all of that. You may need to do this afresh for each test method (so one test doesn't mess up the data for the next test.) The environment that your tests require is called a ***test fixture***.

Besides manually invoking helper methods from within each test case method, JUnit allow you to create test fixtures and work with them easily. Before every test case is run, JUnit 4 will invoke all the methods annotated with **`@Before`** and afterward invoke all methods annotated with **`@After`**. You can have any number of these.

JUnit4 also provides **`@BeforeClass`** and **`@AfterClass`** annotations for a pair of methods that run once per test cycle, to do any one-time set up and tear down (example: initialize a test database, then delete it).

JUnit 5 has similar annotations but with better names: `@BeforeEach`, `@AfterEach`, `@BeforeAll`, and `@AfterAll`.

Mocks and Other Test Doubles

In some cases, you may need to create a number of realistic objects to run your tests on some class. This can happen if you need to unit test your code that should invoke objects of some class, and the developer in charge of that class hasn't written it yet. In other cases, you need to test against public APIs (such as for AWS, Google, Facebook, etc.). Testing against a commercial API can fail from the real service having rate limiting, IP bans, and other anti-abuse mechanisms. Not to mention, using those services costs money.

In general, use *test doubles* for 3rd party APIs your code uses (called ***external dependencies***). Testing against such APIs is best saved for your integration tests, not unit tests. (Just don't forget to test such interactions!)

For example, suppose you want to test an email server (MTA). To test it you might need some MUA (mail client) objects, but for this test you don't want to wait until a fully functional MUA class is completed and tested. You need MUA objects that behave *just realistically enough* so you can test your MTA. Such objects are called ***mock objects*** or simply *mocks*.

As another example, suppose some service registerUser is handled by a different team or a different programmer. You can expect them to assure registerUser behaves how it should by using tests. Since you don't want to test registerUser, just your code that depends on that service, using a test double is the way to go: removing their code from your unit tests makes the errors you get from a test run much easier to read because you're just looking at the logic in one unit.

In practice, the term *mock* is used for a number of related types of objects used in testing. Collectively these are known as ***test doubles***. The different types are:

- Dummy — Dummy objects are passed around but never actually used. Usually they are just used to fill parameter lists as required.

- Stub — A class containing no logic at all. Stubs simply implement the correct interface; the methods ignore their arguments and just return some constant value.
- Spy — A wrapper around a real object, where some of the object's methods are replaced with stubs (and the rest just pass-through to the actual object). Spy objects often have methods that allow you to inspect how various stub methods were invoked. (So not true stubs, they have some code to record when and how they were called.) In you test code, you need to inspect the spy object and have assertions for what you expect.
- Mock — A mock implements the interface of some real object your code depends upon. It is a type of spy, but it has expectations about the way its methods should be called. Using a mock, you can specify which methods should be called, in what order, and how many times. So if your code (that you are testing) should call some method first, and only once, the mock can verify that for you (or fail with some exception). Mocking frameworks can be used to automatically create such mocks from interfaces and a description of the expected method invoking behavior, making them easier to use in general than spies. (*See below.*)
- Fake — A fake doesn't use a mocking framework: it's a lightweight implementation of an API that behaves like the real implementation, but isn't suitable for production (e.g. an in-memory database, which by the way is incredibly useful when testing). Fakes can be used when you can't use a real implementation in your test (e.g. if the real implementation is too slow or it talks over the Internet). You shouldn't need to write your own fakes often since fakes should usually be created and maintained by the person or team that owns the real implementation. (Too often, they don't do that. Do you?)

Test doubles are are rarely used for many types of testing. For example, instead of using a test double for some database interaction, it may be better to spin up an instance of PostgreSQL (or whatever), possibly in a container. Indeed, many people use TestContainers for that. I think using an instance of the actual DB to be used in production is best if your tests will run fast enough.

Test doubles are also rare in end-to-end (e2e) and integration testing.

Rather than create mock objects manually, there are ways to simplify their creation; all you need is the interface for the object, and a mock object that implements it can be created automatically, then used. A popular tool for this used with JUnit is **Mockito**.

Such mocks are useful for testing, in that when you create them you can specify the expected method call sequence on the mock, and later test that your code invoked the correct methods with the correct arguments and in the correct order, the correct number of times. For the purpose of unit testing, you tell the mock object that when method `foo` is invoked, return `bar`.

> To be able to easily use mock objects and "fake" classes for testing often requires extensive use of interfaces. Anticipating testing means that you generally create an interface first, then one (real) class that implements it. In your tests, you can create mock objects instead.
>
> Without an interface you can create spy objects, but that can be more complex (and hence error prone).

A simple demo of JUnit 5 and Mockito: A student enrolls in a course. Here we are testing the `Course.enroll(Student)` method, but class `Student` doesn't exist yet while interface `IStudent` does:.

```
import org.mockito.*;
// Other imports for JUnit go here

@ExtendWith(MockitoExtension.class)
public class CourseTests {

   @Mock
   private IStudent mockStudent;

   public void testEnrollStudentInCourse() {
      // Given:
      Course course = new Course(...);
      // Tell mock what to do when method invoked:
      when(mockStudent.addCourse(course))...;

      // When:
      course.enroll(mockStudent);  // Use mock

      // Then:
      verify...(mockStudent);  // Verify worked
   }
}
```

Notice the `@ExtendWith` annotation, a feature of JUnit 5 to allow 3rd party extensions to work with JUnit. See jmock.org and site.mockito.org for more information.

Most testing frameworks including Mockito have limitations. PowerMock is a framework that extends other mocking libraries with additional capabilities. PowerMock uses a custom classloader and bytecode manipulation to enable

mocking of static methods, constructors, final classes and methods, ~~private methods~~, and other features not available with Mockito alone.

To test a method that produces console output, refactor your code to invoke a method that returns the output to display. If that's not feasible, use something like this:

```
PrintStream savedOutputStream = System.out;
ByteArrayOutputStream baos = new
ByteArrayOutputStream();
PrintStream pw = new PrintStream(baos);
System.setOut(pw);
    // do something
System.setOut(savedOutputStream); pw.flush();
String actualOutput = baos.toString()
```

This sort of thing is ugly, and illustrates a somewhat subtle advantage of test-driven design. There are often small differences in an interface design that make all the difference in how easy it is to test. Test-driven design naturally leads to a preference for designs that can be tested over designs that are harder to test. For example, many output methods could be written to take a `PrintWriter` or `PrintStream` parameter. Methods written that way are easy to test by giving them a `PrintWriter` based on a `StringWriter` or a `PrintStream` based on a `ByteArrayOutputStream`. You can then easily examine what they wrote.

Designing Unit Tests

Each test (method) should be very clear and simple. Do not make the reader struggle with understanding the test. Refactor each test if necessary.

Each test should test a single concept, or aspect of your code. It should do that with as few assertions as possible, preferably just one.

All unit tests should either pass or fail. Avoid tests where a human must examine an external file or log data to see if the test passed.

Avoid interactive code in tests. You cannot (easily) automate tests that require someone to enter data.

Initially you create very few tests, just enough to ensure the methods you're testing compile and run correctly. (Just 1-3 happy path tests are normal.) As you add details to methods (such as error handling), you add more tests. Eventually, you will need to add tests for safety and compliance as well. For critical code, you also add boundary value and other tests.

The performance of tests generally is not an issue; clear and simple tests are the goal. Few things are more annoying than bugs in your test code! That said, some test activities can take hours to weeks, such as fuzz testing and mutation testing.

Running Tests with JUnit

Compile the test suite class and pass its name as an argument to the JUnit test runner:

```
java org.junit.runner.JUnitCore MyTestSuite
```

A simpler way is to add a `main` method to your `MyTestSuite` class that invokes the JUnit 4 runner; then you just use: `java MyAppTestSuite`. Here's how:

```
public static void main ( String [] args )
{   JUnitCore.main( "MyAppTestSuite" );   }
```

(**Show TextKitTestSuite**.) The output will show one dot per successful test case, one 'E' per failure, and one 'I' for ignored tests. Here's a sample output:

```
C:\>java MyAppTestSuite
JUnit version 4.13.2
....I
Time: 0.032
OK (4 tests)
```

When a test fails, JUnit will abort that test case by throwing an exception. This will print a message about the failure and print a stack trace; one line in that will show the line number of your test case that failed. Note the other tests still get run.

JUnit since version 4 no longer includes a GUI runner, but there are stand-alone ones available, GUI ones built into Eclipse and NetBeans, and an `ant` task with XML output. (There's a nice NetBeans IDE plugin "Code Coverage" that highlights code run during testing, so you can see what code was and wasn't tested.)

Addition Notes:

- **IDEs have JUnit 4 & 5 support (2020)**. In Eclipse for example, just right-click the class (or its source file) in the IDE, then select `new...` --> `JUnit Test Case`. A checklist of methods in that class will appear; check the ones you want to generate test cases for. Repeat for each class. To run all the test cases, right-click on the package name, select `Run As...`-->`Junit Test`. You may wish to move the test cases to a single class; Older Eclipse version seems to put each test case into its own class (but all such classes do run). You can run tests via Maven with "`mvn test`"; Gradle is similar.

  > Note Maven runs tests using its "surefire" plugin. This can save the results of tests as XML or as HTML reports. With the proper

configuration, these and other reports show up when you run "`mvn site`". You may need to specify the Surefire version with older Maven to ensure JUnit 5 will work.

- **If not using Maven or Gradle, keep test data files with the test suite classes.** With Maven, such files are in the `src/test/resources/` folder and can be referred to in your Java code as "`/filename`". (It's similar to non-testing code resources, under `src/main/resources/`.) If not using such a build too, keep the test data in the same directory as the test class file and access them with `Class.getResource("filename")`.
- **Avoid side effects in your test cases.** If you must have these, make sure you use `setUp` (`@Before`) and `tearDown` (`@After`) methods to restore the system state for each test case. (JUnit5 uses `@BeforeEach` and `@AfterEach`.) Use `@BeforeClass` (JUnit 5: `@BeforeAll`) to set up expensive resources, such as turning on and setting up database or other servers. (In this case, you need to ensure your test methods leave the servers in the correct initial state for other test methods.)
- **Test to (public) interfaces for unit tests (black-box).** Generally, leave implementation (or "white-box") testing to Java assertions. And **have plenty of assertions**. For code with side effects, you can peek at package-private fields that may have been changed to see if your test worked.
- Some code depends on other code. In those cases, you can **create test doubles (mock objects) that simulate the correct behavior of that other code**, at least well enough for testing purposes. If this isn't easy to do, consider duplicating the production environment for testing. (Include any frameworks and databases.)
- **If you have one class extending another, you should mirror that by having one test suite `extend` another.** All the superclass tests will be run automatically!
- **Don't put your test code in the same directory tree as your application code.** It will be hard to separate out later and you don't want to ship that code (or those `.class` files). When not using an IDE or build tool such as Maven, make sure `CLASSPATH` lists where your code is. Note you can still have classes in the same package even if in different directories.
- **Don't waste effort by testing trivial methods.** Getters and Setters (a.k.a. accessors and mutators) are usually too simple to bother testing. The rule is: don't try to test Java itself. So if the method will work if Java does, then it is too trivial to test.
- **Adding more test cases will eventually reach the point of diminished returns, where further testing is no longer cost effective.** Even knowing

exactly when you've reached this point is unknowable! However, most students and new (and self-taught) developers write far fewer test cases than they should. For critical code, consider using additional testing methods such as fuzz testing (requires lots of assertions in the code) or pairwise or CIT testing.

- **Use white-box unit tests if code coverage metrics are important.**
- **Ensure quality tests with code reviews and/or mutation testing.**
- **Automate Testing as much as possible.** Use scripts, `make`, or other tools such as the `surefire` plugin for Maven.

> The **TestNG testing framework** is a competitor to JUnit. As with JUnit since v4, to define a test method just precede it with the `@Test` annotation. When you later run the TestNG program, giving it the class containing your tests, it finds the test methods using this annotation. Output is HTML that can be viewed in any browser (Eclipse has a plug-in for this). Visit `TestNG.org` for more information.

Coding Style Testing with CheckStyle

Various IDEs have settings that allow you to define a coding style, and to reformat your code according to that style (either as you type, when you save, or manually). Instead of enforcing style, there are plugins such as CheckStyle that report style violations as errors.

CheckStyle can be run from the command line as well as from within an IDE, used with automated testing (a best DevOps practice, and part of *continuous integration*). Here's an example (Try with "`/google_checks.xml`" too):

The code file to check is `Foo.java`:

```
public class Foo {
    public static void main ( String [] args ) {
        double sum = 0;
        for ( double d = 0; d < 10; ) {
            d += 0.1;
            sum += d;
        }
        System.out.println( sum );
    }
}
```

```
C:\Temp>java -jar checkstyle-8.5-all.jar -c sun_checks.xml Foo.java
Starting audit...
[ERROR] C:\Users\wpollock\Temp\Foo.java:0: Missing package-info.java file. [JavadocPackage]
[ERROR] C:\Users\wpollock\Temp\Foo.java:1: Missing a Javadoc comment. [JavadocType]
[ERROR] C:\Users\wpollock\Temp\Foo.java:1:1: Utility classes should not have a public or default constructor. [HideUtilityClassConstructor]
[ERROR] C:\Users\wpollock\Temp\Foo.java:2:5: Missing a Javadoc comment. [JavadocMethod]
[ERROR] C:\Users\wpollock\Temp\Foo.java:2:29: '(' is preceded with whitespace. [MethodParamPad]
[ERROR] C:\Users\wpollock\Temp\Foo.java:2:30: '(' is followed by whitespace. [ParenPad]
[ERROR] C:\Users\wpollock\Temp\Foo.java:2:31: Parameter args should be final. [FinalParameters]
[ERROR] C:\Users\wpollock\Temp\Foo.java:2:37: 'String' is followed by whitespace. [NoWhitespaceAfter]
...  Some output omitted
Audit done.
Checkstyle ends with 15 errors.
```

If using CheckStyle in your IDE, you will likely want to turn off the IDE's own style enforcement; you don't need both, especially if they conflict.

Java Assertions — Implementation Testing

Useful for testing the implementation details (white-box testing), *assertions* were added to Java in version 1.4. You embed the assertion right in your code. The

assertion contains a boolean expression that gets evaluated. If false, an error will be thrown (aborting the execution and showing information about what happened, such as a stack trace). Assertions should be used to check for anomalous conditions that should never happen: precondition failure, postcondition failure, or invariant violations. Assertions are invaluable for fuzz testing as well.

By keeping the test with the code, it becomes easy to change the tests when you change the code. **Assertions are complementary to unit tests**: one tests black-box (against the design), the other tests the implementation details: `private` methods and changes to fields and local variables, and `private` nested classes that are not part of the `public` (or `protected`) interface.

Assertions used this way are "asserting" what you expect of the code. When that is violated, there is a logic error. Stating your code's intentions this way is better than having "what" comments (what the code is supposed to do) and leaving correctness up to code reviewers. With assertions, your logic errors often show when testing.

Assertions should **not** be used when the design calls on a method to work a certain way with certain inputs. For example, if a `public` method takes an `int num` as an argument that should be greater than zero, that's a *precondition*. If you chose to test that (as opposed to merely stating that in a doc comment and assuming callers of your method never make mistakes), you should use an `if` statement and `throw` an `IllegalArgumentException` if `num` is negative or zero. (And unit tests should confirm this behavior, not assertions.) If you ignore the issue, your code has ***undefined behavior*** which is a very bad thing.

> Preconditions, postconditions, and invariants are discussed with design, later in the course.

However, for a `private` method you don't need to worry about this; there's no specification requiring you to test preconditions or even state them in a comment. You should still test the precondition, but it should not be tested every time in the production environment; just when testing. Assertions are perfect for this:

```
assert num > 0;
```

The beauty of assertions is that they can be turned on or off at runtime. If turned off there is (almost) no runtime overhead for them! **You can selectively enable assertions** too, to limit them to small parts of your application that you are currently testing. Assertions are generally enabled when running tests and off by default (for production).

An assertion looks like this:

```
assert booleanExpressionThatShouldBeTrue;
```

If assertions are enabled (as when testing), if the expression is false then an **`AssertError`** is thrown (`Errors` are not `Exceptions` but they are `Throwables`), containing information about where the problem was located. You are not expected to use a try block for these.

You can also add a second argument to `assert`:

```
assert booleanExpression : expression2;
```

Which will evaluate `expression2` and pass it to the `AssertError` constructor. This will in turn convert it into a `String` (it usually is a String literal) and display it when the error is thrown. Expression2 is often an informational `String` concatenated with the values of various variables (or method return values) that would aid in debugging.

> **Never invoke methods with side effects in assertions!** (Methods with side effects should be avoided in any case.) Consider:
>
> ```
> assert methodWithSideEffect() > 0 : "Whoops";
> ```
>
> Such programs behave differently when assertions are on versus off.

One mistake to avoid is to use assertions when you should be using `if` statements and throwing exceptions (e.g., checking parameters to `public` methods, a.k.a. *preconditions* in a `public` method). **When something is part of the contract (the specifications) and should then be part of the implementation, don't use assertions.** Remember that these can be turned off (and usually are). That very likely will break your software.

On the other hand, any comments that state "should never get here" or "At this point we know `num` must be zero", can and should be converted into assertions. **Use assertions for proving pre- and post- conditions, and invariants** (i.e., accounts balance) if within a private method or otherwise is not part of the specifications for your code. Also use assertions for `switch` statements with no `default` case. (See assertion usage notes and/or programming with assertions on Oracle's website for more information on using assertions.) For example, if one method invokes another that should not return null, you can check this postcondition in the calling method (or just before the return of the called method if convenient) with an assertion.

> The checker framework can do this for you automatically, by using its annotations such as `@NotNull`.

How many assertions should be used? As many as are called for. Remember that in production, assertions are turned off, so there's (almost) no penalty to having lots of them! In practice, it is common to have a couple of assertions in every non-trivial method.

Assertions are disabled by default. To enable them requires the use of the "`-ea`" (or the longer "`-enableassertions`") command line switch to the "`java`" command. **There is also a "`-da`" switch to disable assertions. These can be combined to selectively enable assertions just where you want:**

`-ea`	*enable all assertions (except for JRE classes)*
`-ea:`*`package`*	*enable in just this package*
`-ea:`*`package`*`...`	*enable in just this package and any sub packages*
`-ea:...`	*enable in the default nameless package*
`-ea:`*`class`*	*enable in just this class*

You can repeat each of these as needed. For example:

```
java -ea:mypkg... -da:mypkg.Foo mypkg.MyApp
```

To enable assertions for system (JRE) classes and packages, use the "`-esa`" and "`-dsa`" switches, which take no arguments. There is rarely if ever any need to do this.

Posted in `comp.lang.java.programmer` on 11/24/07 by Stefan Ram (with comment by Patricia Shanahan: "I was just writing a small and simple program to calculate a random number from the set {0,1}:

```
import java.security.SecureRandom;
...
final SecureRandom rnd = new SecureRandom();
final int r = rnd.nextInt();
final int result = ( r / 69313 )% 2;
```

"Just for the heck of it, I added:

```
assert result >= 0 && result < 2;
```

"As I wrote this, I thought: 'This assertion is completely unnecessary, because it is obvious that the result of »% 2« is always in the set {0,1}. So what am I doing here? Just stating the obvious to needlessly enlarge the source code?' The next thing happening then, of course, was:

```
Exception in thread "main" java.lang.AssertionError
```

"[PS:] The moral, perhaps, is that the more deeply you are assuming something, the more valuable it is to assert it. For those not used to the '`assert`' keyword yet, the key concept is "invariants". The presumed invariant wanted in Stefan's example was "that the result ... is always in the set {0,1}"; it just happened that ... »`% 2`« was not enough to ensure it. `Math.abs()` would have helped ensure it [since `nextInt` can return

negative numbers -WP]. This is an excellent example of how assertions help make a program bug free."

Enterprise Application Testing

Unit testing EJBs, web services, and other enterprise application code is harder than simple unit testing, in part because such code depends on services provided by the application server (EJB container). Entity beans (also called *data transfer objects)* don't usually contain much logic (they just represent persistent data) so unit testing them is usually sufficient. However, session EJBs (or some equivalent such as Spring) contain your application logic and are managed by the container, which instantiates them as needed, passes them around, invokes their methods, hooks them up to database connection pools and other resources, etc. (An analogy is testing a complex AWT or swing application without using a JVM that provides the event handling.)

One way is to test using only remote interfaces. You set up the code on a test server and run it and then the tests connect to the server as a client would. Such testing can be quite through for EJBs that use RMI and thus tend to have extensive remote interfaces.

When the web server (presentation tier) runs on the same physical host as the application server (business logic tier), the two usually use *local* interfaces to communicate rather than remote interfaces. (i.e., simple method calls or communications via *sockets*.) In this case you may need to deploy your tests as another application in the same application server, and have the test EJBs invoke the application's EJBs. This is much harder to setup and control, and the testing code must be removed before deployment.

If the application is simple it may be possible to create "stubs" for the application server objects. The app talks to your stubs rather than a real application server. (See Jakarta EE Unit testing blog post from Oracle.)

Instead of (or in addition to) unit testing enterprise applications, developers generally perform integration, functional and acceptance (QoS) testing. A popular tool for this is Arquillian.

Testing the EIS-tier (database interaction) usually requires access to a test database with sufficient data to appear realistic. (A *sanitized* copy of a production database would be ideal.) This approach still requires you to create a set of SQL scripts to restore the DB in an initial state for each run of your test suite, and a set of queries to examine the state of the DB after the code executes to see if the test succeeds or fails. DB code is central to many applications and can be complex: caching, transactions, and rollbacks in the event of errors (or terminated sessions) all must be tested.

Testing the Web-tier is difficult too; you must deal with some browser/user unique issues: re-submission, submission from multiple windows, timeouts, implications of the back button, browser compatibility, caching proxy servers, and resistance to denial-of-service attacks are all important elements. JSP pages have additional difficulties, as they don't even exist (the java code in them is collected and compiled into servlets by the server when invoked). Cactus can be used for this testing too, as well as for acceptance testing of the web tier.

Additional tools that can help with UI testing are Marathon, used to test AWT or Swing GUIs, and Selenium, used to automate web browsers. (Both are open source.)

> Modern (since 2016) web browsers are no longer easily hacked to support such testing tools. Today you need to set up a *proxy* to intercept and measure all requests and server responses to your load testing tool. This can be difficult to set up. A simpler tool for just performance testing is Gatling.

Marathon runs GUI based acceptance tests against Java Swing applications. It provides an integrated environment for test script creation and execution. Currently, Marathon supports Jython and JRuby script models for recording the test scripts. You can download it from sourceforge.net/projects/marathonman.

Selenium is used to automate web browsers from a script. While this can be used for many things, mostly it is useful to test web applications. Selenium can be integrated with JUnit as well. Selenium can query the state of anything in the result HTML page's DOM or JavaScript environment. It also has the ability to inject data from a database into the test session, simulate clicking on buttons, and programmatically confirming that the generated HTML matched other the data from the database.

Additional GUI testing tools include Jubula (which is integrated into Eclipse nicely) and Google's WindowTester Pro.

To load and stress test web interfaces, about the easiest tool is Microsoft's free Web Capacity Analysis Tool (WCAT, also from here), formerly called the *Web Application Stress Tool*. Other good tools are *slamd* from `www.slamd.com` and Apache JMeter. Selenium could be used as well.

Much real-world software will require performance guarantees. To see if your code meets performance requirements, you need to do **performance testing**, also known as benchmark testing. This is used for any code, not just the user interface. There are a number of tools you can use that will run your software a large number of times (including a warm-up set of runs to take into account caching and JIT effects) then show statistics about the runs. Once such tool (there are many) is Hyperfine, which runs from the command line on most platforms, and is open source and free. JUnit 5 also has benchmark test support.

Benchmarks generally cannot tell you where in your code the time is being used, although JUnit does support benchmark tests of units (as in unit tests). For more details, you need to *profile* the code and/or use *tracing tools*; these are discussed below.

Performance Flame Graphs

Most CPU performance tools today show output as a flame graph. As your code runs, every few nanoseconds or milliseconds an OS-specific tool will collect data from various counters the hardware provides, and also snapshots the call stack. This data can be used to report how much CPU time is spent by each of your methods. However, reading that data is too hard for humans.

In a flame graph, each method is shown as a horizontal bar. The lower down bars are the caller methods of the bars stacked on top. The width of a bar shows how much time that method (or some method it called) spend running on the CPU. For multi-threading, each stack is shown; the left-to-right ordering is arbitrary. The color of the boxes is generally arbitrary, although some modern flame graph software allows you to assign some meaning to the colors.

The performance bottlenecks easily stand out on such a graph. Wide boxes are the "hot" methods (hence the name *flame graph*). Note you cannot tell from flame graphs (or the performance data) whether a method with a wide box is taking a long time to run or if it is called often. (See a sample SVG flame graph from acm.org.) Flame graphs can be used for other purposes with some effort, such as memory use, storage I/O, or network I/O.

Valgrind

One last tool worth mentioning is **Valgrind memcheck**. Valgrind is a simulator, a kind of virtual machine for an x86 CPU running a Linux binary. As Valgrind runs an application, it looks for all kinds of interesting errors such as array overruns, reading from uninitialized memory, stack overflows, memory leaks, and so forth. Valgrind finds problems that can slip through other tests. And, when Valgrind does find an error, it can dump the developer directly into a symbolic debugger at the exact point where the error occurs. Note Valgrind cannot be used for Java code, but you should know about this tool regardless.

A comment about the search engine project:

If you have had difficulties editing each other's code, now you know why. Y**ou should first implement the public interfaces with comments and with full group agreement**. Then and only then, do you each work on your portion of the code. Other group members never need your code, just the agreed interface, for their testing of their code. (They can use testing doubles.) If you decide to change the method signatures then you need to update the interface and have code review for the change.

Also, **it pays to use the MVC design pattern (discussed later) for your user interface code when you can**. For this project, the various (two) user interfaces are the *views*. The list or table containing the files added, and the search results, are the data used by the views and are called *models*. The code that reacts to typing and mouse clicks are *controllers*.

Most everything else is the business logic layer: the actual searching in this case. Even this part can be split into two layers, the persistence (long term data storage) layer and the *business logic* layer. (Simple applications with very little business logic just include that in the controllers, and thus don't have a business logic layer.) For you search engine, the persistence layer just needs a class with a couple of methods: one or two methods to initialize the file list and the inverted index (the map) from a file (or create new ones if there is no valid saved data) and return them, and one to save the in-memory list and map to a file.

Such a design is easier to work on, in groups. There is very little coupling between the different parts so everyone can work on their part of the project independently. In addition, you can change parts of the application easily, say to use XML files or a database, or to change a stand-alone application to a Java-EE web-based one (only the UI layer is changed), or to change the implementation of (say) the PHRASE search method. Such low coupling also makes testing far easier.

Other Uses for Test Frameworks

Test cases have some interesting additional uses. One is to help design your API. If you know only vaguely what you want some code to do, writing test cases forces you to design the method signatures and expected behaviors. If you get to a point where it becomes difficult to write such tests, you can easily redesign the API at this stage; you haven't implemented anything yet so changing your mind at this point is cheap!

Unit tests can be used to document requirements. Human language is vague and often ambiguous and confusing (which is one reason why we need lawyers). Naturally, requirements documents often end up ambiguous and confusing. Unit tests on the other hand are a precise statement of the expectations of some code.

Low-level code can be difficult to experiment with, simply because it requires calling code. A test framework can be used as a *driver*, the code that invokes the low-level code you want to write.

Tests can be a great way to learn somebody else's code. You just write simple tests to invoke their code correctly. Your other choice is to write whole programs to test that you can use their code correctly. Once all your tests pass, keep them!

When new versions of their code are released, simply rerun your test to see if anything need changes.

Test-Driven Development (TDD)

When using TDD, you design your software by writing tests using your test framework (such as JUnit). You are designing the API details by trying to use the code (that you have not yet written). Trying to use code before finalizing the API generally leads to better and easier-to-use APIs. It's also faster to try to invoke method calls, then change your mind and invent new method calls, than it is to implement code and later discover you have a terrible API.

When doing this, you often create a few (often just 1-3) happy-path tests at first. This test code won't compile as the classes and methods you just invented don't exist yet. Next, create a skeleton with stubs so your tests can compile and run (but fail). Then spend time make the tests pass by implementing the methods.

As you get one or a few tests to pass, reflect on (think about) the API and code, and see if any changes would make better API or implementation, or are needed for other reasons. Additional tests (boundary-value and/or failing case tests) should be added as needed. This will depend on how critical the code is and how acceptable failure is. So if you create a project with 3 classes and each has 5 methods, you'll end up with at around 20 to 45 test cases, sometimes more.

TDD works well for designing reusable code to be used by other code, such as for a library. It is less suited for designing stand-along GUI programs and is not used for that. TDD is discussed later in the course (along with the whole topic of *design*).

Lecture 13 —Logging and Tracing

Logging [*partly adapted from Apache.org/log4j documentation*]

Inserting log (print) statements into your code is a low-tech method for debugging it. (Discussed in Java I course as *scaffolding*.) However, this is often the best way to debug enterprise (distributed) applications. Adding `printf` statements at various points in your code allows the developer to see the *state* of an object as it changes, to catch any unexpected values when they occur. But there are problems with this approach. One is, do you really expect any system administrator to be watching a console window for any log messages that may scroll by, 24X7? Another is remembering to remove all such statements before production code is committed. (But if you need to debug in the future, you will need to put them all back!)

Any added statements increase the size of the code and may reduce its speed, even when logging is turned off. A preprocessor such as used with C++ can be used to strip out debugging/logging statements when building for production. However, any changes can affect the program, especially a multi-threaded one running on multi-core systems. So just turning logging on (inserting the `println` statements) or off (removing them) can cause problems to manifest (or fail to). In Java, a preprocessor is not generally available anyway. But in Java, the logging statements only have a very small, fixed overhead.

Deciding where to insert log statements into the application code requires some planning. **In some applications, about 4 percent of the code is dedicated to logging!** A medium or large sized application may contain hundreds or thousands of log statements.

Some logging (of real-world software) is almost always required: all login/logout attempts, successful or not, all access to sensitive data, etc.

Some applications are *instrumented* to collect metric (performance) data. In such applications, the metrics can indicate many issues and problems you would otherwise use logs for, and thus less logging (especially logging for debugging) is needed.

> Logging is related to ***tracing***. The difference is that tracing is often at a lower level and often doesn't require statements to be added to the code. (For example, the *Java Flight* Recorder (JFR) doesn't require any additional or changes to your code to use.)
>
> Tracing output has more detail, can be used to trace hardware, firmware, kernel, or application code and events, but will generate much more data. For example, if some application fails due to some permission error, but the developer didn't include enough information to pinpoint the problem, you

can trace the application to see what file(s) the application tries to access and where that fails. (*Show trace output from class resources; point out the trace output can be filtered to reduce the clutter.*)

Related is *distributed tracing* (or *request tracing*), useful for following the processing of some request as it passes through various micro-services.

In addition to tracing, you can instrument to code to collect metrics (discussed after logging), which will include the time spent in each code block/method and how often a given block of code is executed. This is known as *profiling*. Like logging, the developer needs to add statements to the code at places of interest. Most enterprise software will include logs and metrics reporting. Some frameworks for this include Micrometer and Dropwizard Metrics.

A popular logging tool used today is **Apache.org's log4j**, however Java now includes logging classes in Java SE. With either logger, it is possible to enable and disable logging at runtime without modifying the application binary. Java logging is designed so that log-generating statements can remain in shipped code, without incurring a performance cost (if logging is turned off). Logging behavior can be controlled by editing a configuration file or other means. (This is similar to how assertions can be enabled/disabled.)

No matter which logging framework you use, the use of a logging framework by the programmer is the same: You create a logger object and use its methods to generate log messages. Loggers have methods to control which log statement to ignore, where to send the output, and the format of log messages (e.g., plain text, syslog format, XML).

Logging statements are put into the code at key points (any spot in your code at which you *might* want to generate a message). Each log statement is assigned a ***log level***: one of TRACE, DEBUG, INFO, WARN, ERROR, or FATAL. (Note: the available log level names depend on which logging framework you use, but they are all similar.)

Logging output is controlled by selecting a log level and a part of the application: A whole package or just part of a class. If you set the logger's level to, say "INFO", then only log statements of level INFO or higher will produce any log messages; other log statements won't do anything.

The raw log information needs to include a timestamp, the location in the code where the log message was generated, and the log message itself. Most logging frameworks will add these to all log messages automatically. When you configure logging, you must specify the log output format as well as a destination for the log messages (not as easy on a Jakarta EE application running on multiple hosts!)

Logging provides developers with details about application failures. On the other hand, testing provides quality assurance and confidence in the application. **Logging and testing should not be confused**. They are complementary.

> Log statements can be sent to the console. Accordingly, using logging instead of `println` scaffolding for debugging is useful even for small projects.

Enterprise systems should log all security related events (whether successful or not), all errors, all access to sensitive data, all modifications or deletions of any data (including application configuration files), and some performance data, such as entry/exit to key (or all) methods. Also log any unavailability of external services, timeouts, etc.

Remember, if also using metrics then you don't need log statements to measure performance. For example, log statements at every method entry/exit may not be needed if you have metric statements there already. (Even so, you might add those log statements anyway, as you get useful debugging messages without needing to correlate logs and metrics.)

> From James Turnbull, The art of Monitoring:
>
> - Measure and log requests and responses, such as to specific web pages or API endpoints. If you're instrumenting an existing application then make a priority-driven list of specific pages or endpoints and instrument them in order of importance.
> - Measure and log all calls to external services and APIs, such as if your application uses a database, cache, or search service, or if it uses third-party services like a payments gateway.
> - Measure and log job scheduling, execution, and other periodic events like cron jobs (scheduled tasks).
> - Measure and log significant business and functional events, such as users being created or transactions like payments and sales.
> - Measure and log methods and functions that read and write from databases and caches.

Measure resources such as memory or storage space, and output logs (of warning level) when running low. You can also log slow response times (warning), SLA violations (error), database failures (warn or error), application failure (fatal), etc.

Make sure the logs include all information a developer might find useful, such as the current user/account, the source IP address of the client, the values of parameters passed in, etc.

Add any additional logging statements that you think will help troubleshoot a failing system; you can set those at level debug and normally they won't be used.

Don't duplicate log data, such as your web server's access logs.

> Log messages should not be visible to end users as they may contain sensitive information. That said, it is usually a bad idea include passwords in log messages! The error message the user sees need only report the time of the error and a general message; with that timestamp, a developer working on the issue can find and view the appropriate log messages.

Make logs easy to parse by tools such as `grep` (that use regular expressions) and by eye. (Sometimes, a special database is used to hold logs that permits easy querying.)

Use common fields with a common format at the beginning of each log message; this makes using log data easier. These fields should include the date and time (and time zone if not using UTC), severity (*log level*), application name, subsystem and source code line (or at least package, class, and method). Remember, nearly all logging frameworks will add these fields automatically.

Make sure you normalize and sanitize log data by removing sensitive data (such as passwords) first; if that data is needed in the log, make sure your log files have sufficient access controls so that only authorized users can view the log data. (Normalization, validation, and sanitation of untrusted data is discussed below.)

Consider digitally signing log files, or even individual log entries. Doing so can preserve the evidentiary value of the logs in the event that legal action might possibly be taken. In some cases, you can partially obscure the data to make it safe; an example is showing only the first and last few digits of a credit card number in a log, rather than the whole number, or replace some data with a hash of it.

Logs often contain user supplied or other untrusted information in the message, so make sure you encode the log data using UTF-8 or just ASCII, and **encode (or strip out) newlines or other weird characters**. For example, replace newlines with "\n". Also make sure log data doesn't exceed your system's line length limits.

Because of the large volume of data that may be produced, logs should have controls to turn on or off logging by message type, severity, and subsystem. Remember that security logs will likely be needed for years, but trouble-shooting and performance logs may only be needed for hours or days.

(There's lots more to know about logging, such as log file rotation, retention, and aggregation. These are usually considered *operational* issues rather than *developer* issues, but current (since 2014) thinking is that both developers and system

administrators working on web or network-based applications or services need to work together and understand both sides. This approach is called ***devops***.)

> Be cautious when log messages include data from external sources. Users can create fake log entries that way. For example:
>
> Jun 5 22:29:27 YborStudent sshd[21183]: session opened for user auser
> Jun 5 22:29:29 YborStudent [21184]: user auser attempted to access file **foo**
> **Jun 5 22:29:31 YborStudent sshd[21195]: session opened for user fake**
> Jun 5 22:30:14 YborStudent sshd[21183]: session closed for user auser
>
> Where the **bold text** indicates the filename used by auser. To prevent this, you should normalize and sanitize any external data before using it in a log message.

Using SLF4J

There are a number of popular logging frameworks in use, such as Log4J and java.util.logging, both discussed below. The problem is that while all provide the same functionality (and even use the same concepts), the details are slightly different. Just enough so that using multiple logging frameworks is a problem. A popular solution today is to use the Simple Logging Facade for Java, SLF4J. This is not another framework for logging; rather, it is a good use of the facade design pattern.

SLF4J provides a simple API you use in your code. If your code was developed using some particular framework, SLF4J include a migration tool to modify logging statements in your code. The SLF4J API can be connected to most logging frameworks, at compile time, by using an appropriate *binding* jar. See sf4j.org for more info, and Maven Central for downloads.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Foo {
  public static void main(String[] args) {
    Logger logger =
      LoggerFactory.getLogger(Foo.class);
    ...
    logger.info( "Entered bar, num = {}.", num );
```

As you can see, SLF4J is easy to use. To bind SLF4J to a particular framework, just add the appropriate Jar file (in addition to slf4j-api-*version*.jar), such as slf4j-log4j13-*version*.jar or slf4j-simple-*version*.jar. (Note the dependency is on the binding jar only; the SLF4J-api-*version*.jar will be added automatically.)

For Jakarta EE applications, the Spring framework includes SLF4J and has defaults for it already! You just need to use the logger object it creates for you, although it is common to eventually change the defaults.

Using Log4J

Although Apach log4j is common, Java includes logging built-in. **See `java.util.logging` API.**

The log4j environment can be configured programmatically. However, it is more flexible (and common) to configure log4j using configuration files. Currently (version 1.2) configuration files can be written in either XML or in Java properties (key=value) format.

Here is a trivial but complete example, stolen from the log4j docs (download the log4j Jar file, and put it in on CLASSPATH):

```
import org.apache.log4j.*;
public class MyApp {
 // Define a static logger var that references the
 // Logger instance named "MyApp":
 static Logger logger = Logger.getLogger(MyApp.class);
 public static void main ( String [] args ) {
  // Use a simple conf. that logs on the console:
  BasicConfigurator.configure();
  logger.info("Entering application.");
  Bar bar = new Bar();
  bar.doIt();
  logger.info("Exiting application.");
}}
import org.apache.log4j.*;
class Bar {
  static Logger logger = Logger.getLogger(Bar.class);
   public void doIt() {
     logger.debug("Did it again!");
}  }
```

The output (on the console) will look like this:

```
  0  [main] INFO MyApp - Entering application.
  36 [main] DEBUG Bar - Did it again!
  51 [main] INFO MyApp - Exiting application.
```

(The number is elapsed milliseconds since the logger was initialized. This is only the default layout; you can include lots of different information, in any format you can imagine — or that is required by your project requirements).

Using java.util.logging

After log4j became popular, Sun added logging facilities to Java 4. There is a cool companion project with log4j called Apache *chainsaw*, that has a GUI log viewer tool that works with both Java logs and log4j logs, and even shows live events.

The concepts of logging are the same for Java's logging as for log4j, but the exact details differ. (*Show the logger demo Web resource*.) One difference is, with Java's logging, you create a hierarchy of loggers. You can have a *root* logger for the whole application, children loggers for each package, grand-children loggers for each class, and even lower level loggers for parts of a class. The logger names can be anything, but using the fully-qualified class name (for example, "`Foo.class.getName()`") is common and provides a natural hierarchy. By changing the configuration settings, you can enable/disable loggers without changing the code.

Each logger has a log level set on it. Any messages with a lower level are simply and quickly ignored. Otherwise, the logger creates a log record and sends that to any output handlers defined for that logger. It also sends the record to the parent logger, which works the same. It is the handler that produces the output. By default, the root logger uses a console handler, and loggers you create have no handlers. (So by default, your log messages travel from your code, to your logger, to its parent logger, ..., to the root logger, and then to the console handler.) There are a few standard handlers, and you can create new ones.

Note, every logger and handler has its own log level. By default, the root logger's level is INFO. If you don't set the level of your own loggers, it defaults to "null", which means it uses the same log level as its parent. For a log message to appear, every logger and handler it passes through must have a level set that would permit the message. It is common to only have a handler set on the root logger.

If setting the log level at any point isn't enough control for you, every logger and handler can have one or more filters attached. Each filter takes a log record and returns a boolean to indicate if it should be dropped.

Here's a trivial example:

```
import java.util.logging.*;

class MyApp {
    // Define a static logger var that references the
    // Logger instance that sends logs to the console:
    static Logger logger = Logger.getGlobal();

    public static void main ( String [] args ) {
        logger.setLevel( Level.INFO );
        logger.info( "Entering main" );
        logger.info("Creating a Bar object.");
```

```
        Bar bar = new Bar();
        bar.doIt();
        logger.info( "Exiting main" );
    }
}

class Bar {
    static Logger logger = Logger.getGlobal();

    public void doIt() {
        logger.info( "Entering doIt" );
        logger.finest("Did it again!");
        logger.info( "Exiting doIt" );
    }
}
```

The output (on the console) will look like this:

```
Dec 03, 2015 9:47:53 PM MyApp main
INFO: Entering main

Dec 03, 2015 9:47:53 PM MyApp main
INFO: Creating a Bar object.

Dec 03, 2015 9:47:53 PM Bar doIt
INFO: Entering doIt

Dec 03, 2015 9:47:53 PM Bar doIt
INFO: Exiting doIt

Dec 03, 2015 9:47:53 PM MyApp main
INFO: Exiting main
```

A more realistic example — loging the time it takes for a database action:

```
    startTime = System.nanoTime();
    ps        = conn.prepareStatement(insertLine);
    ps.setString(1, className);
    ps.setLong(  2, pid);
    ps.setLong(  3, free);
    ps.setLong(  4, rt.maxMemory() / divideValue);
    ps.setLong(  5, total);
    ps.executeUpdate();
    ps.close();
    stopTime = System.nanoTime();
    logDatabaseAction("preparedStatement
saveMemoryLineNoGC", stopTime - startTime);
```

And logDatabaseAction is:

```
private static void logDatabaseAction(
         String message, long duration) {
   logMessage(String.format(
      "%-40s took %e milliseconds", message,
       duration * 1E-6));
 }
```

I like to have the times aligned under each other. When the message is longer, I just change the -40. To have a better (in my opinion) display of the times, probably I should change `%e` to `%.2f`.

The output:

```
14:55:47: preparedStatement freeMemoryThread  took 1.366532e+02 milliseconds
14:56:48: preparedStatement saveMemoryLine    took 1.603480e+02 milliseconds
```

Metrics

Metrics are just numbers with a timestamp, a name, and sometimes other data (one or more "tags"). They work in your code similarly to log statements: create a metrics object, configure it, then invoke its methods at various places in your code to collect and output a metric *event*. Metric events are usually stored in a *time-series* database that is well-suited for this. Each event/metric emitted by your application should have a format such as:

location.application.environment.subsystem.function.actions

For example, on the *Production-A* server, the application *Foo*, the *development* environment, the payment collection subsystem, and the type of data (job count):

productiona.foo.development.payments.collection.job.count

(Simpler events can use shortened names.)

Suppose you think there's a performance problem. What do you do about it? A lot if you have instrumented your code to include metrics collection. (If not, some performance data can be calculated from the timestamps in log messages.) A good method of trouble-shooting such problems was proposed by Brendan Gregg, called the USE method. (He also wrote the excellent book, System Performance.)

Although I consider it a tracing tool, the Java Flight Recorder can be used to collect some metrics as well.

(The section on Service Management & Monitoring later has more about metrics, including sample code.)

> ***Open Telemetry*** is a vendor-neutral open-source observability framework for instrumenting, generating, collecting, and exporting telemetry data such as traces, metrics, and logs. As an industry standard it is natively supported by a number of vendors and quite popular (2023). See OpenTelemetry.io for more details. However, ease of use seems to be an issue and proprietary

systems continue to be popular (2025). Some of these systems can enable open telemetry internally.

Here is a demo of using both log and metrics (see full demo on wpollock.com):

```
package com.wpollock;
import com.codahale.metrics.*;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import java.util.Random;
import java.util.concurrent.TimeUnit;

/**
 * Demo of Logging and Metrics collection.
 * This is a trivial but complete example of using
 * SLF4J for logging, and dropwizard.io Metrics
 * package.
 * Logging the output goes to the console.  The console
 * logger is called "simple" and can be configured
 * in several ways; here I use the
 * simplelogger.properties file.
 * For metrics, we use a single "meter" type metric
 * and it too is reported to the console (so non-GUI).
 * You can also use Graphite GUI console by changed
 * the reporter.
 *
 * @author Wayne Pollock
 */
public class LogMetricDemo {
  final static MetricRegistry metrics
    = new MetricRegistry();
  final static Meter requests
    = metrics.meter("requests");
  final static Logger logger
    = LoggerFactory.getLogger(LogMetricDemo.class);

  public static void main( String[] args )  {
    logger.info("Application Starting");

  // Configure metrics output to send non-GUI
  // report to console:
  ConsoleReporter reporter =
    ConsoleReporter.forRegistry(metrics)
      .convertRatesTo(TimeUnit.SECONDS)
```

```
        .convertDurationsTo(TimeUnit.MILLISECONDS)
        .build();
    // produce report every 3s:
    reporter.start(3, TimeUnit.SECONDS);
    for (int i = 0; i < 10; ++i) {
      try {
        handleRequest(i);
      } catch (InterruptedException e) {
        return;
      }
    }
    reporter.close();
}
    /**
     * Fake method that "handles" requests to demonstrate
     * logging and metrics.
     * @param i the iteration number
     */
    static void handleRequest(int i)
       throws InterruptedException {
      logger.trace("Entered handleRequest(" + i + ")");
      assert( i >= 0 && i < 10,
        "i value out of range: " + i);
      requests.mark();
      // ... handle request here ...
      Random random = new Random();
      int delayInMillis = random.nextInt(1000);
      // "Handle" request by sleeping for 0..1 second:
      Thread.sleep(delayInMillis);
      logger.trace("Exited handleRequest(" + i + ")");
    }
}
```

As you can see, a lot of the code is devoted to logging, metrics, and assertions. (And testing, not shown here.) This may seem silly, but keep in mind this is a trivial application so the overhead is a much greater percentage of the code than normal.

Lecture 14 — Internationalization (I18N) and Localization (L10N) (*Skip to Safe Text Handling)*

Qu: Why internationalize? Aspects of language are sometimes determined by geographical *region* such as: language, time data (including format, time zone, and month and day names), units of measure for quantities, and numbers (currencies, percentages, and plain old numbers). Java provides *some* support for this, but not for unit conversion, e.g., gallons to liters. Java also provides limited support to convert between computer platforms (directory path separator character). Your code should do different things depending on the region settings, so you need some way to detect a region, set (a default) region, and store information about different regions. In Java (as in most languages and OS's), regions, languages, and other related information is kept in `Locale` objects. (Historically, not time zones however.) Many standard Java classes and methods are already aware of locales.

I18N is also related to security and proper functioning of Java programs. You need to know that data, such as a user name, might be entered in one or another encoding. All such external data needs to be normalized and validated (and possibly sanitized) before use, but this is complex and often overlooked. See the current news for stories of web or other applications that were broken this way!

For example, **a failure to validate was the cause of the Hearbleed bug** that undermined the security of the Internet and most servers and network devices:

"Dr Seggelmann, of Münster in Germany, said the bug which introduced the flaw was 'unfortunately' missed by him and a reviewer when it was introduced into the open source OpenSSL encryption protocol over two years ago.

'I was working on improving OpenSSL and submitted numerous bug fixes and added new features,' he said.

'In one of the new features, unfortunately, I missed validating a variable containing a length.'

After he submitted the code, a reviewer 'apparently also didn't notice the missing validation', Dr Seggelmann said, 'so the error made its way from the development branch into the released version.' Logs show that reviewer was Dr Stephen Henson.

Dr Seggelmann said the error he introduced was 'quite trivial', but acknowledged that its impact was 'severe'."

— *From an Interview with Dr. Segglemann reported in the Sydney Morning Herald.*

Related to locales is the language used for the text that is displayed by your code. If the current locale indicates the user speaks English, you want to display all text (window titles, button captions, menu items, etc.) in English. But if the locale indicates the user expects Spanish or some other language, then all your text should be displayed in that language. Of course, not all languages will be supported by any given application! It takes time and money to obtain a high-quality translation, so you may use a single language and just format dates, numbers, etc., according to the locale. Or you may provide only one or two languages.

The basic idea is to separate the application logic from the strings, messages, and numbers that are locale sensitive. The strings etc. are kept in a *resource bundle* object, and there is one such object for each locale the application supports. Numbers and dates don't need translation into different languages but must still be formatted according to the locale in use. The first step is representing text.

Text Concepts: Encodings and Character Sets and Grapheme Clusters, Oh My!

> TL;DR version:
>
> Best advice for Java in my opinion is to store characters (actually, grapheme clusters) not as a `char`, not as a code point (`int`), but only as a `String`. **Just use `String`!** Java can split `String`s into grapheme clusters so there's no reason to worry about code points or `char`s at all.
>
> Always set a *locale* before processing text. While Java since version 1.1 supports grapheme clusters, handling has become much simpler since Java 9. Java's regular expressions can be used for this. If you have need for low-level code point handling, use the latest ICU library for anything difficult to do with Java.

Q: What is entered when the "A" key is hit on the keyboard? Ans: a *number*.

> When you hit a key, a number is sent that identifies the key, such as the "A" or the "right shift key". This number depends on the type and vendor of the keyboard you have. A device driver in the OS converts this *scancode* into a (more or less) standard number called a *keycode*. If a sequence of keycodes (usually just one) corresponds to some character (not PgUp, ALT, etc.), then the keycode sequence can be converted into a number representing that character, a Unicode code point (or a sequence of code points). (This is not a Java issue, it is an operating system issue. Java only receives keycodes from the OS.)

Every letter and symbol is assigned a unique number according to some *scheme*. This is called an *encoding*.

A ***character*** is just an abstract minimal unit of text. It doesn't have a fixed shape (that would be a ***glyph***), and it doesn't have a numeric value. The English letter "A" (uppercase letter A) is a character. A character is technically known as a *grapheme cluster*.

A ***character set*** ("charset" or rarely, charmap) is a collection of characters. (A ***font*** is a collection of glyphs, each glyph representing some character. But that's another story.) Usually a related set of characters comes from a particular (human) writing system, called a *script*.

To represent the characters in the character set, each is assigned a number called its ***code point***. This assignment of numbers to characters is often referred to as an ***encoding***. Originally there were two such encodings used in the U.S.: EBCDIC and ASCII. These two only covered the English script (and without accents).

> A character set with each character assigned a code point is a *coded character set*. This terminology is too precise for most of us; all character sets in use today are technically coded character sets. The term only comes up in the Unicode official documentation.

The reality is more complex since some written languages (or *scripts* or *character sets*) are not based on letters, but characters that represent syllables (e.g., Japanese Katakana), Abjads (writing systems in which each symbol stands for a consonant, in effect leaving it to readers to infer or otherwise supply an appropriate vowel; Devanagari, Arasbic, and old Hebrew are examples), or concepts (e.g., the digit "1" meaning "one"). Not only that, but with Unicode some characters require multiple code points; worse, some code points refer to different characters depending on the context ! On top of all that, there are accents and case to consider. (And there are three(!) distinct cases: upper, lower, and title.)

Unicode is not perfect, but is as close as we can get. Real programmers (and all programming students) ignore all that.

> Consider this example taken from The Rust Programming Language Book:
>
> Look at the Hindi word "नमस्ते" (meaning "hello") written in the Devanagari script. As UTF-8, it is stored as an array of 18 (unsigned) byte values that looks like this:
>
> [224, 164, 168, 224, 164, 174, 224, 164, 184, 224, 165, 141, 224, 164, 164, 224, 165, 135]
>
> If we look at this as Unicode codepoints (aka Unicode scalar values), those bytes look like this:
>
> ['न', 'म', 'स', '्', 'त', 'े'] = "\u0928\u092e\u0938\u094d\u0924\u0947"

There are six `char` values here, but the fourth and sixth are not letters: they're diacritic marks that don't make sense on their own; they are part of a *surrogate pair* (see below).

If we look at them as ***grapheme clusters*** (the closest Unicode has to "characters"), we'd get what a person would call the four letters that make up the Hindi word: ["न", "म", "स्", "ते"].

It can get more confusing, as some code points take up two code units (chars in Java) instead of one each. (These are called *surrogate pairs*.) For example, the Deseret character LONG I with the code point U+10400 is defined with this pair of surrogate values: U+D801 and U+DC00. The String length is two, but there is one code point. Hungarian has a letter "dz" that requires two code points. All emoji are such pairs. There are many examples.

Without knowledge of the (human) language used, it is not possible to have general library functions that can work on characters/letters. This is why text is hard! Java only supports working with codepoints; meaning and character (grapheme cluster) extraction are up to the programmer. Dealing with surrogate pairs is also left to the Java programmer. (So use the standard ICU library for this!)

After a few years ASCII became the dominant text encoding. The whole world used the ***ASCII*** (coded) character set. For example, "A" is assigned 41 (hex) or 65 (dec). ASCII is limited to a–z, A–Z, 0–9, and about 30 other punctuation and space characters (plus ~30 control characters, originally designed to control Teletype machines: advance the paper to a new line, return the carriage, pause transmission, ...).

ASCII is very simple and easy to work with, but as computers became internationalized, ASCII wasn't enough. Accented characters and other types of punctuation marks (and even a few additional letters and symbols) were needed for European languages. Also, the "new" dot-matrix printers could print a wider range of glyphs, but were limited to ASCII.

ASCII only defines code points from 0 to 127, but stores each one in a *byte*. Each byte can hold numbers (code points) from 0 to 255. So the extra 128 numbers could be used to represent extra characters.

The problem is which code points to use for which additional character? **Every vendor had a different answer!** Thus a text document written on one type of computer (or word processor) used some type of encoding that may or may not be

readable on another computer brand, or with another program, or printer. How annoying!

Microsoft supports many different encodings, but with no automatic conversion between them: Windows 98 uses "**CP-1252**" or "**CP-437**" or some other encoding, all known as "DOS Text" or "ANSI" or some similar name. In other countries completely different encoding are used, especially where the alphabet is different (e.g., Japan or Russia).

> When saving text to disk, each code point is saved in a file. When referring to the contents of files we use the term ***code unit*** (i.e., a byte) instead of *code point*. Initially code points and code units were identical: 1 code-point = 1 code unit so you didn't actually need two names. But today they may not be the same (1 code-point may be represented by more than one code unit), hence the extra terminology.

An expanded encoding was standardized (along with others) by the ISO and is today known as "***ISO Latin I***" (***ISO-8859-1***). Many text files are compatible with this encoding and most printers support these characters. Note the code points from 0 to 127 are the same for ASCII as for Latin I. All the additional characters are assigned code points in the range 128 to 255. Thus, an ASCII file is also a legal Latin I file!

While adequate for most western alphabets (those based on "Latin" letters), ISO Latin I is still limited: Very few math, science, business, or other symbols are present which limits the use for magazines and other publishing. The global Internet reaches beyond Europe to countries that use different alphabets such as Russian (Cyrillic), or don't use an alphabet at all such as traditional Korean, Chinese, and Japanese (Kanji).

> The world needs more than 256 symbols (from Ned Batchelder):
>
> Hello, world!
> Здравствуй, мир!
> Ողջո՛ւյն աշխարհ
> !مرحبا ، العالم
> !שלום, עולם
> 여보세요 세계!
> नमस्ते, दुनिया!
> **你好，世界！**

Ultimately we need a single encoding that supports characters for all written languages/scripts (plus a rich set of symbols). Various national organizations and businesses got together and defined such an encoding. (A very rare example of international cooperation!) This list of characters is called the ***Unicode standard*** (`www.unicode.org`).

The version 17 of Unicode (released in 9/2025), has 299,056 code points allocated but only 159,801 have been assigned characters. An additional 137,468 code points are reserved for private use, 2,048 are used to enable the mechanism of surrogates, and 66 are designated as non-characters. In total, there are 149,641 graphical characters (some of which do not have a visible glyph, but are still counted as graphical) and 237 special purpose code points for control and formatting.

The first 256 code points are the same as for ISO Latin I. This encoding was standardized by ISO as **ISO-10646**. The set of characters in Unicode that have been assigned code points is referred to as the ***Universal (Coded) Character Set*** or **UCS**, but is commonly just called *Unicode*. This means that any valid ASCII or ISO Latin-I text file is also a valid Unicode file (UTF-8 encoded).

> ISO-10646 came before Unicode, but didn't define as many code points as needed. Since the year 2000, Unicode and ISO-10646 are developed in tandem so they match exactly.
>
> Every so often, characters from a new set (some new language script) are assigned code points and a new version is produced. Currently (2025) Unicode 17 is the latest version.
>
> Well over 1 million code points are reserved by Unicode and ISO-10646, but many are unassigned to allow room for future growth.
>
> Besides assigning (encoding) characters from various scripts to code points, Unicode defines the properties of each: a letter, a digit, etc., and give each one a unique name such as "LATIN SMALL LETTER O WITH STROKE" (U+00F8, or "ø").

Raw Unicode is almost always shown by hexadecimal numbers, and written with the prefix "U+". So the numerical code point for the character 'A' is written as "U+0041". In Java, you represent Unicode code points as "`\uxxxx`" (four hex digits).

To store each code point directly would require 4 bytes per character (actually 21 bits today, so three bytes would be sufficient). A 4-byte per code-point encoding scheme quadruples the size of files that use ASCII or Latin I, and bloats Internet traffic. But most any single publication or web page only uses a relatively tiny subset of those characters.

Rather than store or transmit each code point as a 4-byte number, **code points are *translated* to "code units" using a method confusingly also known as an *encoding*.** Technically this is called a *character encoding scheme* and translating characters (from a language script) into code points is just an *encoding*. But everyone calls both "encoding". (Since everyone uses Unicode now as the

encoding, the term "encoding" today just means the translation of code points to code units.)

> The most common *scripts* (writing systems) had their characters assigned code points from 0 to 64K (0xFFFF). Each can be represented with exactly two bytes per code point. This is a common enough situation that this set is known as the ***Basic Multi-lingual Plane*** (or ***BMP***). ***Supplementary characters*** are characters with code points outside the BMP, in the range U+10000 to U+10FFFF. These supplementary characters are the ones that need multiple code points per character.

A given Unicode character (generally just one code point) may be encoded as one or more code units. A *variable length* encoding uses one code unit for the more commonly used characters and two or more code units for the rest. Note in Java, each code point is represented internally using one or two `chars` (a `char` in Java is two bytes). All code points in the BMP take one `char`. (So if you limit yourself to these languages and scripts only, and don't require emoji either, you can use one Java `char` for one code point. Generally in school that is an acceptable simplification. Maybe also for internally used software in western countries.)

If a document or web page uses fewer than 256 different characters in total, it can custom encode the document (the code points) using one byte per code unit, which reduces its size. (I don't know of any popular software that supports this.) The most commonly used code units are bytes (with a variable number of bytes per code point), but 16-bit (2 byte) or even 32-bit (4-byte) integers can be used.

There are a number of such encoding schemes popular today. UTF-32, UTF-16, and UTF-8 are character encoding schemes for the coded character set of the Unicode standard. But ASCII, ISO Latin-I, "Big5" (Taiwanese), and other sub-sets of Unicode are still common.

> Unicode files sometimes begin with a special sequence of bytes known as the ***byte order mark***, or "BOM". This can be used to identify the "endianness" of text, indicate the presence of Unicode text, and even identify the encoding used. (See BOM FAQ.) Note that in general it isn't possible to determine the encoding scheme used; a BOM can help in many cases but will confuse some software (some Web Browsers will try to display the BOM for example.)

UTF-32 simply represents each Unicode code point as the 32-bit integer of the same value. It's clearly the most convenient representation for internal processing, but uses significantly more memory than necessary if used as a general string representation, and is rarely used.

UTF-16 uses sequences of one or two unsigned 16-bit code units to encode Unicode code points. Values U+0000 to U+FFFF (the BMP) are encoded in one 16-bit unit with the same value. Supplementary characters are encoded in two code units, the first from the high-surrogates' range (U+D800 to U+DBFF), the second from the low-surrogates' range (U+DC00 to U+DFFF). The values U+D800 to U+DFFF are reserved for this use in UTF-16; no characters are assigned to them as code points. This means software can tell for each individual code unit in a String whether it represents a one-unit character or whether it is the first or second unit of a two-unit character. This is a significant improvement over some traditional multi-byte character encodings, where the byte value 0x41 could mean the letter "A" or be the second byte of a two-byte character.

> The encoding of BMP characters (code points) only, as 2-byte code units is called **UCS-2**. Until the year 2000 there were no supplementary code points defined in Unicode or ISO-10646. According to Wikipedia, the People's Republic of China (PRC) ruled in 2000 that all computer systems sold in its jurisdiction would have to support the GB18030 encoding. This required computer systems intended for sale in the PRC to move beyond the BMP and UCS-2 wasn't sufficient anymore.
>
> UCS-2 was later extended to permit one 2-byte code unit for characters in the BMP, and two 2-byte code units for the others. This popular encoding is known as "**UTF-16**" and is used internally by most operating systems today as well as by Java. Microsoft calls this the "Unicode" encoding. The two code units that comprise a code point are called a *surrogate pair*. Two Java `char` units act as a surrogate representation of Unicode characters in the range U+10000 through U+10FFFF.
>
> Note that Java only supports Unicode character literals that fit into one char. To work with supplemental Unicode characters in Java, you need to enter a pair of these. For example, the code point "U+10428" is entered as `"\uD801\uDC28"`. Or an `int` can be used instead, with some methods. (Other languages permit either 4 or 8 hex characters after "`\u`", but not Java.)

UTF-8 uses sequences of one to four bytes to encode Unicode code points. U+0000 to U+007F are encoded in one byte, U+0080 to U+07FF in two bytes, U+0800 to U+FFFF in three bytes, and U+10000 to U+10FFFF in four bytes. UTF-8 is designed so that the byte values 0x00 to 0x7F always represent code points U+0000 to U+007F (the *Basic Latin* block, which corresponds to the ASCII character set). These byte values never occur in the representation of other code points, a characteristic that makes UTF-8 convenient to use in software that assigns special meanings to certain ASCII characters. Here's a UTF-8 sample:

48	69	e2 84 99	c6 b4	e2 98 82	e2 84 8c	c3 b8	e1 bc a4
H	i	ℙ	ƴ	☂	ℌ	ø	ἤ

Without any doubt, UTF-8 should be used whenever possible. Java 18 and newer assume that when performing file I/O.

[*Demo changing encoding in Chrome, after installing "set character encoding" extension.]*

(Fun facts: UTF-16 has a limit of 1,112,064 codepoints it can encode. UTF-8 originally supported 5-byte and 6-byte encodings to allow for many more, but was restricted to 4 bytes to be compatible with UTF-16 in 2003. If Unicode ever grows larger than UTF-16 can handle, a new encoding will have to be created (or we can use UTF-32); maybe the 4-byte limit of UTF-8 might be lifted at that time. But Unicode has a long way to go before it ever reaches the UTF-16 limit. Unicode is more than 30 years old and has not reached 284,000 as of 2020.)

Native Encodings

There is no way to mark a text files as using a certain encoding. You just have to know the encoding used for any text. (I suppose you could include the encoding in the file's name or in it's meta-data, but nobody does that.) To help with this, **every operating system has a default or *native* encoding**. When reading text files (or standard input), you need to translate the external encoding to Java's internal encoding of UTF-16. When writing output, a translation from UTF-16 to an external encoding is needed. By default, use the platform's native encoding. (Or things like the copyright symbol end up funny looking in other tools.) Note UTF-8 is recommended but is not the default encoding on Windows! **If you don't specify an encoding, Java 18 and newer assume UTF-8 and not the operating system's native encoding.**

Of course, most software allows the program to specify the character encoding to use. In addition, some file formats including XML, web pages, and some others can specify the encoding that was used in a *header* (special data at the top of the document, usually assumed to be ASCII or UTF-8).

Since some text files start with a Unicode byte-order mark or BOM, it is sometimes possible to determine the encoding used. However, BOMs are rarely used.

For any computer you can easily enter the characters represented by keys on the keyboard. But, **how do you enter accented letters or symbols that don't have keys of their own?**

On Windows you can enter these using the ALT key and the numeric keypad digits. Holding down ALT, you type the decimal code point number using the keypad. For example, to enter "é" you would type "ALT+130" and for "¿" you would type "ALT+168".

Some applications have a short-cut way to enter characters with accents. In Word for example, you can type "CTRL+'", and then some letter such as "e" or "E". For "ü" you can type "CTRL+:" and then "u". (Word and other applications support many such shortcuts.) Keys such as CTRL+' are sometimes called *dead keys* since nothing appears immediately when you type; the key appears "dead".

You can also lookup the ALT number for any character using the "Character Map" utility (or the Internet). That also allows you to copy and then paste those characters. In addition, it will tell you the Unicode (hex) value for that character's code point.

Working with Unicode in Java

All the examples and information below are obsolete! Java internal libraries are based on old versions of Unicode and each year, Unicode releases a new version that can (and has in the past) broken compatibility.

Use the ICU Library (for working with Unicode) and the associated CLDR project (for locales) for any production-quality text processing. This includes simple stuff like string length and equals comparison!

The `java.io` package has classes that do encoding and decoding to/from Java's internal UTF-16 encoding and the system's default encoding automatically for you: `InputStreamReader`, `OutputStreamWriter`, `InputStreamReader`, `OutputStreamWriter`, and `PrintWriter`. For example, to save a file as ISO Latin I, use this:

```
PrintWriter out
   = new PrintWriter( filename, "ISO-8859-1" );
```

To see the OS native encoding used, you can read this system property:

```
String encoding = System.getProperty("file.encoding");
```

The native encoding on Windows (since "Vista") is UTF-16, but Microsoft calls this "Unicode". For older versions of Windows, the default encoding is CP-1252. That has the official name of windows-1252, but which Microsoft calls "ANSI". (Apparently MS based CP-1252 on an early draft of ISO 8859-1 (Latin I) which they obtained from ANSI, the U.S. member of the ISO. CP-1252 was never standardized by ANSI.)

Internet domain names as used in URIs are limited to a sub-set of ASCII. Recently this has been internationalized with URIs replaced with IRIs that can use a sub-set of Unicode instead. To normalize these IRIs and to support older browsers that are limited to ASCII, IRIs are often encoded as a sub-set of ASCII using a method called punycode.

Having Java use the system's native (default) encoding means files written on one platform may use a different encoding than on the reading platform. Currently (2021), it is best practice to always use methods that let you specify the encoding and not rely on the unknown system default.

As of Java 18, the PTB (*powers that be*) put an end to this nonsense and declared that Java's default encoding will no longer be system-dependent but rather always be UTF-8. Yea!

Java's use of UTF-16 causes some problems:

- Methods that only accept a `char` value cannot support supplementary characters. Newer methods accept an `int` instead.
- Instead of `String.charAt()` you may have to use the newer `String.codePointAt()` method.
- `String.length()` returns the number of code units, not code points. That is, it returns the number of bytes/2. If you want to know the number of characters in a `String` you must instead use the method `String.codePointCount( 0, string.length() )`.
- `String`s don't compare well due to case changes and accent marks. Another potential problem is that text can have multiple internal representations. For example, the name "Michèle" contains the Unicode character sequence "M i c h è l e". However, you can also use the sequence "M i c h e ` l e". The second version of the name uses a *combining sequence* ('e' + '`') to represent 'è'. These two representations will appear the same even though their internal character sequences are slightly different. `String.equals` method says that these two strings have different text. They are not ***lexicographically equal***, but they are definitely equal *linguistically*.

  This is a common problem, not just with Java. Unicode defines a *canonical* or ***normalized*** form for any Unicode string. **Use `java.text.Normalizer` to transform ("normalize") `Strings` to normal form before comparing them.** (Or use the ICU library.) Be aware however, that normalization removes some data in some cases.

This can be a security concern as well, as users can slip illegal characters into URLs or other strings, by encoding them is a strange but valid way. See below for details on normalization.

When **comparing text strings**, there is a difference between *equivalence* (different sequences of code points appear identical and should be treated as such) and *compatibility* (sequences that might appear different but have the same underlying meaning).

An example of canonical equivalence is the character "Å" that can be represented as U+00C5 (the letter "LATIN CAPITAL LETTER A WITH RING ABOVE") or as U+212B ("ANGSTROM SIGN"): these two values should be considered equivalent.

An example of Unicode compatibility is the code point U+FB00 (typographic "ff") is compatible, but not equivalent, to U+0066 U+0066 (two Latin "f" letters).

Equivalent strings are always compatible, but compatible strings are not always equivalent. Usually, you want equivalent or compatible text to compare as equal, but not always.

- You can **use `java.text.Collator` to compare Unicode `Strings`** by ignoring *tertiary* differences (for English this means case-insensitive, so "e" = "E" but not "é"), that and *secondary* differences (so "e" = "E" and "é" but not "f"), or that and *primary* differences (e.g., different ASCII control characters are not a primary difference). (Java also has an *identical* setting, where any difference is significant. See below for more details and an example.)
- When reading or writing text to/from files or a network, or converting a `String` to/from an array of `bytes`, you must specify the encoding to be used.
- The list of available encodings, and their names, is not well standardized. As a public service the Internet Assigned Numbers Authority (`IANA.org`) maintains a public registry for encoding names at `iana.org/assignments/charset-reg`. Each version of Java supports a sub-set of that list; the method `Charset.availableCharsets` returns a list of the available ones. Since Java 7, the guaranteed Java supported set have constants defined for them in java.nio.charset.StandardCharsets. (Mention issue with multi-byte encodings, and big-endian vs little endian. See this StackExchange discussion, or this Wikipedia article. Note that UTF-16 is assumed by default to be big endian, and that applies to Java literal numbers in class files (2, 4, or 8 byte numbers are big endian in Java.)
- The Java compiler also has problems with encoding; it too only knows a limited number of encodings and expects ASCII by default. The JDK comes with a tool **`naitive2ascii`** that will translate non-ASCII characters to the

"\uxxxx" notation in your Java source files (or any text file, such as a Java properties file).

A **`Charset`** (note the lowercase 's') in Java means an external encoding, or a translation from Java's internal UTF-16 to/from a sequence of bytes. (**See `java.nio.Charset`**.) Unfortunately, there is only a small list of encodings guaranteed to be supported (about 6). *There is no relationship between encodings and locales*. (See RFC-2978 for related information.)

(Show Unicode Font Lister.)

Working with Unicode Text Correctly and Securely

Because modern text may have multiple bytes per character, in multiple encodings, and may be represented as non-identical but either equivalent or compatible strings, it is difficult to work correctly with text from external sources in a safe manner. (Most programmers don't try, which is one reason so many security flaws exist in the wild.)

> For an example of what can happen if you don't take care, see the blog post Malicious NPM package uses Unicode steganography to evade detection.

Safe Text Handling

Freshly read data should be assumed to be *tainted* and should be validated and *sanitized* (after normalization), before any use. ***Sanitizing* in this context means removing or replacing any non-text bytes.** ***Validating* then looks for bad/illegal data** such as: XML tags, SQL code, quote marks, in a username field, ".." in a pathname, etc. (Normalizing is discussed below.)

Normally, input data is not sanitized: **If data contains bad bytes, just reject it**. However, if the bad data is then logged or otherwise used, it needs to be sanitized or encoded (or both).

> Data should be considered ***tainted*** (potentially bad) whenever it crosses a trust boundary: when your program obtains the data from a user, a network connection, or from persistent storage that is not used exclusively by your code. In addition, you need to make sure data is sanitized when output from your code to elsewhere (crossing a trust boundary for output).

You can run the tainted text through some regular expression to validate and/or sanitize it. If some of the data is bad, you can decide to ignore all of it, or clean it up and use the safe parts of the text. (Side note: some languages such as Perl support automatic tainting of external data, requiring you to process such data one way or another before use, or your program won't run.) For example, if the text is assumed to be UTF-8, check for illegal UTF-8 byte sequences before encoding as a Java String. If any are found, either remove them before proceeding or stop processing of the bad data. (You should encode it cleanly using something such as

Base64 (added in Java 8 as `java.util.Base64`) or URL (a.k.a. percent) encoding (`java.net.URLEncoder`), before using in logs and error messages.)

> There are two ways to validate data. *Denylisting* looks for invalid data. *Allowlisting* looks for valid data only. **Allowlisting is better when you can manage it, since with denylisting, there is a chance for unanticipated bad data to slip through.**
>
> For example, consider a username field. Denylisting all illegal characters (such as quotes) requires you to have a comprehensive list of all bad data. Allowlisting requires you to state what is allowed (for example, ASCII letters and digits only); everything else is considered bad.

Note that these steps (normalize, validate, and sanitize) must be done in a trusted environment. Usually that means on the server, not by Javascript running in the end user's web browser. For web-based applications, you cannot trust the user's web browser! Using JavaScript or webassembly on the user's browser can provide faster feedback to the user for errors, but for security you still need to validate on the server too.

Once the data has been processed this way, it is no longer considered tainted. Remember to encode bad text (as base-64 or URL-encoding) before use in logging.

Normalizing Unicode Text

Before tainted data can be sanitized and/or validated, it must be *normalized*. Normalization means putting the data into a simplified form; a simple example is to convert all letters to uppercase. Equivalent data always should have the same (unique) binary representation. Normalization ensures this. If you don't do normalization first, bad data can slip by your verification/sanitation steps (e.g. "`\uFE64script\uFE65`" will slip by a regex looking for "`<script>`".) (When working with pathnames of files, normalization is known as canonicalization.) Normalization of bad data will fail.

Normalization of ASCII is fairly easy: remove all non-printing characters (or replace them with printing sequences). For usernames, email addresses, and host (DNS) names, case doesn't matter and you should probably convert to all upper (or lower) case.

Unicode is harder to normalize correctly, as there may be many different sequences of code units that result in identical-looking text. Unicode defines four standard ways to normalize text:

NFC (Normalization Form Canonical Composition) Characters are decomposed, and then recomposed by canonical equivalence.

NFKC (Normalization Form Compatibility Composition) Characters are decomposed by compatibility, then recomposed by canonical equivalence

NFD (Normalization Form Canonical Decomposition) Characters are decomposed by canonical equivalence, and multiple combining characters are arranged in a specific order.

NFKD (Normalization Form Compatibility Decomposition) Characters are decomposed by compatibility, and multiple combining characters are arranged in a specific order.

The preferred Unicode normalization method for safety and security is known as "KC" (or "NFKC"). Use this for user IDs, etc. For non-security cases, "NFC" normalization will preserve more of the original data. For example, with NFKC ligatures such as "fi" become "fi" (two characters) and superscripts such as "n²" become "n2". That doesn't happen if you use NFC, making the result closer to the original text, but harder to work with correctly.

(The two NFD forms are used as steps in the NFC forms. They might be useful directly to remove accents, as they will use combining characters rather than a unified character. See Unicode Normalization Forms for more information.)

> When normalizing data, you often need to also preserve the original data for presentation to the user, or even for storage or for logging (in which case, you need to normalize that data before every use). This is because **normalization may lose data**. (For example, filenames on Windows (case preserving), or converting text to all uppercase.) Not all normalization methods lose data.
>
> If you do preserve the original (untrusted data), be sure to *encode* it somehow before output. For example, replace any non-ASCII printable characters with question-marks, or the hex equivalent of the bytes (e.g., "`%0E`"). That includes newlines, tabs, and runs of spaces. Any very long text should be truncated to some maximum length on output.
>
> (One useful technique, not popular for some reason, is to encode with punycode, a method designed for international URLs in web browsers, and that can convert Unicode text to ASCII text and back again.)
>
> Data that is so bad it cannot be normalized (the normalization step results in an error) should not be used at all, or else it should be treated as binary data, and encoded as text (for example, using base-64 encoding), trimmed to some maximum length. This text version can be safely used for log messages or other purposes. Another possibility is to use the hexadecimal string produced by hashing algorithms, such as MD-5.

~~Fortunately, Java includes Unicode normalization, so it becomes easy to do.~~
(Don't do this! Use the ICU library's API to work with Unicode!)

Here's an example using `java.text.Normalizer` and regular expression matching (for sanitizing and validating):

```
String normalizeAndValidateXML ( String tainted )
       throws IllegalStateException {
  assert tainted != null;
  // Normalize:
  String s = Normalizer.normalize(
      tainted, Normalizer.Form.NFKC );
  // Sanitize: Replace any non-ASCII (in this example):
  s = s.replaceAll( "[^\\p{ASCII}]", "\uFFFD" );
  // Validate: Scan normalized string for XML tags:
  Pattern pat = Pattern.compile( "[<>]" );
  Matcher m = pat.matcher( s );
  if ( m.find() || s.length() > MAX || s.length() < 1 )
      throw new IllegalStateException();
  return s;  // normalized, sanitized, and verified!
}
```

> Not shown here are the steps of reading the input bytes and attempting to encode the data from its external encoding into Java's internal String format of UTF-16. However, Java will do that for you automatically in most cases. This conversion by Java will throw exceptions if illegal byte sequences are present.
>
> (Handling of a possible byte-order mark is not shown either.)

Qu: Is the validation code above allowlisting or denylisting?

The sanitizing code above replaced illegal characters with "`\uFFFD`". This is a special Unicode character reserved for this purpose. As mentioned previously, it is often best to simply reject text with illegal characters rather than try to sanitize it and continue.

> What you do when you detect illegal characters is up to your application. You could throw an exception, or strip out the bad stuff and return the rest, or convert the bad data to `\uFFFD` (Unicode for *bad* character), which should now be harmless. (Before Unicode, question-marks were used.)

Usually you always normalize first. If there's a problem, the offending text can be sanitized for use in an error or a log statement or encode somehow (such as URL encoding). You generally sanitize the normalized text next, before trying to validate it, but if not a high-security issue, sanitation is often skipped; if the normalization step doesn't choke of the text, you can consider it safe to use is some situations.

Before use, another round of normalization may occur, depends on what the data is supposed to represent. If the data is a number, you may want to remove leading zeros and any embedded thousandths separators (e.g., “012,345” to “12345”), or possibly pad or truncate numbers of a given number of digits. Or round floating point values. Such transformations are (sadly) also called normalization.

If the data is a file’s pathname, extra steps are needed to normalize (*canonicalize* is the term used for normalized pathnames) it as well. Suppose for example, some data file is to be read, and the user enters a pathname to the file. Such a file should be under the user’s home directory (even on Windows). You can check for it like so:

```
String canonicalizePathName ( String pathTainted )
      throws IOException {
  // normalize and sanitize path String first:
  String path = Normalizer.normalize(
    pathTainted, Normalizer.Form.NFC );
  path.replaceAll( "[^\\p{ASCII}]", "\uFFFD" );
  // Now we can safely canonicalize path:
  String home = System.getProperty("user.home");
  File f = new File( home +
  System.getProperty("file.separator") + path );
  String canonicalPath = f.getCanonicalPath();
  // Now validate the path:
  if ( ! canonicalPath.startsWith( home )
    throw new IllegalArgumentException();
  return canonicalPath;
}
```

[*Todo: convert this example to using class PATH of NIO2.*]

[*Todo #2: convert all examples to use PATH of NIO2, and to use ICU.*]

(URLs/URIs/IRIs also require special steps to normalize; different data requires different, application-specific normalization, sanitation, and validation.)

Useful validation is possible using `Scanner` (after normalization):

```
Scanner in = new Scanner( System.in );
System.out.print( "Enter a number: " );
if ( ! in.hasNextDouble() ) {
   String badInput = in.nextWord();
   System.out.println( badInput
      + " is not a number" );
   // abort or something
}
```

```
double num = in.nextDouble();
// ...
```

Passwords (and Other Secrets)

You shouldn't store secrets in unprotected files on a disk or in the cloud! It's not wise to use String objects for secrets either, except very briefly. The content of `Strings` is plaintext, remains in memory unless garbage-collected (which is never guaranteed), and might be swapped to disk where anyone can access it. A common Java pattern to pass around secrets as `char[]`, and when done using them, use `Arrays.fill()` to erase the data. But one can do much better!

The best solution today (2025) is to use your operating system's security vault with a GuardedString. But that's not in the standard library; Oracle provides an implementation. In Java SE, you can use a SealedObject instead. (StackOverflow has examples of both.) These classes protect a `String` object cryptographically while it's in memory; if someone views the data in memory, it will look something like "`javax.crypto.SealedObject@27eba407`".

There are also solutions such as HashiCorp Vault, or Infisical. These provide complete secrets management and auditing. A cheaper solution would be to simply encrypt a disk partition holding the secrets files on a separate server and have your Java program use SSH to retrieve it; SSH won't allow unauthorized access and provides logs of attempts.

The best solution is to keep your secrets off-line (say in a USB thumb drive or Yubikey), but then you have to be physically present to use it. (PKI certificate authorities do this.)

Note that no matter your solution, your Java program still needs the decrypted secret for a second or two when setting up the database connection (or accessing some 3rd party service) across the network (using encrypted protocols of course). But this is the best one can do.

Using `SealedObjects` or `GuardedStrings` minimizes the time the secret is available to evildoers who have admin access to your servers. As that is an unlikely scenario, this solution is actually pretty secure.

Here's an example of using `SealedObject`, taken from StackOverflow:

```
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SealedObject;
import java.security.Key;

public class SealedObjectDemo {
   public static void main(String[] args) {
      // The try-blocks are omitted for clarity:
```

```
        KeyGenerator keyGenerator =
           KeyGenerator.getInstance("DESede");
        Key key = keyGenerator.generateKey();
        Cipher cipher = Cipher.getInstance("DESede");
        cipher.init(Cipher.ENCRYPT_MODE, key);
        SealedObject so = new SealedObject(
           getSecret("nameOfSecret"), cipher);

        //connect string except for the secrets:
        StringBuilder url = ...;

        // Use of unencrypted secrets is brief:
        Connection conn = DriverManager.getConnection(
           url.append(
              (String) so.getObject(key).toString())
        );
      }
      String getSecret (String name) {
          // Fetch and decrypt secret from external src
          ...
      }
}
```

(If the whole connection string fits into a cache line, the decrypted secret may never make it into memory at all!)

Similarly, do not pass secrets on the command line (these can be seen) or in environment variables (these can be seen too). In the real world, you may not be able to avoid doing such things, but there are ways to make such use safer (for example, you can harden Linux to not permit non-root users access to process' memory or environment; Windows includes a hardware module, TPM, to protect a vault of secrets.).

Validation and Sanitation with Regular Expressions

Many validation libraries exist, especially in Jakarta EE, and they can be used instead of REs. However, using REs is common. Sadly, there are many security issues with REs! Here are some:

Make sure that your RE implementation handles Unicode and uses locale (language) specific case-insensitive comparisons. For example, in the "`tr`" (Turkish) locale, "I" and "i" are different letters! The lowercase version of "I" is dotless-i "ı". So **when ignoring case, the locale matters**.

Some RE implementations do no handle the NUL character correctly. You should confirm that yours does. If that's a problem, you need to scan data for those before using REs.

Always use "whole data" matches. Don't use the RE "`cat`", use "`^cat$`". Note the anchors "`^`" and "`$`" don't always anchor the whole text but instead just a line, so use `\A` and `\z` or whatever your library supports for this. (These where added to POSIX regular expressions in 2025, but were commonly found in most RE libraries.)

A related issue is precedence of branches: the RE "`^AA|BB$`" means the data starts with `AA` or ends with `BB`, not a whole-data match of either `AA` or `BB`! The correct way to use branches is to use parenthesis "`^(AA|BB)$`".

Check if "`.`" matches newlines. Usually there are options to control that, such as "multi-line" or "dot" mode.

Many RE implementations are vulnerable to ReDoS attacks. (For details, see https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS. Such libraries support backtracking, and can take a very long time to check some data that malicious users can figure out to use. Use a linter to catch such "evil" REs such as "([a-z]+)+" and rewrite them to avoid backtracking, or use a different RE library that never backtracks. Using "possessive modifiers" that do not backtrack instead of normal "*" and "+". Use repetition counts ("{0,10}" and not "*" or "+" if possible).

Never allow users to supply REs.

You should add test cases for your validation REs to make sure they reject bad data such as the common errors discussed here. (Some static code linters can detect bad REs, as can the Checker framework.)

Finally, when the text input data is structured, you have other issues to watch for. XML-EE attacks allow well-formed XML to download and include arbitrary files! CSV data fields that start with special characters such as "=" can run arbitrary code during processing! JSON is safer but there are still some security issues to watch for, especially overflow of numbers.

> When validating numbers, when possible only allow positive integers that fit into an `int` in data. If you must allow floating-point numbers, watch out for all the special cases such as Infinity and NaN.

There are many more security issues as well. See SecureCoding.cert.org for a comprehensive list of issues and best practices for Java.

Examples of SAFECode Secure Development Practices

- Threat Modeling
- Use Least Privilege
- Implement Sandboxing
- ~~Minimize~~ Eliminate Use of Unsafe String and Buffer Functions
- Validate Input and Output to Mitigate Common Vulnerabilities

- Use Robust Integer Operations for Dynamic Memory Allocations and Array Offsets
- Use Anti-Cross Site Scripting (XSS) Libraries
- Use Canonical Data Formats
- Avoid String Concatenation for Dynamic SQL Statements
- Eliminate Weak Cryptography
- Use Logging and Tracking
- Determine Attack Surface
- Use Appropriate Testing Tools
- Perform Fuzz/Robustness Testing
- Perform Penetration Testing
- Use a Current Compiler Toolset
- Use Static Analysis Tools

(Also SAFECode's Fundamental Practices for Secure Software Development.)

I'll show one more data safety issue here. (I'm pretending that (a) you care and (b) I haven't already turned your brain into mush.) After normalizing and sanitizing data, you still need to make sure you don't process non-text bytes in the data. How can there be any? When filling a buffer (array) with text, Java will prevent a buffer overflow by limiting the number of bytes added, but truncating the imput to the size of the buffer. But you must be careful that the final bytes actually form a complete character; combining characters, ligatures, etc., make this difficult to determine.

The only safe way is to walk through the (normalized, sanitized, and validated) string, one grapheme cluster at a time, until you hit the end. The `java.text.BreakIterator` is designed for that (and other things). (Show `BreakIterDemo.java`.) Other ways to walk through (iterate over) a `String` have been shown previously. (Or, use the ICU library!)

Fizz-Buzz problem revisited:

Given a non-empty string and an int N, return the string made starting with char 0, and then every Nth char of the string. So if N is 3, use char 0, 3, 6, ... and so on. N is 1 or more. Examples:

```
everyNth("Miracle", 2) → "Mrce"
everyNth("abcdefg", 2) → "aceg"
everyNth("abcdefg", 3) → "adg"
```

In this version, normalize the input correctly, and process any Unicode text correctly. Document what your code will do if normalization fails. (*Show solution*.)

Visual Spoofing

Visual spoofing happens when two different strings are mistaken as being equivalent, because the look identical or very similar. Attackers will sometimes use visual spoofing as part of an attack.

Spoofing by decomposition means using a different sequence of code points to represent visually identical characters. For example, "z̷" may be expressed as U+007A U+0335 (z + combining short stroke overlay), or as U+01B6. Normalization solves this problem.

Spoofing by mixed-script means using code points that look the same but mean different things. For example, the Greek omicron and the Latin "o" typically look the same, even though they are in different sections of Unicode. Normalization doesn't help with this.

Same-script spoofing simply substitutes characters that look similar. E.g., "-" Hyphen-minus U+002D vs. a plain hyphen "‐" U+2010. Examples include using the ASCII digit "0" instead of an uppercase letter "O" and the digit "1" instead of a lowercase letter "l". (Fun fact: The earliest keyboards didn't include keys for one or zero, you were supposed to use the el and capital-O keys instead.) For example, an attacker might try to create a malicious `paypa1.com` domain instead of `paypal.com` (one verses ell). The sequence "`rn`" is sometimes misread as the letter "`m`". Many fonts take care to make such confusable characters more distinguishable, but not all commonly used fonts do.

Bidirectional text spoofing is yet another type of malicious text. Some languages are mostly right-to-left, but switch in certain situations to left-to-right. Thus, Unicode includes mechanisms to indicate direction of parts of a string of text. But this means that the string "`olleh`", surrounded by "use right-to-left" markers will visually look the same as "`hello`".

Visual spoofing can be very challenging to counter in general. Normalization and using distinctive fonts are not always enough, but can help a lot.

> A standard technique to combat visual spoofing in URLs is called punycode and is supported by all modern browsers (if not always turned on by default).

Here are some current best practices for working with text data in Java. Firstly, follow the safe practices discussed above: input a limited number of bytes, convert to a Java String using the expected encoding, remove DOM if present, use an appropriate normalization method, validate (using an allowlist if possible) the text including the length, log bad inputs (but encode them first), and finally sanitize the text if appropriate. (It's worth mentioning that when exporting text, use an appropriate

encoding and include a BOM if byte order is uncertain.) Now you can process the text.

Iterate Over Unicode Strings (*Skip to Design*)

To iterate over a String s one grapheme cluster at a time, use the ICU library!

To iterate over a `String` s one code point at a time, use code such as this:

```
for ( int cp = 0, i = 0; i < s.length();
          i += Character.charCount(cp)) {
    cp = s.codePointAt(i);
    ...
}
```

Or:

```
for(int cp : s.codePoints().toArray()) { ... }
```

Java 8 added `Streams`, allowing another way to iterate over a `String`:

```
s.codePoints().forEach(cp -> ...);
```

While much shorter to type, Streams are likely to be as efficient.

To get the `i`th code point (character) as opposed to the `i`th code unit (`char`), use code like this:

```
String s = ...;
int positionInString = s.offsetByCodePoints(0,i);
int theChar = s.codePointAt( positionInString );
```

To count the number of code points (characters), in a String, use:

```
int numCharacters = s.codePointCount(0,s.length());
```

> As mentioned elsewhere, don't worry about `chars` or code points. Just use `Strings` to hold characters (really, grapheme clusters) and use the methods shown previously to work with them. Remember that you likely want to iterate over grapheme clusters (characters) and not code points! Use the ICU library for that!

Text Files and End of Lines

A related issue to encoding of text, is that of storing text in files. It may seem simple to convert grapheme clusters to code points to (say UTF-8) code units, and write out the code units to a file, and later read them in and convert back.

But text files are defined as zero or more *lines* of text. Lines have an (implementation-defined) length limit. The end of line (EOL) is marked differently on different systems, including some multi-character sequences on

some. On other systems there is only an EOL between lines (line separator), not after every line (line terminator).

On text data transmitted across the Internet there is supposed to be a '`\r\n`' sequence terminating each line (according to a very old standard). DOS and Windows typically uses this, but Unix systems typically just use '`\n`'. The problem is made worse by poor programming and incorrect use of C language features. Newline is a good article on this from Wikipedia.

The Java approach is to write text files using the native EOL convention and try to read in any text file assuming native EOLs. The `Scanner` class is apparently intended to recognize input from different OS's. In `Scanner`'s methods, a line break is something that matches the regular expression: "`\r\n|[\n\r\u2028\u2029\u0085]`". A `BufferedReader.readLine` is similar.

The `PrintStream.println` method adds the native EOL as a line terminator, confusingly known as the system property `line.separator` (see `System.getProperty`).

For mobile apps, or client-server (enterprise Java) applications, most likely you will need to keep any text portable regardless of the native EOL string. It may be best to write "`\r\n`" (or avoid Windows and just use "`\n`") explicitly and avoid `println` methods, but to be flexible on reading text (so use `Scanner`).

Locales — Cultural Localization (Appearance) of Data

Background Information: *Locale support* refers to an application respecting cultural preferences regarding alphabets, sorting, paper sizes, and formatting of numbers, dates, addresses, and other items, for a given culture, usually identified by the dominant language and the geographical region. A *locale* is an object that represents a culture by language and region (and sometimes also a variant).

The input and output of data is dependent on the current locale setting. Items affected include dates, times, numbers, currencies, percentages, names, addresses, phone numbers, and even the language used in messages.

Locales use short, formatted strings for their names of the form "*language_region*". For example, "`sv_SE`" is the locale for Sw edish (`sv`) as spoken in Sweden (`SE`). Other possibilities might include "`en_US`" (U.S. English) and "`fr_CA`" (French Canadian). Some web browsers require hyphens instead of underscores in locale names.

If more than one character set can be used for a locale, then the name can take the form "*language_region.variant",* where *variant* represents a codeset (another term for a charset, or an encoding of Unicode). For example, "`fr_BE.UTF-8`"

represents the French language (`fr`) as spoken in Belgium (`BE`), with a `UTF-8` character set encoding.

> Most systems including Java use the two letter English version for the *language* code and for the country code (*region*). These codes are standardized and maintained by the ISO (show links in resources).

A *variant* is user-defined, and may represent differences other than charsets. It might represent a dialect, the U.S. military time representation for en_US, or to distinguish between Windows and Unix text file formats.

Locale names in Java have the form "*basename_language_country_variant*", where *basename* is usually the name of an application and selected by the programmer. Multiple variants are allowed, separated by an underscore.

The list of locales supported by Java depend on the version and what additional locales are available on that system. You can get the list of all supported locales with `Locale.getAvailableLocales()`. (That method exists in a few other classes as well).

The Unicode Consortium has released the ***Common Locale Data Repository*** (CLDR) project to "support the world's languages, with the largest and most extensive standard repository of locale data available." The CLDR has become the de-facto standard for locale data. The CLDR's XML-based locale data has been incorporated into the JDK 8 release, however it is disabled by default.

> A design flaw in Unicode means that the same sequence of code points might have different meanings, depending on the locale! Use the ICU library to process text and always set a locale to use first, even if you must guess which one!

To select the desired locale data source, use the `java.locale.providers` system property, listing the data sources in the preferred order. For example:

```
java.locale.providers=HOST,SPI,CLDR,JRE
```

The default behavior is equivalent to the following setting:

```
java.locale.providers=JRE,SPI
```

Java 8 and newer includes an enhanced locale API: `Locale.filter` methods return a list of `Locale` instances that match specified criteria, and `Locale.filterTags` methods return a list of language tags that match specified criteria, as defined in BCP 47. `Locale.lookup` methods return the best-matching locale or language tag, using the lookup mechanism defined in BCP 47. `Locale` class now supports some locale extensions (beyond just region and language).

Locales are in `java.util` package. There is a default locale set by the user (in Unix via the `LANG` environment variable; in Windows via the *Region and Language* control panel).

Creating Locales

If not using the default locale, you can request one from the user (via a "Language" menu perhaps), or on the command line for stand-alone applications~~, or from `PARAM` tags in an Applet~~. You can also use a text (e.g., Java properties) file to hold a locale name to use.)

To create a locale object for a specific locale, use code such as this:

```
Locale frenchCanadian = new Locale( "fr", "CA" );
```

Class `Locale` has several useful methods including the country and language names (*displayNames*) and static methods for getting and setting the default `Locale`. In addition, there is a large number of locales provided as static properties of class `Locale` for you to use. (These include support for language-specific names for days and months.)

Locale-Sensitive Sorting and Comparison

One use of locales is when sorting text in a language-specific alphabet. The normal `String.compareTo` method won't work and thus the `java.util.Collections.sort` method doesn't work either. This is true even for normalized text. Consider that the sort order of "ch" or "ll" is different in Spanish than in English, or the sort order of *eszett* (sharp-S in German) vs "ss". For this reason, ~~you need to **use a `Collator` object** instead~~. you should use the latest ICU library instead.

The `java.text.Collator` class provides natural language comparisons. Natural language comparisons depend upon locale-specific rules that determine the equality and ordering of characters in a particular writing system.

A `Collator` object can understand several *levels* of character differences. A collator has methods that control how precisely you want the comparison to be: Different letters ("`w`" and "`p`") are considered *primarily* different. The difference between accented and unaccented versions of the same letter ("`o`" and "`ŏ`") are considered a *secondary* difference. Different cases of the same letter ("`A`" and "`a`") are *tertiary* differences. So depending upon how you configure a `Collator` instance, you can consider the words "`Michèle`" and "`Michele`" to be equal.

To sort according to a locale you must also use a **`Collator`** object, custom built for the locale you want. This object has a working `compare` method you can use:

```
Locale loc = ...;
Collator coll = Collator.getInstance( loc );
```

```
collator.setStrength( Collator.TERTIARY );
...
if ( coll.compare( a, b ) < 0 ...
...
Collections.sort( myList, coll );
```

(*Show again the ISO docs for country codes and language codes.*)

This shows how to compare English text ignoring case and accent marks:

```
Collator coll = Collator.getInstance(Locale.US);
 coll.setStrength(Collator.PRIMARY);
 if( coll.compare("abc", "ABC") == 0 ) {
     System.out.println("Strings are equivalent");
 }
```

Formatting Locale-sensitive Data

The `java.text` package contains several classes used for formatting numeric and date information. Last term we discussedthe `DateFormat` abstract class whose objects can be used to generate date, time, and date-and-time formatter objects, customized for a given locale. (Ex: 10000.23 ==> 10,000.23 or 1,0000.23 or 10.000,23 or ...) These formatters know about two dozen locale conventions, all the major ones you might need for business in the Western world. However there is a way to specify new formats for numbers and dates (see also `SimpleDateFormat`).

> Using `String.format` or `java.io.PrintStream.printf` (which in turn use `java.util.Formatter`) are the modern replacements for some of the formatters discussed here.

These formatting objects generally have a **`format`** method to produce output, and a **`parse`** method to read a String according to the formatting conventions.

In addition to `DateFormat` there is `Calendar`, `GregorianCalendar`, `DateFormatSymbols`, and the new `java.time` classes to help format dates and times:

```
DateFormatSymbols dfs = new
DateFormatSymbols( Locale.FRANCE );
String [] monthNames = dfs.getMonths();
Calendar cal = Calendar.getInstance( Locale.FRANCE );
System.out.println( monthNames[ cal.get(cal.MONTH) ] );
```

It is possible to set a local time zone from the locale as well, to ensure dates and times display the way the user expects. Note setting the time zone doesn't affect the date or time, merely the way it is displayed. (So dates and times for network wide resources can be compared correctly *and* displayed correctly too.

Numbers are formatted using `NumberFormat`:

```
NumberFormat nf =
NumberFormat.getInstance(Locale.FRENCH);
String val = nf.format( 10000.22 );
Long lObj = nf.parse( "1,000" );  long l =
lObj.longValue();
```

You can define custom number formats with class `DecimalFormat`. To format numbers as percentages (i.e., "0.53" as "53%") use `getPercentInstance`, to format as a currency use `getCurrencyInstance`.

There are two other formatters (`Choice` and `Message`) discussed later.

Finally, this package `java.text` contains class `Collator` and `BreakIterator`. A `Collator` can be used to compare and sort strings in a locale-sensitive way, and deal with capitalization and accents. (e.g., Spanish: a, b, c, ch, d, e, ...) A `BreakIterator` can be used to find character, word, line, and sentence boundaries (not so easy in some languages!).

Warning: Both of these classes are based on old Unicode libraries. Use the latest ICU library instead.

`ResourceBundles` and property files

ResourceBundles are classes that convert a *key* to a *value*. These are often used to internationalize an application. The values are put in `.class` files that are named for their locales, for instance *Msgs_en_US.class*. To use a resource bundle, you use code like this:

```
ResourceBundle msgs =
   ResourceBundle.getBundle("Msgs", locale);
```

The system locates the correct resource bundle by constructing the class name from the supplied basename ("`Msgs`") and the requested `locale` (or default locale). Then the individual localized values are fetched this way:

```
String s = msgs.getString( "key" );

// The value need not be a string:
int population =
  ((Integer) msgs.getObject( "key" )).intValue();
```

The set of keys should be the same for all supported `ResourceBundles` in a given application. If a key isn't found, a `MissingResourceException` is thrown. The key names must be `Strings`, the values can be anything but usually are translated strings, currency or other symbols to use, icons, conversion values, and so on. (Actually there are useful for any key-to-value mappings.)

`ResourceBundle` classes can be created in a number of ways, such as by extending `ListResourceBundle` and overriding the `getContents` method. But the easiest way is to **create a text *properties file* and let the system create a `ResourceBundle` class automatically**.

The properties files are easier to work with but have a limitation: The values must be ISO Latin I text (not even Unicode, so use the `naitive2ascii` tool that comes with the JDK). The name of the file is the same as for the class, except the extension is "`.properties`":

***bundleName_LanguageName_Location_variant*.properties**

All parts but the *bundlename* and extension are optional. **The system searches for the most specific one it can find, looking first for .class files and then .property files.**

The contents of a property file is a set of lines in the form *key* = *value*. Blank lines and lines starting with "#" (comments) are ignored. Spaces are allowed around the "=" and in front of the keys. The file should be in ASCII; the `native2ascii` tool (included with the JDK) can be used to convert property files to and from other character encodings.

> For resources that aren't strings (such as icons), it is easier to extend `ListResourceBundle`.

The system locates all relevant resource bundle classes and builds a hierarchy to use in the lookup algorithm. So you don't need to put every key in every resource bundle; you can instead create defaults and over-ride them in more specific resource bundles. When looking for a specific bundle, the system first looks for a .class, then a .properties file if no class is found. Suppose the default locale is `en_US` and the requested locale is `fr_FR`. Then the lookup will search for bundles in this order (the variant is ignored here to simply the example):

```
Msgs_fr_FR.class
Msgs_fr_FR.properties

Msgs_fr.class
Msgs_fr.properties

Msgs_en_US.class
Msgs_en_US.properties

Msgs_en.class
Msgs_en.properties

Msgs.class
Msgs.properties
```

It is important to have a default (`Msgs`) bundle, and also any intervening bundles even if empty. (For example if you supply `Msgs_fr_FR` you must supply `Msgs_fr` as well, and should supply `Msgs` too, or the lookup algorithm may not work correctly.)

Java 6 added XML property files and other new features using the `ResourceBundle.Control` class. You can use XML this way:

```
ResourceBundle bundle = ResourceBundle.getBundle(
  "Test2", new XMLResourceBundleControl() );
String string = bundle.getString( "HelpKey" );
System.out.println( "HelpKey: " + string );
```

and the XML file `Test2.xml` looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM
     "http://java.sun.com/dtd/properties.dtd">
<properties>
  <entry key="OkKey">OK</entry>
    <entry key="CancelKey">Cancel</entry>
    <entry key="HelpKey">Help</entry>
    <entry key="YesKey">Yes</entry>
    <entry key="NoKey">No</entry>
 </properties>
```

Unlike the older property file syntax, the XML format allows values with leading spaces.

The exact syntax of Java property files can be found in the Java API docs for java.util.Properties.load method. Some of the new features include: using ":" instead of "=" (you can also just use whitespace to separate keys and values!); using line continuation of <backslash><end of line><whitespace> to allow long "logical" lines; and allowing some backslash escapes, including \u*xxxx*.

Messages and Choices [*Skip this part.*]

When translating text some problems can arise. The position of numbers and other values within text changes from language to language. The other problem is in dealing with plurals.

A `MessageFormat` works a bit like `C`'s `printf` statement: a string with placeholders for the values is given (called the *pattern*), along with an array of values to substitute. The `MessageFormat` class then formats the values for the proper locale and substitutes them in the string in the right places. The string can be supplied from a resource bundle. For example, the en_US string may look like "`You have {0} files older than {1} days`".

The placeholders may just indicate which value to use "{0}" in which case the value is a String, or it may indicate the type (and optionally the format) of the value, as in "{1,number,$'#',##}" will produce a number format with the pound-sign quoted, with a result such as: "$#31,45". Some other possibilities include "{1,date|time|number|choice,*format*}" where *format* for dates and times can be short|medium|full|long|*customFormat*, for numbers can be currency|percent|integer|*customFormat*, and for choices must be a choice format.

A `ChoiceFormat` is used to convert values to strings. This is done to convert day of week numbers into day name strings, or to handle plurals: "There are no seats available", "There is 1 seat available", "There are {0} seats available". The strings used can be kept in a resource bundle. See the Java doc API for examples.

See IGreet.java (ListResourceBundle demo) and Stocks.java (Property file demo).

(Note for text output use String sep=System.getProperty("line.separator");)

> As of 2005, everyone on Earth uses ISO-216 paper sizes except the US and Canada. Every country has adopted the metric system except for the US, Liberia, and Myanmar (formally Burma). See web resource links.

Other Issues — Names, Addresses, paper sizes (for printing), etc. [*Skip this part.*]

A student once had their name on the class roll as *Vivekananda Lnu*. His last name was not Lnu. His name was Vivekananda. "Lnu" stood for *Last Name Unknown*. Well, his last name was not unknown, it was nonexistent—the student only had one name. He finally gave up. In the US, on all of his official documents, he is now Vivek Ananda.

At one university, if there's no first or last name, "." (dot) is used as a placeholder. The reason for the dot was not because the student only had one name. For him, it was because of the way his passport was printed from his home country. The entirety of his name (both first and last) was printed as a first name. He explained that getting it fixed on the passport was extremely difficult and because of immigration issues, the name in the school record needs to match the name on the passport, so he just left things as they were.

In many cultures, the family name and given name are not always in the same order as in English. Sometimes a patronymic name is included as well. Further, where the family name begins for purposes of alphabetization depends on the traditions of the language and culture.

In alphabetizing, apostrophes and hyphens should be ignored (O'Reilly and OReilly are equivalent).

How to alphabetize names from different cultures/languages in a single list? For example, where would you put "Борис Николаевич Ельцин" (Boris Nikolayevich

Yeltsin)? One possibility: Alphabetize using first the Latin alphabet and then other alphabets in the order of their names in English (Cyrillic before Greek).

> In some cultures, people's titles are first, last, or even in the middle of their names! When sorting, you need to ignore the titles. (Or all the "MD" doctors will sort together.)

Unicode allows various characters to be written as a single "pre-composed" code point or as a sequence of a one-character code point plus one or more combining code points. Consider "á": it could be written as a single code point that corresponds to the ISO-8859 "á", or as two code points, one being ASCII "a" and the other being the combining code point "add acute accent mark". There are characters that can have five or more different representations! This makes comparing strings, hashing URLs, computing MD5 or SHA digests, etc., problematic.

The solution is to use ***Unicode normalization*** rules that are used to transform Unicode strings into a single form before use. But what normalization rules to use? There are four normalization methods available (discussed previously). Often it won't matter which you use. Always use the latest ICU library for this. (Another standard way is to use Internet IRI punycode normalization.)

Some Common Issues with Locales

There are a large number of "gotchas" when communicating with people in other locales. You can test your code for different cultures (locales) by using code like this:

```
// Pretend to be running app in Türkiye:
Locale.setDefault( Locale.of("tr", "TR");
```

Here's a short list:

- Parsing dates (from Strings): the US format of mm/dd/yyyy is not universal or even common. Always specify the formatting of dates and times.
- Parsing numbers (from Strings): three-digit grouping or using comma for thousands' separator is something that changes.
- Comparing Strings: Figure out which Unicode normalization to use, and don't forget to remove accents or to convert to the same case before comparison, when sensible. (Note that is some languages, dropping accent marks changes the meaning of the words! In such cases, you can try *transliterating* the strings.)
- Parsing Strings with regular expressions has the same issues as comparing strings. Be sure to use appropriate normalization first, then use appropriate RegEx flags (such as to ignore case). Note that digits used in parts of the world are different from the 0-9 digits you are probably used to.

- Cultural assumptions are almost always wrong. Consider the length of a field that holds a person's name. How long should it be? Whatever you think, be prepared to handle exceptions. (For example, Spanish-speaking people tend to have very long full names, in case that a complete name is needed. The famous painter Picasso actually had the full name "Pablo Diego José Francisco de Paula Juan Nepomuceno María de los Remedios Cipriano de la Santísima Trinidad Ruiz y Picasso".)
- Some locales have bugs in them! Türkiye, Thailand, and others too. On top of that, there are oddball locales that are correct but violate common assumptions. (For example, the locale for "en-GB" abbreviates September to "Sept" whereas all other months have only three letters. In other English-speaking parts of the world, The abbreviation is "Sep".)

Summary of Text, Unicode, and Locales

Handling text data safely requires a deeper understanding of text and its issues than most programmer have.

Characters (very loosely speaking) are represented by one or more numbers called *code-points*. Which sequence of code points corresponds to which character is determined by Unicode, which has assigned over 150,000 code points (as of 2025). In most programming books and examples online, the simplification is made that each character is one code point. But don't make that assumption for production-quality code!

Initially Java uses one `char` (two bytes) to represent each code point. Today they use one or two `chars` per code point. Working with text on a character-by-character basis means being aware that the code points may not fit into a `char`, so use an `int`. Worse, some characters (grapheme clusters) are sequences of code points. Just use Strings to hold characters.

Use newer code-point aware methods to calculate `String` length or to access a character by its offset in the string, from the ICU library. (Unicode changes its rules from year to year, so you must use the latest version.)

Lengths of Strings are usually specified in bytes, not characters. To determine an appropriate number of bytes to hold a specific number of characters is impossible. Consider that characters in a string may be encoded using an unknown number of code points. The best advice is to to use a huge buffer for user input, then normalize, and then allocate the correct number of bytes for it.

When inputting or outputting text, use the correct *encoding*. While every computer has some default encoding, it works best to use UTF-8 when possible.

Limit the amount of input read to avoid absurdly long inputs (either by error or malicious).

Convert the read bytes into the expected encoding; abort if not in that encoding. Note if the input was length-limited, the final bytes might only be part of a code-point and should be removed.

If the previous step didn't handle it already, remove any BOM (byte order mark). Multi-byte encodings such as UTF-16 and UTF-32 both have two forms: "little endian" and "big endian".

If you don't know if the recipient expects big-endian (using in networking protocols) or little-endian (used by most CPUs today), you should add a byte order marker (BOM) at the beginning of the text to make sure that the receiver interprets it correctly, and when receiving such text, your application needs to pay attention to that. Avoid such issues by using UTF-8 when you have a choice.

After validating the number of bytes and that they for a String with the expected encoding, the next step is normalizing the data. Then validate it for length and (if possible) use an allowlist. While you should generally reject input containing bad data, sometimes you need to sanitize then use it (such as removing leading zeros from numbers) or encode text to make it suitable for log messages.

Normalization, sanitation, and validation need to be done in a trusted environment, one that a user cannot bypass (or substitute bad data after validation). For web-based applications, these tasks should be done on the server. Client-side validation may be used in addition to this, to provide quicker feedback to the user.

When comparing text Strings, you need to consider both canonical equivalence (different sequences of code points appear identical and should be treated as such), Unicode compatibility (sequences that might appear different but have the same underlying meaning). For example, when a user enters their name it might not be byte-for-byte the same as what you stored in a database. Normalization helps with this. You normalize both strings before comparing them.

Visual spoofing (making equivalent text appear different or vice-versa) should be dealt with if possible during validation. Consider using punycode.

Finally, don't forget to manage locales and use locale-aware sorting, comparison, and formatting of text.

Lecture 15 — Object-Oriented Design (and Related Topics)

> ***Write your code as if the next person to maintain it is a homicidal maniac who knows where you live.*** *–"Head First Servlets & JSP" p.314*

[*Some of the following material adapted from Cay Horstmann's excellent "Object-Oriented Design & Patterns", (C)2004 John Wiley & Sons, Inc.*]

Following a design procedure to produce software (or anything) is an absolute must. (I grant it can be hard for a student to see that from the simplistic programming assignments often given in a classroom setting.) There are many reasons for this, some of which are discussed below.

> If you are hired as an entry-level software developer (programmer), you will not be expected to be able to design important software systems yourself. But you will need to be able to understand and work with more senior developers (sometimes called software engineers). They will do "high level" designs (system "architecture") and you will need to fill in the details to make a more complete design. So no matter what you expect to do later, you will need to know about software design.

You should practice using a design process on the assignments given, even though you won't need to. But first, what is design? Let's start with a related question: What is software engineering and why should you care?

The field of ***Software Engineering*** includes software development as well as the management of software projects and the process of producing software. Legally speaking, "software engineering" (a.k.a. software design) is not a recognized engineering discipline in many parts of the world. This is changing though; the IEEE offers certifications for this, and provides the software engineering body of knowledge (SWEBOK). Many states in the U.S. now require state licensing before you can call yourself a "software engineer". This includes Florida since 2012, (last checked in 2020) and about 10 or more other states have or indicated they will have such license requirements. Most of these states used the NCEES SE exam (discontinued in 4/2019 due to low demand). (IEEE study guide for the exam can be purchased here. See also the exam description.)

> Software engineers may have different titles including software designer, architect, or senior programmer.

In any professional field, not following standard procedures carries severe consequences, including legal penalties. For programmers, this means that you cannot be sued for malpractice unless you are a licensed software engineer. (This fact does account in part for the bad quality of much of the software used in the world.) Following proper design, testing, and auditing are often required by government and industry regulations to ensure ***software quality control.***

There are many standards and certifications that apply to software. Most concern the process of development, and rarely force a particular software design choice. These are intended to ensure quality software. Examples include ISO 9000 certification (required to sell software in the EU), DO-178B/DO-254 certification (required for avionic systems), etc. These standards state how often to conduct quality control audits (to ensure all developers are following the agreed upon methodology), and other activities.

There are also applicable security and safety standards for various industries (such as PCI-DSS standard for handling credit card data, HIPAA for health data, etc.). There is also a growing number of data privacy standards, including the EU's General Data Protection Regulation and the California Consumer Privacy Act (CCPA).

ISO 12207 is a standard for *software lifecycle processes*. It aims to be "the" standard that defines all the tasks required for developing and maintaining (quality) software (although there are other standards used in the world). The standard has the main objective of supplying a common structure so that the buyers, suppliers, developers, maintainers, operators, managers, and technicians, involved with the software development, use a common language.

ISO 9126 is related to the others. It aims to address some human bias from the definition of quality, especially for projects that were started without a proper definition of success in the first place. Unlike ISO 9000, this standard defines the quality of the final project by measuring characteristics (*metrics*) such as functionality, reliability, usability, maintainability, and others.

(Show xkcd cartoon about why so many standards.)

So why must you follow a design methodology? Many common programming tasks seem simpler than they really are and can require a genius to get them right, especially the first time you encounter them. Some of the issues "production" or "professional" quality software should address include thread safety, resource management (e.g., memory or resource leaks), security (protecting confidential data), and safety. Following standard design patterns and design procedures means your software will use the best-known solutions.

Not following proper design and testing methodology leads to systems that in the past have bankrupt companies and endangered human life (see Wikipedia "Therac-25").

More recently, the CBS-owned TV news station WCCO in Minneapolis-St. Paul, Minnesota reported in June '10 that a woman being moved from one nursing home to another died while being transported on the 22nd of April because of a software problem in the ambulance's onboard oxygen system.)

In 2013, a bug, apparently present for many years, was disclosed about Xerox copiers. In many cases, the copiers would change numbers in the copies! The damage is large: legal contracts, financial reports, and other vital documents were mangled.

In 2016, it was reported that the database that is used to map IP addresses to street addresses, MaxMind, had a design flaw: MaxMind decided to set default locations at the city, state and country level to the center of that region when it knows only roughly where some IP address is located. If it knows only that an IP address is somewhere in the U.S. and can't figure out anything more about where it is, it will point to the center of the country. That happened to be a farm in Kansas, whose residents get accused of all sorts of Internet crime, and have their home raided frequently. The same happens to the center of a state and city. (After 16 years, MaxMind figured out their role in the troubles of so many people and have changed the defaults to be in nearby bodies of water, so no people get harassed.)

In 2021, it was reported that a software bug sent 39 people to prison and over 700 lost their jobs in the UK, for stealing tens of thousands of pounds from the postal service over the past 20 years. All were innocent! The postal service used and trusted a program named Horizon to manage the funds but held local postmasters responsible for its report of lost funds. (Worse, it seems management may have known about the issue for some time and kept quite to "ensure trust" in the postal service.) Proper testing and code review were not performed, nor were reports of these bugs investigated.

> See (and avoid!) this list of the top 25 errors from SANS.org, made when developing software.

More and more often, software developers are getting sued (or losing judgments in binding arbitration) for not following industry best practices and standards, including ethical standards. (This applies to project management too.)

> Another reason to follow proper design methodology is cost. **Changes and fixes can cost 10 times as much to do in the design phase as in the analysis phase, and 10 times as much to do in the implementation phase as the design phase.** (Don't even think about changing the requirements or design in the maintenance phase!) An example is HealthCare.gov, a website

> that cost over $400 million initially and ended up costing 1.7 billion dollars. Guess why!
>
> When changes requested by the customer will cost a lot, they may not realize that. (From their point of view, it might seem a simple change.) In such cases, you should explain the costs involved and give them choices. You should say things such as: "Can that feature wait for version 2.0?", "If we do that, what other features can be dropped for now, so we can finish on time and under budget?", "Adding that will delay the project and cost more, is that okay?". (Saying "okay" then telling your programmers to work 20 hours a day until it can be shipped, is not okay! And this happened to me!)

Proper design helps eliminate many problems, such as missing proper checks for under- and overflow (and typecast truncation), data validation, ***graceful degradation*** (fail gracefully, not endangering life or corrupting data during a failure), or a non-accessible user interface.

All "production quality" software needs to meet several requirements beyond what you find in formal requirements documents. **Software should meet requirements for security, reliability, availability, fault tolerance, adaptability, human safety, interoperability, long-term evolvability, and trustworthiness.** The software should provide diagnostic ability (logs and assertions), testability, metrics production, and support audits (both security and compliance) and assurance evaluations. Enterprise applications should have contingency plans in place if dependent on data centers and networks operated by other organizations (i.e., the cloud).

> "There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies." — C.A.R. Hoare

Privacy and Data Protection Regulations

There are many legal and regulatory rules you need to follow to ensure the privacy and security of data. For example, the GDPR in the UK is a comprehensive privacy law. In the US, there is not a single privacy law, but many laws for specific types of data: Family Educational Rights and Privacy Act of 1974 (FERPA) for student education records, the Health Insurance Portability and Accountability Act of 1996 (HIPAA) for health-related data, the Children's Online Privacy Protection Act of 1998 (COPPA) for data related to children, and the Fair and Accurate Credit Transactions Act of 2003 (FACTA) for some financial data.

In addition, many states now have privacy laws. The California Online Privacy Protection Act (OPPA) of 2003 and the California Consumer Privacy Act of 2018

(CCPA) which became effective in 2020, affects people and organizations outside of California too. (See also this guide to the CCPA.)

Math Skills for Non-Entry Level Programmers — *Do in-class if time permits*

***Consider this common sort of design question*:** Suppose you are working on a new system for U.S. military veterans and you need to design a record format. What should be the minimum length of a String to use for an ID?

Assume there are about 15,000,000. Using the standard ASCII upper and lower case letters and digits and the eight special characters "!@#$%^&*", in order to create a unique ID for each one while allowing for as many in the future, and not make it easy for someone to guess a valid ID?

After reading the task, the developer should have a question: *how hard is "not make it easy for someone to guess a valid ID"?* That's pretty vague! Let's assume that "no more than 1 in a million possible IDs should be valid" is hard enough, according to the client.

Most entry-level programmers would be at a loss on how to figure this out. It takes some (what are considered basic) math skills. Here's how to work it out:

Let's start with a "back of the envelope" calculation: 2 times 15 million is 30 million IDs needed. How many binary bits are needed to give each one a unique number? Well, remember that $2^{10} \sim= 1{,}000$, so $30{,}000{,}000 \sim= 2^5 * 2^{20} \sim= 2^{25}$ individuals, so we need 2^{25} unique Strings. To obfuscate (make it hard to guess), we want a million additional non-ID's for each real ID. (That is, only one in a million potential ID strings is an actual ID.) $1{,}000{,}000 \sim= 2^{20}$ and so we need 20 extra bits in addition to the 25, 45 bits in all.

So we need 2^{45} possible Strings. The alphabet selected has 26+26+10+8 = 70 $\sim= 2^6$ characters, so each character is worth a little over 6 bits. That is, a one-character string suffices for 70 IDs, a two-character string suffices for $70^2 = 4{,}900$ strings, and so on. How many such characters are needed for 2^{45} ID strings? The answer is to simply divide the exponents: 45 divided by "a little over 6" is about 7, so strings of about 7 characters should about do it.

Check: To double-check such a calculation is always a good idea. Let's calculate the answer another way and see if we get about the same result:

30 million individuals times 1 million for "not too easy" equals 30 trillion, or $3X10^{13}$ potential ID strings, according to my calculator. As before, each character in the string provides for 70 IDs. Remembering that the $\log_2$ of some number is the "number of bits" for that number, we can divide "bits needed" by "bits provided per character" to see how many characters are required:

$$\log_2(3\ X10^{13})/\log_2(70) \sim= 7.3 \quad \text{(by calculator)}$$

That's in agreement with the "back-of-the-envelope" figure. To include a safety (fudge) factor, round up the value. The final answer is therefore 8-character Strings are needed.

(Follow-up question: If (G_d forbid) the number of IDs needed is much greater, what can be done? Answers: do nothing, making it easier to guess IDs; use a bigger alphabet (more symbols); or use a longer String.)

Top-Down Versus Object-Oriented:

Before OOP, software was designed by identifying the algorithms to be used. Next, the data types were chosen to make those algorithms easier to implement (and more efficient). The problem with this approach is that it doesn't scale up to very large programs; the resulting software is tightly coupled with low cohesion, making changes and bug fixes nearly impossible. Consider that a web browser might have 1,000 procedures (probably lots more), all of which have access to the same data. No wonder it would take a team months to years to produce anything!

If designed using OOD, you may end up with (say) a set of 100 classes, each of which has 10 methods. The 100 classes are implemented independently, and their objects only interact in a limited fashion. The resulting code has low coupling and high cohesion.

This object-orientation makes the larger problem understandable (you only need to understand a few classes at a time while coding). It also limits where to look for bugs; if the data in some object is wrong, you only need to examine that one class to see where the problem could possibly be.

Traditional, procedure-oriented design usually starts at the top, a method called *main* in most programming languages. Then you decompose the problem into smaller sub-problems, each of which can be further decomposed. Eventually, you should reach a sub-problem that is small enough that you know what algorithm to use for it. This is known as ***top-down design***. (There are other non-OOP approaches too, such as *data-driven development*.)

Design by Contract

This term was coined by Bertrand Meyer for the Eiffel programming language in the 1980s. The notion was based on earlier work by Tony Hoare, who in turn based his work on the earlier work by Robert W. Floyd. The basic idea is to ensure software does what we think it should, using a contract for each unit of code specified by preconditions, postconditions, and invariants. Today the notion lives on informally and developers still specify these things.

A *precondition* is something that should be true before invoking some code (in Java, some method). In a formal system such as Eiffel, these were

checked by the compiler and automatically at runtime. For informal systems such as Java, if a precondition is not met before calling some method, the behavior of the method becomes undefined (and might do anything).

In Java, preconditions can be checked by an if-statement in the caller (before invoking a method) or by the method (which should throw a suitable exception if the precondition is violated). Unit testing should ensure any method behaves acceptably even when a precondition is not met.

Usually, the check should be made by the caller. However, if the method developer considers trusting all users of their code to do so is too risky (critical code should not have any undefined behavior), or the design specifies the method will (for example) throw an exception for violated preconditions, then the developer must put the check in their (method) code.

Either way, the precondition should be documented in the doc comments.

postconditions are what a method says it will do. These can be checked with unit tests. These too should be documented, not just the return value but any side-effects too. (Note that methods with side effects are difficult to test, so avoid them as much as possible.)

Invariants are what a method guarantees won't change. Obviously, lots of things won't change but you need to list relevant stuff, such as the *state* of any objects passed as arguments. Again, these should be documented.

For non-public methods, you can do the above but more commonly in this case, assertions are used to ensure preconditions, postconditions, and invariants.

Basic OOP Concepts and Terminology:

With OOP, there is no real top. (While there is generally a *main* method, it is only used to create a bunch of objects, build the user interface (the "UI"), and initialize databases and network connections.) Instead, there is a bunch of objects talking to each other, in response to events. This takes some getting used to; where do you start program design? The answer is, **you must first decide what objects (and thus, what classes) are needed**. This is discussed further, below.

The main method of any real application should be so short and straight-forward that it needs no testing. Likely your main will call a method to parse any command-line arguments, environment variables, and other configuration data, and return configuration settings (as an object of some sort), call another method to configure logging and metrics (and return an error indication or OK so it can be tested easily), and call another method to start the application (set up a GUI or start a service). Main might also handle errors from these methods but maybe not. That's all that main should

ever do. All other methods should be testable: they should return something (a status if nothing else).

From the point of view of an OO designer, ***objects* are things that have *identity*, *behavior*, and *state***. A *class* describes a group of similar objects. Each object of that class has the same methods and the same fields (usually holding different data, or *state*) as every other. However, **no two objects have the same identity**, even if otherwise similar (e.g., in a purchasing system, two Order objects have different identity even if they are orders for the same items). An object's identity never changes for its lifetime.

The ***state*** is information that objects store in their *fields* (or properties, or attributes or instance variables or ...). This state allows object methods to remember the results of previous method calls, and to *cache* data that would be expensive to lookup or calculate each time. Thus **the required state is what is required to implement the object's methods**.

In (nearly) every design, **the state is private and accessible only to the methods**. In this way, **the exact implementation of state is not part of the public interface** to the object and can be changed as needed, without fear of breaking anything.

The ***behavior*** of an object is determined by the set of its `public` (and `protected`) methods and constructors. This is determined by the design. However, some methods determined by the design may be too complex to be easily implemented or tested, or may contain duplicated functionality with other methods. Such methods can be broken into several smaller `private` methods (similar to top-down design). Note that **only the public methods define the behavior**; the rest of the code is implementation detail that can be changed as needed.

Sometimes the best way to implement state and behavior is to define "helper classes" and use such objects to hold state information or to implement some behaviors. Such classes should not be visible outside the current package. In some cases, using private nested classes is a more convenient or efficient implementation.

Software Development Life Cycle (SDLC) and Workflows:

Software projects are developed in phases: identifying features (analysis and requirements), planning (high-level or architectural design), doing the actual development (researching, collaborating, producing a detailed design or *specification*, creating tests, coding, unit testing), committing the source code changes (code review), running builds and tests (unit tests again, plus integration, functional, and acceptance tests), deploying to production, and maintenance (and monitoring). The set of phases is known as the ***software development life cycle*** (SDLC) and is discussed below.

No matter your personal or team development process, all these phases (tasks) get done, just in different orders. The order they are done can vary and is called a ***workflow*** or *development methodology*. Some of these include the traditional waterfall workflow or newer agile methods such as Scrum and XP.

Historically, the phases didn't overlap; each phase had to be completed before the next could start. If the design or requirements changed, you started a new iteration of all the phases. This is known as the ***waterfall SDLC model***. While still useful and still used, especially for security critical and human-safety critical software, today the rules are generally relaxed somewhat; phases often overlap in time, and some phases can be repeated. There are other SDLC models in use as well.

> The ***agile*** workflows changed the way some software development teams worked (from the waterfall workflow). One of the key principles of this methodology is delivering working software frequently. The focus moved to releasing incremental software, rather than big waterfall releases which took months or years before teams attempted to integrate their software. (In a VCS, this agile workflow encourages trunk-based development (no long-lived branches).
>
> Agile workflows were over-hyped for years. They often fail (for some definition of fail); to succeed you need a very experienced project lead. Too many managers thought that agile means to *start coding now, figure out the requirements* later. As a result many agile projects fall behind deadlines and have large cost overruns, while never meeting the project goals. However, agile is a good workflow for *noodling-around* (experimenting) while learning.
>
> Some agile practitioners argue that you shouldn't implement general-purpose mechanisms right away; implement a minimal special-purpose mechanism to start with. Refactor into something more generic later, once you know that it's needed. Although that make some sense, it discourages investing development effort up front to have a cleaner, less-complex overall design.
>
> Many newer workflows have cropped up since the original agile workflow, including various version control workflows as part of the methods. Also, automating building, testing, and in some cases deploying are part of that too. Finally, although old, the waterfall workflow is still used successfully in many cases.

Modern software is much more complex than software developed before 2000. Often, hundreds of third-party software is included as dependencies. Such software has proven to be very insecure and fragile. To address this, modern software should (but too often doesn't) follow what is called the *secure SDLC*, a set of standards and best practices that address security in every phase of the

SDLC. (See Microsoft SSDLC and the EC Council for more information.) This includes keeping track of all libraries of code (and tools) used, often by generating a SBOM and examining all libraries for known security issues.

Critical components should not be some hobby project of a single programmer; who knows when or why development and support will stop. Either have an alternative available for such fragile dependencies or use open source and be prepared to take over future development in-house.

Regardless of your workflow, you need to set deadlines for each part of a project, called *milestones*. These are ***deliverables***. (Money often changes hands after each milestone, as do performance reviews.)

> In many real-life projects, milestone deadlines are set starting with a management mandated delivery date. However, there is a temptation to skimp on the initial analysis and design this way. *Always allow sufficient time for design and analysis*. If you don't, you won't make your deadlines anyway.

Software lifecycle stages — The Steps to Develop Software:

As discussed above, the activities of software design, development, and support (maintenance) are grouped into distinct stages or *phases*. In each phase the activities are different. Sometimes developers specialize in specific areas of development: quality-assurance engineers specialize in testing, front-end developers in user interfaces, back-end developers in database and related technology. Some specialize in security, networking, or monitoring.

> **A person who performs the *analysis* and prepares the *specification* (requirements) is often known as a *System Analysist*.** These two stages generally require different skills than general software development, including understanding of the problem domain (they speak with the clients in their own language) and knowledge of what is possible given modern technology, laws, and regulations.

While you don't need to be an expert in all these areas to develop software, you need to know a little about each of them to develop even a simple phone app.

These phases may be done in sequence, over-lap, or iteratively (where you go back to earlier phases again and again). Broadly speaking, you can define the first three phases as:

1 **Analysis** (*initial idea + overall design*) Here you need to determine what the software should do. The results of this should be the *requirements* or *specifications*. (Also called ***Software Requirements Specifications*** or **SRS**.) Initially you have a vague description (a *project proposal*, sometimes called a *RFC*, *RFP*, or RFQ), but you must determine complete and exact requirements,

including performance, cost, safety, compliance (and audit procedures), development methods (such as ISO-9000 compliant), and any technology requirements (e.g., Oracle SQL DB, Amazon cloud, etc.). **Remember that what is obvious to the VP of marketing that created the project proposal may not be obvious to you or other developers!** Two common ways of doing specifying the requirements (usually both are used) are:

- **Use cases (or scenarios)** A *use case* is a description of a sequence of actions to accomplish some task. It is like a storyboard, showing each step of the user interaction with the system (or any task flow). A set of use cases can fully describe the requirements of a system. This can also be called *task flow* or *workflow*. It is also known as a *business process*. Use cases can be used to describe either requirements, design, or other things. **UML** (described later) has standard icons and notations for GUI use cases, or you can use textual descriptions. (*Show samples*.)

  A related notion, ***user story***, is an *agile* development tool used mostly for project proposals and as a starting point for requirements. The story is written in natural language in a few sentences, describing in general terms some function the user wants the system to do. Later, each user story is used to generate one or more user cases, which detail the specific steps the system must do.

- **Functional (text) specification** A *functional specification* must fully describe all the terms and concepts needed, then using these, describe every task. It is difficult to make a functional specification that is complete, unambiguous, free from contradictions, and readable. (This is what lawyers are for!) However, text specifications are usually the best way to document non-functional requirements: performance and operational requirements (e.g., uptime, min. sustained load, burst loads, response time, failure modes, and so on); see below.

  Often, a combination of these techniques is used.

A part of the requirements, often overlooked, is ***technology requirements***. The customer/client may be assuming a certain type of server or middleware or EIS technology, or just assuming a hardware budget much too small for the application and performance requirements expected. For example, you need to make sure you and the customer/client understand and agree on the platform used (cloud, cluster, SAN, etc.), memory and disk space available, network bandwidth available, specific database product to use, etc. It pays to find out what products they already have and have local expertise with.

Operational requirements (also known as ***non-functional* requirements**) are also often overlooked. These are the requirements that make the system manageable and monitorable. These often apply to enterprise applications and rarely to small desktop or smartphone apps. Some of these "production quality" features include: proper and useful logging, plain text configuration files (so they can be easily versioned, searched, and compared), the ability to toggle on/off individual features and services (e.g., rich user interface vs plain interface, ad injection, read-write DB access vs read-only, and so on), access control and rate limits, graceful degradation, ability to deploy in a cluster (distributed or cloud computing requirements), queue draining, backup and restore of all data, and instrumented code that allows monitoring (and auditing) and the collection of performance data (metrics). (Such performance data include timing data and counters, such as a counter incremented at the top of certain methods.)

> Many, perhaps most open source projects are run informally. They don't necessarily fit or use standard methodologies and typically generate a lot less formal documentation. Instead of formal requirements, such projects use roadmaps (sometimes just private lists of features, with expected timelines for each), issue trackers (every feature becomes an issue), bug report lists, proposals, pull-requests, to-do lists, or just test suites (especially when following TDD).

2 **Design** In this phase of the SDLC you determine the overall system architecture, frameworks to use, and other decisions that must be made early such as testing framework, workflow, version control, and so on.

> Quite often the initial design of a complex software system does not cover every detail, just the overall design. Often no classes are designed at this time. This can also be called the ***system architecture***, *preliminary* design, *high-level design*, or sometimes the *system model*. A software engineer or senior developer may (for example) specify the architecture as a set of "modules" and their responsibilities and leave the details of designing individual modules/packages to others.
>
> See the excellent (and short) overview Software Architecture Guide and a list of popular *architectural design patterns* such as common software architectural patterns in a nutshell.

A good design should be as simple as possible (but no simpler), and extensible since requirements change over time. (It often pays to think about the more likely ***potential changes*** and create a design that would make implementing such changes cheap and easy, later.)

The design is sometimes called *logical* or *system modeling*. It is sometimes followed by another design sub-phase, *physical modeling*, which means roughly

the same as the data *implementation*. (Sometimes "physical model" is the name given to a *database schema* to be used with the software system.)

The ***detailed design*** is the one that specifies classes and their public interfaces. Here you figure out which objects are needed to fulfill the requirements. (You also design database schemas, file formats, and message formats (if any).

The detailed design process can be broken down into the following three steps, each comprises several sub-tasks and related concepts. Note however they are not done in this order. Rather you jump around the steps, repeating as necessary.

2.A **Identify the classes.** This really means identify the objects required. This often starts with looking for *nouns* in the requirements documents. In the end, some proposed classes will be trivial and end up as simple primitive types (or Strings), and thus as fields of other classes. Such simple classes are called *foundational* and generally have minimal methods (and no responsibilities). Some proposed classes have no behavior and are just database records. Try to minimize such classes; generally they lead to a bad design with one (or a few) "god" classes will all the responsibilities.

Each proposed class should have a single responsibility or a (very) small set of related and interdependent responsibilities. Only experience will help you pick the right classes.

Class discussion: enroll a student in a class. Which class gets that responsibility? Either your code will look like

```
someCourse.enroll( someStudent );
```

or:

```
someStudent.enroll( someCourse );
```

After finding the easier proposed classes, you need to think about the big picture (the system architecture). What else is needed to make the design complete? **You should mentally run through the use cases and make sure your design can fully support each one.**

Consider a voice-mail system. What are the obvious objects? (Users, messages, mailboxes, passwords, phone extensions.) What else? (Menus that can be altered over time, or perhaps multi-language menus, timestamps.) These are the obvious objects, but how will messages be stored in mailboxes? You need a *queue* (or FIFO), perhaps several (for new and saved messages). Here is a list of categories of classes; which do you need?

- **Things (real-world items, devices, systems)**. Whole systems (`VoiceMailSystem`) are convenient top-level classes that can contain a main and system initialization and shutdown code.

- **Actors, users, agents, and roles**. (Often have names ending in "or" or "er", e.g., `User`, `Administrator`, `Student`)
- **Events and transactions** (e.g., `Sale`, `Update`)
- **Collections** and other foundational (standard library) classes (`Set`, `String`, `Rectangle`, `Date`)

(*Show the list with examples in the course resources.*)

Name your classes with names that are descriptive of (an) object of that class: `Message`, not `Messages` (unless the object is a collection) or `MessageObject` or `Msg`. Avoid generic class names such as `User`, `Agent`, `Item`, `Event`, etc. If the obvious name turns out to be a verb (e.g., `BuildX`, `MakeX`) then you should rethink the design. (This can happen if the name is just a noun; the tip-off is if the class has a single "`doit`" method!) If the obvious name contains the word "and", you should rethink the design.

> "One of the tests I use for my program design ideas is whether the thing I'm thinking up has an obviously suitable name. Trouble thinking of a name for a class or method is a warning sign that it may not be a really good idea." *Patricia Shanahan, posted in comp.lang.java.programmer on 28 Jul 2011.*

Identifying classes is hard! With experience, you learn to **avoid having one "god-like" overly complex class plus a bunch of trivial classes**. Each class should do some of the work. (Qu: Why?)

2.B **Identify the responsibilities (behaviors) of the classes.**

Responsibilities are at a high level of abstraction. This means, don't try to determine the methods required at this point. A typical responsibility might be "Manage Passwords" rather than listing "read password" and "update password" methods. You find these by looking for the "verbs" in the requirements.

Responsibilities often fit into a natural ordering or layering (*level*) of abstraction; that is, some responsibilities are low-level (processing mouse-clicks), middle-level (Manage passwords) or high-level (initialize the whole system). **When assigning responsibilities to classes/objects you should make sure all the responsibilities are at the same level for any one class.**

Each responsibility becomes one or more public methods of the class; everything else discovered later is implementation details and should be private. When doing this step, you will often determine *state* or instance variables are needed in order to do the tasks you identified.

Classes should have neither too many nor too few responsibilities. A class with none is probably a *foundational* class and is too trivial to worry about. It becomes an attribute, for example a class `Point`, or a database record (also known as a *data transfer object*, or an *entity*). Such classes are needed, but generally only clutter a class diagram and are usually not shown.

All the responsibilities of a class should be related. If not, you should split the class into two or more simpler classes. Ideally, every class should have between 1 and 3 responsibilities. (One is ideal.) The degree to which all the parts of a class related to each other is known as ***cohesion***. One goal of your design is to have all classes with a high degree of cohesion each.

Never add responsibilities that aren't required, even if it seems that implementing them would be easy!

> A good way to measure cohesion is by checking how many instance variables are referenced (read or written) by the instance methods. If all methods reference all instance variables, your class is 100% cohesive. If you find some subset of methods that reference a subset of instance variables, you probably have two different abstractions in that one class.

(There is also *method-level cohesion* which means all parts of a method are related.)

In practice there are many ways to group methods: functional (all methods relate to the same task), informational (all methods relate to the same data), temporal (all methods are done at the same time such as initialization/cleanup, or done in some specific order), and others. Functional cohesion is usually best, but be careful it doesn't lead to many one-method classes.

2.C **Identify the relationships between the objects.** A bunch of independent objects that do not interact would make design, implementation, and testing very simple, but the resulting system would not be very useful. The degree in which different objects depend on each other is known as ***coupling***. A goal of your design should be to minimize coupling, and to use standard design patterns when you have two or more objects interact.

> Loose coupling is also called the *Law of Demeter* (LoD), or the *Principle of Least Knowledge*.

In general, we say that interacting objects are *collaborating* and have an *association* or *dependency* or *link* between them. In a class diagram, you shown these associations with lines between the classes. There are many types of association. Three common types are:

C.i **Dependency**: (commonly called the ***uses-a*** relationship) Class A *depends on* (or *uses*) Class B if objects of class A examine or manipulate objects of class B.

For example, consider a purchasing system with Account, Order, and Item classes. An Order object may invoke an Account object's methods (say, when ready to check out and place the order). I can imagine a method call such as "`Order.addToAccount(customer);`" from the Order object's checkout button event handler. In this case, Order uses-a (has a dependency on) Account. Of course, it is possible to have Account objects invoke methods of Order objects too. (Think about a *display recent orders* button.) So there could be a uses-a relationship in either or both directions. But in this example, Item objects have no relationship with Account objects.

The lack of any relationship between class Account and class Item means these classes are completely independent. They can be designed and built independently. Any bugs in one, even a design flaw that requires one to change the public methods, won't have any effect on the other class. Even changing the public interface of one won't affect the other. This is very desirable, and is known as *loose coupling* (discussed further, below).

Typically, the uses-a association is transitory, with a method of a class A object invoked with an object of B as an argument.

Consider a voice-mail `Message` class with the responsibility of playing itself. One design of a method to play the message could be:

```
public class Message {
   private AudioClip message;
   public void play ( Device dev ) {
      dev.output( message );
}  }
```

This design has the `Message`, `Device`, and `AudioClip` classes coupled, and we can say that a `Message` *uses-a* `Device`. (And if you look carefully, you'll see that `Device` *uses-a* `AudioClip` too.)

A good design will minimize coupling and thus minimize dependency. What if we give the responsibility of playing the message to a `Device` object? The revised `Message` class might then look like this:

```
public class Message {
   private AudioClip message;
   public AudioClip getMessage() {
      return message;
}  }
```

Both designs would work, but the second has less coupling and would thus be a better design. (Qu: What would the `Device.play` method look like?) Note that unless the `Message` class has some additional responsibilities not shown here, the class becomes a useless wrapper around an `AudioClip`.

C.ii **Aggregation**: (the ***has-a*** relationship) This is the association when one object contains objects of some class over a period of time longer than a single method call. (We're not talking about parameters to methods or local variables.) A `MessageQueue` object could be said to *has-a* `Message` object(s). Think MANAGES-A or OWNS-A for aggregation.

Not all instance fields of a class correspond to aggregation. A single instance of some foundational class (`String`, `LocalDate`) is really just an attribute of some class, not an association between classes. This distinction is very subjective, but if in doubt, prefer an attribute to an aggregation.

> In some cases, you can distinguish between regular aggregation and ***composition*** ("***is-a-part-of***"). If some object X is created and destroyed when the containing object Y is, and is never referred to by other objects, the relationship between X and Y is composition. (Examples: "titlebar is-a-part-of a window" is composition, but "doctor has-a patient" is aggregation.) In Java, the implementation of both types of aggregation is likely to be the same.

C.iii **Inheritance**: (the ***is-a*** relationship.) This is actually an association between classes, not individual objects, and has been discussed previously.

With either dependency or aggregation, the association has a property called ***multiplicity***. This can be 1:1 (one-to-one: one student object has one ID number) or 1:*n* (one-to-many, as in one `MessageQueue` has up to *n* messages). A 1:* multiplicity means an unlimited number (or none at all). A multiplicity of 1:0..1 would mean either zero or one; the zero is often represented by setting the instance variable to `null`. (Aside: Using wrappers such as `Integer` instead of primitives such as `int` allows you to

distinguish between not having the attribute (`null`) and having it with a value of zero. For designs where null values are problematic, you can use class `Optional` as a wrapper.)

> Consider the classic illustration of modeling books and authors. A given author can write multiple books. That would be a 1:*, however books can have multiple authors (which would be a *:1). But suppose a set of authors work on several books together? The true relationship between books and authors is *:* (many-to-many).

Which design is best? Often there are many possible sets of classes that would appear to fulfill the requirements. Rather than go with your first design, spend a little time thinking of the alternatives. In the end it may not matter which design you chose, but often one design is simpler, more efficient, or more extendable then another. (Another school of thought, *extreme programming* or "XP", says the opposite: Just do the minimum now, and add only as needed.)

CRC cards: On the top of a 3x5 index card, write the proposed class' name. Down the left side, list the responsibilities, and on the right list the other classes that collaborate with this one. Make new cards, rip up old ones, add and cross-out responsibilities and collaborators freely. (**Show CRC resource.**)

(Design patterns and UML diagrams are discussed later.)

3 **Implementation** This phase starts off with building the "***stubs***" or *skeleton* of all identified classes and methods. An IDE such as Eclipse has wizards to do this for you.

> ***Stubs*** are empty methods, except that methods that return values must have a `return` statement (that returns some fake but realistic value). A ***skeleton class*** is one with only stub methods and constructors.
>
> A skeleton class can in fact be compiled. As you implement classes and methods one at a time, this skeleton becomes a ***prototype***, a not quite production-ready version of the project. The prototype is expanded and filled in until the whole project is complete. This works well for small and medium sized projects, but more formal implementation processes (with testing and design and code reviews) are needed for larger ones.

It sometimes pays to '**do one design to throw away**", just to gain insight into the problem and possibly to tweak the design before building the production (non-prototype) version. (You can gain insight also by trying to implement

some methods.) This is an exception to the rule that says always to design before implementing.

> Many security errors are done with naive implementation practices. For example, Microsoft reports the use of uninitialized memory (which may hold sensitive date from whatever app used it last) and other memory-related bugs account for 70% of all patches Microsoft shipped in the decade ending 2020.
>
> This is not an issue with some languages such as Java, but other memory issues (failure to normalize, sanitize, and validate; use of share memory with multiple threads, etc.) are.

You should schedule (plan) the implementation steps. It is common to implement methods in an order that facilitates testing, or for use by other developers working in parallel with you. You often build some methods and classes now and defer the others until later. The order may depend on developers' schedules or the customer's priorities.

For large projects, the public interface of classes is decided early in the design and interfaces are created. Other groups can implement their own skeleton classes and mock objects (for testing) from the interface. Having methods whose parameters are interfaces is much easier than having method parameters depend on specific classes! Thus, you often see interfaces with a single class implementing them.

Don't make the (beginner's) mistake of implementing the "easy" responsibilities first and saving the "hard" ones until later! **Start with the hard ones; they are hard because the design may not be complete or well structured, and you may have to go back and change the design to fix that**!

> A design that leads to classes and methods that can't be implemented (while meeting performance requirements and project deadlines) or tested easily is a bad design.

When implementing code using an IDE such as eclipse, it can automatically write much of the code for you. Common examples include having the IDE generate all the get and set methods for the properties of a class. Even once the code is written a good IDE can help re-work the code, a process called ***refactoring***.

As discussed previously (early in the Java 1 course), *large language models* (**LLMs**), a type of so-called artificial intelligence, can also write starter code and test cases. LLMs are statistical engines that show the most likely response to your prompt (query). What they show is based on what code they saw (were trained on) from the Internet. Often the quality of Internet available code is poor, or your specific query had little training material available. Worse, the systems may mix different programs it saw, resulting in garbage

(hallucinations). For some problems, LLMs can provide good starting code, for other problems, not so much. In any case, you must carefully review and validate the code it provides.

> To some extent, each of the software project phases is independent of the later ones. In general, you shouldn't worry about the design when doing the analysis, or the implementation when doing the design. In reality, the design will be influenced by the implementation choices available to you, and the requirements will be influenced by design considerations. (If not, you often end up with unrealistic requirements, or requirements that would cost a lot more than they should.)

Don't try to create clever code (also known as *hacks*). Remember the homicidal maniac that will have to maintain the code after you write it! Compilers these days are very good about optimizing straight-forward code, so clarity is most important.

Bad design has lots of complexity, some of which gets added over time as the code is updated to meet new requirements and to add new features. A complex design is bad! Some indicators of a bad, complex design are:

- Seemingly simple changes require code modifications throughout the code.
- A developer needs to know a lot (too much) in order to make any change.
- It's not obvious where code must be modified to implement some change.
- Exceptions must be handled in a lot of different places.

As Brian Kernighan put it: "*Debugging is [at least] twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.*"

Test-Driven Development (TDD)

Before starting the implementation, the *extreme programming* method (and other *agile* methodologies such as *rapid prototyping* and *Scrum*) would have you write a set of test cases that "cover" the design. This means there should be enough test cases to test every (public) method of every class in your design, except perhaps for the trivial getter/setter (accessor/mutator) methods. The test cases should also cover the interactions between classes. This is sometimes called *test-driven development* (TDD). (Note, if you later *refactor* your public method to invoke private ones, the tests don't change; there is no need to test the private methods directly.)

> Scrum was popular, but less so as of 2025. See more at Scrum.org and ScrumGuides.org.

With TDD and agile methods, you typically decide to work on one feature or one task at a time. (A task may involve several related features. You still work on features one

at a time.) Then you write one or a couple of test cases for each feature. As you do so, you are deciding on the design (names, arguments, and number of methods). If writing the test is difficult, you try a different design. As you refine your design, add another test case or more; two or three per feature is normal. These are typically all happy path tests, but safety-critical code may need other tests up front as well.

When done with that, your test cases won't compile, since the classes and methods they invoke don't exist yet! So you create skeleton classes and stubs for those methods. Now your test code (and the skeleton classes) compile, but all the tests will likely fail.

> With TDD, you write tests only a few seconds or minutes before writing the code. Initially, you often only will have few tests, but after writing those, you should pause and consider if additional tests would be useful (security, boundary-value, failure tests, overflow (and typecast truncation, ...) As you add details to the code, you repeat TDD by always writing new tests, then new or improved code.

Before proceeding, reflect on the design. Is it complete? Does it comply with everything it should? Is the API easy to use and understand? If not, this is the time to address that. Also, it should be easily testable, with short, single-purpose methods that are ideally idempotent. Make sure the design meets any non-functional requirements (security and safety, and performance).

> TDD is a great way to explore an unfamiliar library (API). It is also a great fit when designing APIs for libraries and when using the *microservices* architecture. It doesn't work as well for traditional application development.

Once you think the API you designed is good, you implement the methods until all tests pass. Then you consider making the code better (*refactoring* the code). **Do not skip this step!** This includes instrumenting the code: handling errors such as nulls and overflow (and typecast truncation), expression rewriting, adding logging and metric collection statements, assertions, and any useful feature configuration a.k.a. *feature flags*). Each time you compile the changes, you also rerun all the unit tests.

> It is not uncommon that after several features have been implemented, you'll notice the design is wrong. One almost never gets it right the first time! It is also possible to work on features in the wrong order, as you sometimes see one feature may depend on another that you haven't thought about yet. With experience, you get the design and feature order right sooner, with fewer false starts.

Here's a trivial example: Suppose you have a class Rectangle with a method area:

```
public class Rectangle {
    final int width, height;
```

```
    public Rectangle (int width, int height) {
        this.width = width; this.height = height;
    }
    public int area () {
        return width * height;
    }
}
```

Now suppose a new feature has been requested: a boolean method that determines if one Rectangle could completely fit inside another. To work on this new feature, start with new test cases. Let's call the new method "`canContain`":

```
public class RectangleTests {
    final Rectangle r1 = new Rectangle(30, 50);
    final Rectangle r2 = new Rectangle(10, 40);
    final Rectangle r3 = new Rectangle(60, 45);
    ... // existing tests
    @test void fitsWithin() {
        assertTrue( r1.canContain(r2) );
    }
    @test void doesNotFitWithin() {
        assertFalse( r1.canContain(r3) );
    }
}
```

Initially this will not compile. Add a stub method to class Rectangle:

```
    boolean canContain( Rectangle other) {
        return false;  // Stub
    }
```

Now your test suite will compile, and the older (regression) tests should pass. However, the new test cases do not pass. Before proceeding, *reflect* on the design of the new feature: is it clear? Is this the right API for this feature? Does the class need refactoring? Are additional test cases needed to make it clear how the new feature is supposed to work? Notice how writing tests first makes you design the API first, a good thing.

Assume you are fine with this design. Now implement the feature until the tests all pass. When done, reflect once again: Any alternate implementation to consider? Met safety and security needs? Need to add instrumentation to the code (logging, metrics, etc.)? Are any additional test cases needed?

> When working, personally I start a feature branch and then do a git commit after updating the test suite, after adding the stub, and several times when implementing the feature (if it were more complex than a few lines of code).

> Before code review, I would squash all the implementation commits in the branch into a single commit, then generate a pull-request making code review easier. Note often you would use a feature branch for this but that depends on your workflow.

So as not to leave you hanging:

```
boolean canContain( Rectangle other) {
   return (width > other.width && height > other.height)
      || (height > other.width && width > other.height);
   }
```

Over time, you will need to add additional tests; happy path testing is almost never enough! However, it would be very unusual to try to completely test the system with a full suite of boundary tests, illegal data tests, and so on. Usually, you would add only a couple of such tests (if any) at this stage. As bugs are discovered, or as requirements change, you then go and add more test cases. That said, high-stakes software such as for safety-critical systems or banking software must be more thoroughly tested than some cell phone game app.

> In an effort to make testing more understandable, a refinement of TDD was developed, called *behavior driven development* (BDD). There's a nice summary article for BDD on Wikipedia.

Large Project Design — High-level (Architectural) and Low-level (detailed) Design

A problem with large projects is that you cannot oversee all interactions between all components. It is thus important to reduce the complexity of the project. Class and Sequence diagrams are too detailed for the initial phase of design. This is where you use ***architectural patterns***. These are different from design patterns; architectural patterns structure the entire system at a high level, while design patterns structure units of code to solve a specific problem using best practices. Examples of architectural patterns include 2-tiered, 3-tiered, event-driven, micro-services, and others. Design patterns include builder, observer, facade, factory, decorator, and others.

> There is some overlap between different patterns and it isn't always clear which one is best for a given situation, especially at an early stage of a project. In addition, some patterns can be considered either as architectural or design patterns, for example MVC.

First try to think from a higher abstraction level. Think about major modules and their responsibilities (their interface to other modules), look at some architectural patterns for inspiration (not design patterns, these are too low level! MVC and Multi-Tier are architectural pattern examples). A reasonably large project might have 3 to 5 modules (or levels or tiers). In some cases, each of these components

has a specified interface, also documented here. A typical enterprise app might have a presentation (or *view*) tier (which uses MVC), a persistence tier (which uses JPA or SQL or something else), and the business logic tier (which provides services to the presentation tier and uses the persistence tier).

Each of these modules/tiers/layers is then subject to detailed design, where you express your design patterns and have class, sequence, and other diagrams. The collection of these is the detailed design document.

A high-level design document describes the architecture used in the development of a particular software product. It usually includes a diagram that depicts the envisioned structure of the software system. Since this is a high-level document, non-technical language is often used. An example might be "Uses client-server model with a relational database" or "Will use MVC design pattern for the presentation layer".

From StackOverflow: "High-Level Design (HLD) involves decomposing a system into modules, and representing the interfaces & invocation relationships among modules. An HLD is referred to as software architecture."

The detailed design is the implementation roadmap and includes all the details. Also called the low-level design (LLD), this is used to design internals of the individual modules identified during HLD, i.e., classes, data structures, and algorithms of the modules are designed and documented. So it will include class diagrams with all the methods and relations between classes. The LLD describes modules mentioned in the high-level design in sufficient detail so that the programmer can directly code the program based on it.

The detailed design might include the logic of the module in pseudo code, all database tables with all elements including their type and size, and all interface details with complete API references. It might contain other things too, such as error message listings (and a discussion of failure modes) or GUI appearance.

These documents come from design methodologies which predate object-oriented design, but are still in wide use. With object-oriented design, the high-level design is just the overall system architecture, such as "3-tier design using MVC for the view layer" or "using RESTful microservices" or "desktop containerized application". The detailed design is everything else: class diagrams, other diagrams as needed such as sequence or protocol diagrams, failure modes, persistence details (such as the database schema or exact file formats), and so on.

In fact, with object-oriented design, you don't use the terms high-level design or low-level design, and in the end, there might be only one document named design (after requirements), although in that case it is a living document that changes as the design is finalized (see *test-driven development*). This is because some design decisions should be postponed until after some other details are finalized.

Not included in the detailed design would be pseudocode for each method or function. But the method/function signature (the interface) should be there, along with any notes an implementer would need to do the job right. Such notes can include anything useful, such as what metrics to collect, inputs and outputs of modules (packages in Java) because data crossing module boundaries must be normalized and sanitized or validated, and often logged, algorithms that should be used in the implementation, and so on.

In short, with object-oriented design it's a bit arbitrary what info goes into which document (high- or low-level design), and often only a single design document is ever produced (which is updated as you develop and design your classes). In the end, armed with just the detailed design a junior programmer should be able to implement some class or even a whole module of code and have it work with the rest of the system. No details can be missing at that point.

Testing

While modern software development includes testing as part of implementation, there may be testing done only after implementation is complete. This may include acceptance tests and expensive code audits (you only want to do those once). Testing was discussed in detail, earlier.

Delivery

In this step your code is built (and sometimes digitally signed) and sent to where it can be deployed in production. The version of the set of Jars and other files that is delivered is often called a *release*. The Maven-release-plugin is useful to create releases of your code. It will unit test, update POMs, commit the changed POMs, then build and package your code, add a tag to the Git commit of the code used, and finally run other Maven tasks (such as system tests, deploy, site, and site-deploy). There are other plugins for this job as well, but this one works well for most projects.

So you've tested and built your Jars and executables. Now what? Typically, the final released objects are known as ***artifacts***. These artifacts are stored in a repository of such artifacts as the final step of delivery. The artifacts may be stored separately or in a zip or other type of archive file.

As you may remember, Maven creates such a repository on your local PC in your home directory, with the name "`.m2`". But artifacts published there cannot be easily used by other developers. You generally need to store artifacts in an organization-wide repository. Nexus is popular free repository management software (there's a commercial version too) used for this purpose (there are lots of these on the market). It works much like Maven central, providing version management and dependency management features.

Update: Java 8u20 added support for deployment. Using the JDK tool **`javapackager`**, you can create a self-contained application bundle that can be just run. For more complex cases, you can automatically convert such a bundle into a platform specific executable installer, for Windows (.exe, .msi), Linux (.rpm, .deb), and Max OS X (.dmg, .pkg). See the article self-contained packaging and the JDK documentation for the javapackager tool for more information.

Deployment

Deployment has been discussed previously using Jars. Jakarta EE deployment will be discussed later, but they are similar: bundle your code with the correct additional configuration (usually XML) and other files, sign the Jars, and upload to the production servers. If you deploy a service on your own servers, code signing is not necessary and rarely done. Note configuration files are present from delivery, but will need updating for current IP addresses and other information that is determined at deployment time.

A common issue with deployment of complex applications is that they require multiple Jar files. The problem is that the standard Java SE ClassLoader can't load classes from Jars within Jars. There are several ways around that issue (such as using Maven *shade*), but the easiest is probably to deploy your app in a system-specific installer or a Zip archive. (See the Java 8 deployment guide for details.)

Deployment to production servers can be tricky. Often monitoring systems used by system administrators (or site-reliability engineers) need updating at the same moment. Log files will need *rotating* (closing the old files and starting new ones). If the database schema has changed, that will need to be coordinated with the deployment of the new version of the system. If there's a cluster, you need to carefully update the servers one at a time, cutting-over users slowly. These are all considered operational issues. Fortunately, deployment these days can be automated. (See *continuous deployment*, below.)

Maintenance

Maintenance is the most expensive part of most software projects. It takes the most time. Over the life of the software system, bugs will be found and enhancements requested and implemented. Each change makes the code deviate a little more from the original design. Over time, the resulting "evolved" code gets harder and harder to maintain. Eventually a point is reached when it doesn't pay to maintain it or enhance it anymore, and support is discontinued. (We say it has reached its ***end of life***, or has been "**EOLed**".)

Having developers aware of operational issues and requirements, having system administrators aware of developer time (and other) constraints, and

having both groups work closely together, is known as "devops" (discussed elsewhere).

Entry level positions for programmers often have them deal with this maintenance as their main job. You sign into some ***trouble-ticketing*** (or *issue tracking* or *software project management* or ...) system and work on whatever tickets have been assigned to you. (In some cases, programmers can see what tickets have been entered into the system and choose which one(s) to "own" and work on; this requires you to know how to manage your time and how to prioritize.) Popular software for this include the excellent commercial product JIRA (which has IDE support), and some open source (and free) products such as Bugzilla and Trac.

Delivery, deployment, and maintenance are often referred to as operational issues, but today developers share such responsibilities with system administrations. As already mentioned, this is called *devops*.

Known Error Log

Once the root cause of some incident is known, unless it is fixed immediately it should be entered in the "**known errors database**" (KEDB), also called the ***known errors log***. Such errors should have minor customer impact, or known work-arounds. You list the problem and any work-arounds for it here, along with the date and time the issue was present if known, or the date and time it was detected otherwise. References this issue and any related issues in the issue tracker are included.

Once the problem is fixed, the entry can be deleted, but ideally just the date it was fixed (and the fixed version numbers of effected software) are included in an updated entry.

The known error log is often overlooked in the US but is mandated by the ITIL, COBIT, ISO 20000, and UK law. In other words, you really should have one. It should be created when the software or service initially goes live (is first released). It is vital because all complex systems will have problems, no matter how carefully designed or how through the testing and code reviews.

You can find more information about known errors at cio-wiki.org and www.quickstart.com/blog.

Project Management

Although ticketing systems are the most commonly used, an alternative is using ***Kanban***. Kanban boards are a simple way to organize work, provide a focus on getting a set amount of work done each week (as opposed to ticketing systems), and provide better indications of the current state of work. Some popular Kanban tools include Trello and Leankit (commercial), Taiga and KanBoard (FOSS).

GitHub has this too, under a repo's *Projects* tab (see demo). See LeanKit for some example Kanban boards and learning resources.

(Both can be used. Kanban boards work best for project work, and ticketing systems work best for bug and issue tracking.)

[*From an article in Communications of the ACM Feb. 2011 by Thomas Limoncelli*.] Creating software that can be managed, deployed easily, and updated, should be among the design goals and requirements, but usually isn't. Keeping in mind applications may need to be deployed on hundreds of PCs and enterprise applications on large clusters, try to ensure your product:

- Has a silent install (non-interactive mode with all custom options settable from the command line or from a file);
- Has a non-GUI admin interface. System admins need to be able to script actions to make them repeatable.
- Uses text-based configuration ("config") files. Don't require a Windows registry setting and don't use a binary format. This allows system admins to examine, update, and "diff" configurations using standard tools.
- Provides a management interface that not only says if the (enterprise) application is up or down, but also bandwidth/throughput, memory used, and CPU utilization.
- Have a mailing list, RSS feed, or website to report security issues, bugs, and patch availability. (And report security issues responsibly as soon as known to the outside world, even before you have a patch available, so system admins can monitor the application for intrusions.)
- Uses standard logging systems (that interface to the OS standard logs, syslog for Unix/Linux and Event logs for Windows. (What to log and how is discussed in a later section.) Also should provide appropriate metrics, traces, etc.
- Keeps all its files in one (or a few standard) directories. (Don't put DLLs in `C:\Windows` or a dozen other locations!)
- Provides documentation on-line. And not just for the currently supported releases but all versions! (The disaster recovery procedure for some 5-year-old unsupported installation might depend on being able to find documentation for that version.)
- Have a testable configuration setting that can be changed to dynamically enable and disable all individual features.

Code Reviews

Code reviews are an important part of the software development process. A programmer must *self-review* their code. In a team, usually the entire team reviews the code of each team member, known as *peer review*. Peer review can be conducted formally; however informal code reviews are usually as effective and take far less time. Open source software may be reviewed by anyone; usually, a website contains some support for recording code review results as issues to be addressed by the developers. (GitHub supports this; try it out with your team!) Today (2025), you can also ask LLMs to review your code. (Opinions are divided however, on how useful that is.)

Code may be reviewed by consultants or other experts, to provide assurances that the code is secure, accessible, and generally in compliance with all relevant standards, laws, license terms, regulations, and organizational policies. Such reviews are usually referred to as ***code audits***. While audits may be conducted internally (to the organization), users and customers are (sometimes rightly) suspicious of such. Indeed, some audits are legally useful only if conducted by a proper external auditor. For example, an SOC 2 audit. Security audits, ISO 9000 audits, and other audits are often external audits. However, external audits can be very expensive and are not always useful or required. License compliance audits are among those usually conducted internally.

> Microsoft published some research that suggests code reviews do not find any bugs (and thus are a waste of time). But that study was flawed; see Code Reviews Do Find Bugs for a thoughtful response.

Code reviews can be conducted at any phase of software development, but often don't happen until after much or all of the code has been developed. This is **not** a good idea; code usually can be reviewed in pieces, as soon as each piece or module has been implemented. Well-run code reviews, especially peer reviews, can greatly reduce the defect rate and simplify software testing.

All code should be reviewed before pushing to the main branch. A simple way is to create a *pull request* from your repo that has the changes, in GitHub (see GitHub's Using Pull Requests). Such changes are also called a *patch*. Tools such as Jenkins CI can be used to enforce code reviews before any push to the main branch.

Other team members can review the pull request, make suggestions, or ask questions. If necessary, the author can apply the suggestions and make a new pull request (deleting the earlier one). That too needs a review. Once a pull request has been reviewed and approved by all team members, it can be applied to the main branch. The discussions can take place using a wiki or another tool. This is usually preferred to face to face code reviews, as all comments are automatically recorded.

Pull requests need to be small enough that humans can and will look at them, a few hundred lines should be a maximum per patch; aim for 1-2 dozen per patch. When creating one, make sure you clearly document the intent of the patch. (Several smaller patches are better than one large one!)

Another way to achieve this is to use software designed for this purpose, such as Gerrit (discussed below).

> Another possible workflow is to have some person responsible for quality control. That person would be the only one who can commit to the main branch. A tool such as Gerrit can be configured to require certain people's approval, or a majority, or just one other person's approval.

When reviewing another person's patch, you should be looking for some specific things:

- Does the patch really fix a bug or add a feature that is needed? Is it doing it the right way? (If something is broken, adding an `if` statement to code around the bug, rather than hunt down the source of the bug and eradicate it, is the wrong approach.)
- Does the patch do what it is supposed to? Note that not all patches are for addressing bugs or adding features. Sometimes a patch's purpose is to refactor the code to improve performance, or just to improve the code (for some definition of "better".)
- Is the code quality good? (Variable names, Java doc comments, other comments, using an appropriate design, etc.)
- Does the code meet the style requirements? Note, there are many tools that can automate this check; a better approach!
- Were all appropriate unit tests created and added to the test suite? (One way to verify this is using a code coverage metric.) Does the testing code look correct, simple, and clear?
- Does the patch enforce code safety as appropriate? This includes managing over and under flow, exception handling as appropriate, normalizing, sanitizing, and validating untrusted data as appropriate, and properly encoding data before sending to another component.
- Does the patch cause any security issues? These include correct handling of sensitive data, logging access and modification to such data, etc.
- Are changes to logging, metrics, assertions, documentation, operational procedures, or audit updates required?

There are any number of "code review checklists" available on the web. (A few are listed in the class resources.) When starting out with code reviews, using such a checklist is a useful guide. Note there are different checklists for "security" code reviews, as opposed to normal code reviews.

Some **excellent tips on code review** can be found at IBM Developerworks (based on a Cisco study). Some of these are:

- Aim to review no more that 300-500 LOC (*lines of code*) an hour;
- Don't try to review code for longer than 90 minutes at a time;
- Even if you don't have time to fully review some patch, some review is much better than none at all;
- The author of a patch should do the first review (before submitting to the team) and add notes about what the code is doing and why (should have been in the code comments all along);
- Test all changes actually work and didn't break what was already working.

When reviewing code, be sure to look at the commit history and comments. Example of a good commit comment would be: "*Added off-line storage in files, as requested in issue #123", "The header now includes the filename used", and "If the user didn't set a name in the config file, a unique name is automatically generated".* On the other hand, a commit message such as "*fixed a bug*" or "*improved code*" is not a good sign (although a very few of those may be acceptable).

Some other ways to evaluate code would be to compile the code with all warnings enabled and in a strict-compliance mode. See how many errors and warnings you get. If the code includes any unit tests (and it should), read those to see what the code should be doing. Run the unit tests and make sure the code works. Also use a coverage tool to verify the unit tests cover most of the code. If not, the tests are bad or the code is bloated or both. You can run style checking tools against the source but understand that accepted styles change over time; it may be the authors just used a different style than your tool wants.

> Don't forget to check the license terms for the source code; make sure it can be used for its intended purpose. This is especially important when using out-sourced code. Some good tips include asking the authors for the commit logs and using regular expressions to search for signs of well-known APIs.

Finally, note that code reviews are not insults to the developer. **You must never take any criticism personally!** Nobody writes flawless code, and others will always have opinions over the details and style of code. Try to be grateful for the feedback others have taken the time and effort to provide.

When offering criticisms, try to state your opinions as inoffensively possible. Instead of "you should..." or "you were supposed to...", say "I think this might be better if..." or "what do you think about changing this to...".

A somewhat interesting Java project you can practice code review on is Google ExoPlayer. Take some time to read the README and developer docs, look as one or more source code files, and practice reviewing a pull request such as PR #8178.

Reproducible Builds

"Reproducible builds are a set of software development practices that create a verifiable path from human readable source code to the binary code used by computers." — reproducible-builds.org

Your code is polished, tested, reviewed by your team and approved. What now? The next steps are to build the software, then the result is deployed. These steps used to be bottlenecks, since developers used to hand a zip or equivalent of the code to system administrators, with verbal directions on how to compile the code. Naturally that would fail and everyone needed to be interrupted to work on the issue. As part of DevOps, the concept of *reproducible* and *automatic* builds became mainstream.

This is important since you may (will!) have to redeploy old-but-updated software someday. If the tools, versions of DLLs, the OS, the web, database, or other servers the code depends on have been updated since then, rebuilding the code may (will!) fail. Thus, as part of the build process **the build environment must be recorded in detail and kept**. A year or more later a virtual machine (or container) can be made with an *identical* environment and the code built correctly. To ensure the build environment is correct, first recompile the old version and confirm it is identical with the deployed version. Only then make changes and build the new version.

To have reproducible builds, you need a fully automated build system. (A manual process would be error-prone and time-consuming.) Once the developers finish with code and it is time for someone to deliver it, it should only take a single command. (Deployment can also be automated, if desired.)

For the compilation process to be deterministic, the input to the compiler must be the same regardless of the build environment used. This typically involves normalizing stuff that may change, such as order of input files, timestamps on files, locales, pathnames, UUIDs, ephemeral port numbers, and no doubt other random data. However, it is a bad practice to hard code such data in the first place. (Using a container to build stuff makes this easy.) Also, some randomization done by compilers such as ASLR is a security practice best left in place for production builds.

Also, the compiler must not introduce non-determinism either! This sometimes happens when using hash tables with a random hash seed value. It can also happen when using the address of variables because that varies from address space layout randomization (ASLR).

Build systems such as Bazel, Gitian or GitHub Actions, can be used to automate deterministic build processes.

Timestamps are likely the biggest source of reproducibility issues. Many build tools record the current date and time some binary was built. it is better to use a modification timestamp from the most recently modified source code file used in the build, instead of the time of the build. There are several ways to modify build processes to do this: Set the `SOURCE_DATE_EPOCH` environment variable to the number of seconds since January 1, 1970, using something from the source code. Some build tools that support this environment variable will use its value (when set) instead of the current date and time. Another way is to post-process the built artifacts to remove timestamps, or normalize them. The Linux tool strip-nondeterminism can often help do this.

With the technique known as ***continuous deployment***, you don't even run a command! A CI (*continuous integration*; see below) server such as Jenkins watches the software repositories for approved updated code, and automatically builds, tests, and deploys it.

> The software used for builds is also kept under version control, tested, and reproducibly deployed. Your artifact repository will contain the correct versions of the JDK, all libraries used, the correct versions of JUnit, Maven/Gradle, Git, etc. The source code repository will keep track of all versions and configurations used.
>
> To reproduce an old build, a virtual machine (VM) or container is built (with the correct operating system version, of course), and all the correct versions of tools are copied into it, along with the source code. That can now be built exactly the same as originally, and once you verify that, you can then make changes to that code and rebuild.
>
> Keeping all the required information under source code control (and all artifacts needed in an appropriate repo) is known as *code as infrastructure*. This is necessary for reproducible builds.

Continuous Integration

A modern trend is ***continuous integration***, or "**CI**". "Continuous Integration (CI) is a software development practice where members of a team integrate their work frequently; usually each person integrates at least daily — leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible." [Martin Fowler]

With CI, software is automatically rebuilt and tested. The build can be run automaticly. It can be triggered by time (say every day), or more commonly, by an event; usually, a commit to a project's (main) repo will trigger the CI to execute a script. Such a script can do anything, but usually runs Maven or another build system to build and test. The results of the build and testing are sent to your code review tools (e.g., TeamReview or Gerrit), or to a *chatbot*. (A lot of developers

use a chatroom for group communications, and the chatbot notifies them when something interesting occurred, such as a new version of the code was built.)

Additional features of such a script include alerting the submitter about errors, updating websites, producing reports, etc. A CI server can also monitor deployed applications (servers), and alert to any problems, although more commonly, other software is used for that.

GitHub and GitLab both support CI. With GitHub, check under "Actions". You can create a YAML file `.github/workflows/maven.yml` to define what to do whenever a push is made. (Show repo "Demo" on GitHub by cloning the repo, edit `Main.java`, add, commit, and push. See what GitHub does!)

GitHub Actions have become very popular (as has GitHub). You can find many tutorials and courses for learning about CI with Actions; there's even certifications available!

To clarify CI, note that software development has many steps no matter what workflow you and/or your term use. You must create and edit your code, test it locally, commit the change to a Git (or other type of) repo. More tests can be run before the code gets pushed to a team repo, where it can undergo code review and approval. If all is well, the software gets built. After the build, integration tests can be run on the built artifacts. If you automate delivery too, even more steps must be done.

Each of these steps can include lengthy commands with many options. Additional steps are often needed such as generating a SBOM, checking dependencies for security issues or newer versions, and license compliance. In the old days, developers created scripts (Linux shell, DOS, or PowerScript) to run the steps for them; after editing your code and saving the files, you just typed the name of your script (often BUILD, GO, DOIT or similar). Such scripts were *quick and dirty*, not robust in the event of errors or changes to your development tools. They often broke.

Later, build tools such as Maven were invented that do much of the work for you with short, simple commands. However, you still need to configure those tools with lengthy XML or other types of files, to run the tools you use in the order you want, with the right options.

A CI system such as GitHub Actions uses a file (in YAML format) to define one or more workflows for the project(s) in that repo. Each workflow defines one or more *jobs* to run in order, and when to run them. The jobs are defined by a list of short scripts (which run a command with the right arguments). However, some actions are so common that GitHub provides those steps as predefined *Actions*. So your job is a series of scripts and

Action names. There are currently (2024) thousands of predefined actions; simple workflows can list one job with just Actions (and no scripts)!

What's so nifty about this is one of the triggers of a workflow can be a simple push. GitHub will create a virtual machine (macOS, Linux, or Windows) and fill it with your build and testing tools. You then use an action to copy your source code into the VM where all the steps get run. If all goes well, the push is done. In either case, you receive a notification of what happened.

So you never need to leave your IDE: code, save, commit & push. Keep working. Later, see what happened to your workflow jobs. And using Actions you don't need to write scripts much either (and when you do, they are very short and simple),

There are other CI systems than GitHub Actions, but they all do the same task: automate all your development steps. If you don' like to use a third-party product like GitHub or GitLabs, you can roll your own using an automation server such as Jenkens.

Continuous Delivery

Another modern trend is ***continuous delivery,*** or "**CD**" is automating the creation of installable software. CD can be triggered by a successful (no errors noted, and properly reviewed and approved) CI cycle. CD can be automated using the same tools as with CI. The idea of CD is to keep the software in the main branch in a production-ready state at all times.

According to Martin Fowler who coined the term Continuous Delivery:

"Continuous Delivery is a software development discipline where you build software in such a way that the software can be released to production at any time.

You're doing continuous delivery when:

a Your software is deployable throughout its lifecycle.

b Your team prioritizes keeping the software deployable over working on new features.

c Anybody can get fast, automated feedback on the production readiness of their systems any time somebody makes a change to them.

d You can perform push-button deployments of any version of the software to any environment on demand."

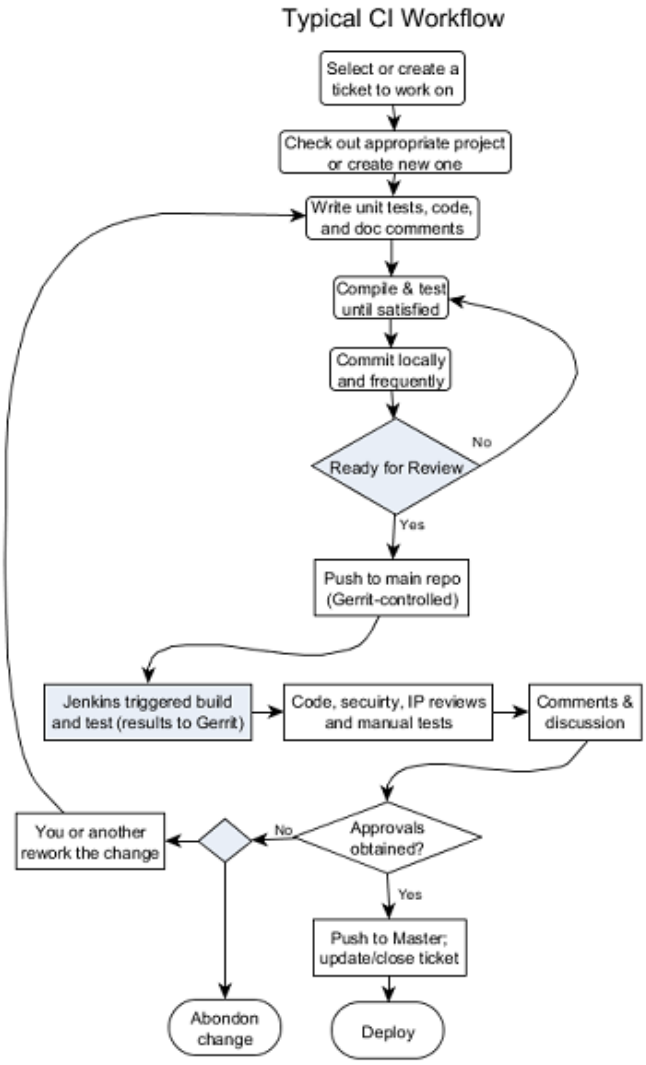
The result of a successful CD cycle is some *artifact* (a package or container) added to some software repository, for example, a Windows Update repository or the

same type of repo for Linux, or perhaps a Nexus repo. The next time an update occurs, the software artifacts in the repo get updated.

Updating enterprise software isn't so simple as the software may be installed on multiple servers and a carefully planned and executed roll-out of the new version is required. This is where CD tools shine.

Deployment can also be automated to either a testing or production environment. (Usually, deployment to production is a manual process however.) When automated and triggered by a successful CD cycle, this is called ***continuous deployment***. (With CD alone, a human must push code to production.)

Integration, delivery, and deployment are the three phases of the *software delivery life cycle*, not to be confused with the *software development life cycle*.



A typical workflow (see flowchart above) starts with an idea for a change, bugfix, new feature, new project, or code refactoring (code cleanup). Let's call the new

work a *feature*. If there isn't one already, a ticket is created for this feature. Once the feature is approved (it will cost money after all) and you will work on it, you take ownership of that ticket. Then you're ready to start.

A commonly use CI and/or CD server is the tool Jenkins-CI. Jenkins (a forked version of "Hudson" which was discontinued in 2020) can be extended with plugins, for example to build, test, and submit Android apps. You can find a decent tutorial on using Jenkins with Java at Vogella.com.

There are other CI/CD tools such as JetBrain's TeamCity, but Jenkins is the most popular after GitHub Actions. There are also hosted tools such as Travis or CircleCI, where you pay to use their servers instead of using your own.

Refer to the workflow diagram above. Step one is to make a new feature branch from the current main branch. (If this work is a new project, you create a new Git repo, for it, configured with the proper access controls for those working on it.) The main branch should always be considered production-ready, working code. You always do all work in feature branches. The CI software used (Gerrit, Teamcity, Jenkins, etc.) should be configured to run a build & test for every branch, for each commit.

The development phase means make the detail design, writing unit and other tests, documenting your code with appropriate doc comments, doing research (answering questions such as *what has been done before for this?* and asking your coworkers for opinions) which should be saved in an email, the ticket itself, some other archive, or a wiki site for the project, and creating code. As you work, you may need to update the ticket that describes the feature.

After completing your changes and reviewing and testing them yourself, it's time to have your work reviewed by others. How you signal that depends on your CI/code review software and its configuration. One common way is to simply create a pull request.

With Gerrit, the process is simple. Gerrit acts as your Git server, hosting all your projects (your organization's private GitHub, if you will). You just attempt to push your work to the official Git repo (in Gerrit) for the project.

The review process is a conversation between developers (and sometimes, administrators who are required to sign-off any changes to some project). If changes are needed, the developer goes back to work, abandoning this pull request, and eventually creating another. Gerrit creates a special branch to hold this feature and all subsequent patches made to satisfy reviewers. Gerrit can be (and often is) configured to send the change request, not only to all project members, but to the Jenkins server as well. Typically, Jenkins will check out the code, run `checkstyle` tools on it, attempt to build the code, run all unit tests, and possibly

other things. The result of all the checks is sent back to Gerrit, which shows the code as *verified,* or *not verified* if any of the automatic checks fail.

Once all reviews are done and the (possibly modified) code is approved, your feature branch is merged automatically or manually into the main branch. (Note if other developers have added changes in the meantime, an automatic merge may not be possible; in a worst case, you will need to merge the updated main branch into your feature branch and create a new pull request for review.)

Once this is done, other software that is used for continuous delivery (CD) will check out the main branch, usually to some virtual machine or container, and then build the software then run all configured tests. These can include the already-run unit tests but usually include integration, automated acceptance. and load/stress tests. If all tests pass, the code is put into an artifact (or binary) repository for clients to use. This is considered as *deployed for testing*, but not for production.

If using *continuous deployment* as well, the artifacts are next pushed onto production servers. This step can be complex as it may include updating load balancers, virtual machine or container orchestration tools (such as vSphere and Kubernetes), monitoring systems, configuration management tools (such as Ansible), and include a phased roll-out rather than a flash-cut over.

The final step is to close the ticket.

> When reviewing code, remember to check for operational features such as logging, configuration hooks, and directions for monitoring. Naturally, review code for quality, regulation, license, and security compliance. A tool such as Gerrit can also notify non-developers of changes, so they can weigh in: marketing or legal departments, for example.

API Design

Instead of designing whole, stand-alone applications, most software today is designed to work with other software. A large (enterprise) application is often written as several modules. Such modules can be coded in parallel by different programmers, but it is vital each programmer understands how to access the services and data from other modules. Such modules are also referred to as *libraries*. To access such a library of code, you need to know its ***Application Programming Interface***, or API. With Java, the standard way to document an API is with doc comments and the `javadoc` tool (which generates HTML pages from those doc comments).

Designing an API means designing the public interface of interfaces (or classes). Bad APIs cause many bugs and lots of programmer frustration, often expensive since generally they are not found until an integration test fails.

Instead of having a module available as code that must be downloaded to use, a modern trend (2020) is to have each module expose its functionality as one or more services. This is known as a *microservices architecture*. Other code uses the module in a client-server fashion, often using HTTP and JSON. Such APIs are sometimes called ***RESTful API***s.

> RESTful APIs cannot be easily documented using Java doc comments. Instead, there are API documentation standards such as OpenAPI. A popular set of tools for such APIs is Swagger. From the Swagger web site:
>
> "The OpenAPI Specification (OAS) defines a standard, language-agnostic interface to RESTful APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection. When properly defined, a consumer can understand and interact with the remote service with a minimal amount of implementation logic.
>
> An OpenAPI definition can then be used by documentation generation tools to display the API, code generation tools to generate servers and clients in various programming languages, testing tools, and many other use cases."

Microservices using RESTful APIs are each smaller and easier to develop. Because each one has an agreed public API, others can work in parallel using stub code (or mocks) for any microservices not yet completed. Finally, microservice designs require less knowledge of the hardware infrastructure. (Many public clouds provide support for such designs and include automatic previsioning of servers, so software developers can ignore that aspect. For example, deploying a bunch of servers for a cluster to provide some enterprise application requires some knowledge of Linux, load balancing, and network security. With microservices deployed as containers (for example, Docker containers), you can leave such details to the cloud.

> An increasingly popular (2018) alternative to RESTful APIs is GraphQL.

An even newer trend (2022) is so-call ***serverless computing***. In this model software developers produces software functions (lambdas in Java) that are uploaded to the cloud (those that provide *functions-as-a-service* or FaaS) and run. The details of how these functions communicate or are provisioned can be ignored.

> Microservices were over-hyped. Today, there is a backlash against using them. It seems that while microservices can solve scalability issues, a new project doesn't yet have any. Yet microservices come with large technical and management costs, and these costs can spiral out of control without proper discipline and management (neither of which is common in start-ups).

Best advice today (2025) is to use a *modular monolith* initially. The modules could be split out into microservices fairly easily later, if needed. (See Google's Towards Modern Development of Cloud Applications for more information.)

Design Patterns

A ***design pattern*** is a standard way to programmatically solve a common type of problem and is generally accepted as a best practice. (Past generations of programmers used other terms for design patterns, such as "paradigms" and "code idioms".)

The purpose of design patterns is to capture software design know-how and make it reusable. Design patterns can improve the structure of software and simplify maintenance. Design patterns also improve communication among software developers and empower less experienced personnel to produce better designs.

Design patterns can be for the detailed design of modules (such as the method chaining pattern) or for the high-level (system architecture) design of the system (such as micro-services). Some apply to both levels of design, but differently (such as model-view-controller).

There are several catalogs of useful design patterns on the Internet (**Show** ootips.org, ...). By becoming familiar with some of these, one can read project proposals and instantly see a pattern that might fit. The book that started it all, *Design Patterns,* was written by *the gang of four (GoF)*: Gamma, Helm, Johnson, and Vlissides, published by Addison-Wesley. Many newer books are available including *Object-Oriented Design & Patterns* by Cay Horstmann (Wiley).

Examples:

Consider classes `Person`, `Employee`, `Customer`, or classes `Person`, `Student`, `Instructor`. It seems logical to have `Employee` and `Customer` extend `Person`, but what if an `Employee` buys something, how can you add them to your `Customer` list? What if an `Instructor` took a course at HCC? These issues are better understood as ***roles***. A `Person` at any given time may be either an Employee or `Customer` or both.

This is a common situation, but tricky to get right in the design. **Qu: How would you do this?** (Ans: Look up the ***adaptive role-playing* pattern** and the *role modeling* pattern, both variants of the classic *decorator* pattern.) Not inheritance at all, but *dependency*: Have `Student`, `Instructor` classes with a `final Person` attribute, and (tediously) add *forwarding* methods.

If you think your design, implementation, and testing would be easier if you could treat `Students` and `Instructors` as `People`, then use a second design pattern too: have a `Personlike` interface that `Person`, `Student` and

`Instructor` fulfills, and `Student` and `Instructor` classes merely forward the method call (i.e., `String getName() {return thePerson.getName(); }`). This is known as the ***delegation*** design pattern (and is related to the ***proxy*** pattern). Note that a single design can be thought of as examples of different design patterns; the various patterns overlap somewhat.

Some additional patterns include the ***composite pattern***, such as used with GUI containers and components; the ***decorator pattern***, such as a `ScrollPane`, which adds to (decorates) a pane; and the ***strategy pattern***, in which there are different ways to do something (such as Window layout).

> Knowing a few design patterns allows you to communicate more effectively with fellow developers. But, don't fall into the trap of forcing your code to slavishly follow a pattern, if it doesn't quite fit.

Earlier we discussed other patterns, such as method chaining pattern and the static factory pattern.

> The static factory pattern can be used in another way, to support a ***service provider interface*** (SPI). SPI enables extensible applications, which one can extend easily without modifying their original code base. For example, Java SE mandates several APIs that they don't provide implementations for, including persistence (database) providers, locale information providers, and others. Using SPI, a given vendor (or end user) can include different third-party implementations of these APIs. Your code generally uses the static factory pattern to get an SPI class (these have names like *Something*`Factory`), which in turn uses the static factory pattern to get an object of the SPI (implementing) class. For example, some XML processing uses SPI:
>
> ```
> SAXParserFactory spf =
>    SAXParserFactory.newInstance();
> SAXParser saxParser = spf.newSAXParser();
> ```

If your design has many objects that must do DNS lookups, or get a random number, or in general, you have client objects that use a common service, you only want a single instance of the service object. This ensures the server can cache results, reuse DB connections, ensure unique random numbers, etc. The pattern is known as the ***singleton*** pattern. You can ensure this by making the constructor `private` (and provide a `readResolve()` method, which is a pseudo-constructor used in *serialization*). You then provide a single instance as a `final` property of the class:

```
class Foo {
   private static final Foo foo = new Foo();
   private Foo() {...}
   public Foo getInstance() { return foo; }
   ...
}
```

The singleton pattern isn't commonly used. The problem with singletons is that they introduce global state into a program, which allows any code to access it at any time (this is called *high* or *tight coupling*). This has proven to be bad, often doubling (or worse) development and testing costs. Jakarta EE doesn't support it well either (you can't always know which JVM will be used, and you can only make a singleton per JVM).

The best way to make a singleton as of Java 5 is to use an `enum` with one element:

```
enum Foo
{  INSTANCE;
   ...
}
```

If the singleton object is expensive to create, it is best to defer creating it until and unless it is used. This is the *lazy-loaded singleton pattern*, also known as initialization-on-demand holder idiom:

```
public class Expensive {
  private Expensive() {}
  private static class LazyHolder {
    static final Expensive INSTANCE =
      new Expensive();
  }
  public static Expensive getInstance() {
    return LazyHolder.INSTANCE;
  }
}
```

The expensive object is created only once, the first time `getInstance` is invoked.

One aspect of design is the common confusion in situations such as classes `Box` and `Rectangle`. It may seem as if a `Box` *is-a* `Rectangle` with a depth (a specialization). But what if I have a collection of `Rectangles`, one is a `Box`, and I try to invoke a `getArea()` method on each? (`Boxes` don't have a clearly defined area the way `Rectangles` do.) By extending `Rectangle` with `Box` we have broken our definition of `Rectangles`.

In practice, objects are defined by their methods. (Eiffel is an exception in that you can specify pre-, post-, and invariant conditions.) But most computer languages only require the name and parameter list be the same to over-ride a method. Unfortunately, that makes it easy to mess up!

Review:

- **preconditions** – The requirements which a method requires its caller to fulfill before invocation of the method.
- **postconditions** – The promises made by a method to its caller.
- **class invariants** – The object's state (values of variables at various points) is always valid.

In Java you can (and should) document pre- and post- conditions in the Java doc comments. Proper design leads to the following important principle:

An extension of a class should *never* modify the behavior (semantics of existing methods) if you treat the extension as though it were one of its superclasses that is, when using over-ridden methods). This rule is called the ***Liskov Substitutability Principle*** or LSP. Also known as *subsumption*. Mathematically you can state this as "Let *p* be any property of objects of type T. Then *p* should be true for objects of type S where S is a subtype of T".

The proper relationships between classes is often a matter of debate. For instance, few people agree on the actual relationship between Square and Rectangle. One design (and ignoring other shapes including Rhombuses, Parallelograms, and Quadrilaterals) is to have Square extend Rectangle, with both classes immutable. A square is-a rectangle with an additional property that the width equals the height; in LSP terms, all the properties of Rectangles apply to Squares too. Something like the following:

```
public class Rectangle {
    private final double w;
    private final double h;
    public Rectangle (double w, double h) { ... }
    public Rectangle scale (double scaleFactor) {
        return new Rectangle( (w * scaleFactor)
                  (h * scaleFactor) );
    }
    public Rectangle setWidth (double w) {
        return new Rectangle( w, this.h );
    }
    public Rectangle setHeight (double h) {
        return new Rectangle( this.w, h );
}   }

public class Square extends Rectangle {
    private final double s;
```

```
        public Square ( double s ) { ... }
        public Rectangle scale (double scaleFactor) {
            return new Square( s * scaleFactor );
        }
        public Rectangle setWidth ( double w ) {
            return new Rectangle( w, this.s ) ;
        }
        public Rectangle setHeight ( double h ) {
            return new Rectangle( this.s, h );
    }   }
```

Another design possibility is to consider there is no IS-A relationship here. Instead have two classes. The `Rectangle` class can have an `isSquare` method, a `squareValue` method to convert a square `Rectangle` into a `Square` (this is a *factory* method), and a constructor taking a `Square`.

> There is a difference between mathematical is-a and software is-a. In math, a square is indeed a type of rectangle. In software it is not, as a rectangle has `setHeight` and `setWidth` methods which allow the sides to be set independently, violating LSP for squares. Using immutable objects side-steps the issue.
>
> In Java, don't use inheritance simply for code reuse; composition is better for that. Don't use inheritance simply to guarantee behavior; interfaces are better for that. Do use inheritance for defining subtypes (is-a).

Another example of LSP is cars. Humans associate "A `Ford` is a `Car`" so they make `Ford` extend `Car`. They also then say "`ModelT` is a `Ford`", so `ModelT` extends `Ford`. The mistake is confusing `Make` (Brand) with `Type` (Class), and further confusing `Model` with `Type`. This kind of mistake can lead to a very ugly class hierarchy when you end up with classes like `FordModelTWithRedPaintAndLeatherSeatsAndPowerWindows`.

A simpler design hierarchy would be to have a `Car` that "has-a" `Make` (`Ford` is a `Make`), "has-a" `Model`, "has-a" collection of `Options` (`SeatMaterial` is an `Option`) and "has-a" `PaintColor`.

As a final example, consider the binary search algorithm shown in COP-2800 (*SearchDemo.java*). The problem lies with this calculation:

```
int mid = (low + high) / 2;
```

This calculation seems correct but contains a hidden assumption: That "`low+high`" can't overflow an `int`. Now that is a reasonable assumption on today's systems, where a process is limited to 2 GiB of addressable memory. So really big arrays can't exist. But that isn't true on all systems, and not on

tomorrow's systems. It won't allow you to run binary search on a 6 billion name phonebook. This is an undocumented precondition.

One possible fix is **to document the precondition**, and let any callers deal with the problems. Another is to make sure that `array.length < Integer.MAX_INT/2`.

Another way is to prevent any overflow. One way is to do calculations using `long`s, not `int`s. Sometimes, a simpler solution is to re-arrange the calculation so that it can't overflow. Here are several ways (I like the first one, for clarity):

```
int mid = low + (high - low) / 2;
int mid = (low & high) + (low ^ high) / 2;
int mid = (low + high) >>> 1;
```

Not all languages have binary operators that are guaranteed to work on `int`s as Java does (the third solution does potentially overflow, but it still works). Not all languages have any problem; Python integer overflows just promote to an integer type that can handle arbitrarily large values correctly. In C-like languages, note you need to know how computer hardware does math to even be aware of the issue.

Dependency Injection (DI)

DI is a design pattern used when coupling (dependencies between classes) is unavoidable in your design, and you want to minimize that. It is one way to do what is known as ***Inversion of Control*** **(IoC).** Simply put, DI means to pass dependencies directly into a class, rather than letting the class work them out for itself. Instead of

```
SomethingNeeded foo = new SomethingNeeded(); // or some get method
```

you pass dependencies into your methods and constructors as arguments, or use some framework to initialize fields for you.

> DI is related to the ***dependency inversion principle*** (the "D" in SOLID), which states that high level modules (those that set business policy for example) should not depend upon low-level modules (that implement some detail). The reverse is also true (hence the word inversion). Both should depend upon abstractions (interfaces) instead. Consider for example shopping carts, payment collection, persistent data storage, sending asynchronous messages, or logging. Although there are many ways of implementing a shopping cart, the way you would use one should be relatively stable in time. If you can extract an interface that represents the concept of a shopping cart as needed by the calling code (who "own" that interface), this interface should be much more stable over time than its implementation. Also, calling code should be much less affected by changes

you could make while maintaining or extending that implementation. You also get to test easily different implementations or choose different ones under different circumstances. That would not be possible under the older way of having the calling (higher-level modules) directly use your shopping cart class.

The interface comes from the design of the higher-level (calling) code, sometimes when you sit to write unit tests for it; the lover-level (called) code that implements the interface can be designed and implemented later.

DI is often done through a container or framework which takes responsibility for resolving these dependencies and injecting them into your code (usually constructors). The end result is looser coupling between components that makes it easier to switch implementations, particularly when it comes to injecting mock (dependent) objects for testing.

The technique is called *inversion of control* because, rather than having your code call methods of some framework, we have some IoC framework invoking the methods of your code.

Such a framework usually works by using reflection to determine the objects needed, builds them, and injects them into your code (say by annotating something as needing this). Other frameworks used configuration files to determine what to inject. (Spring DI uses annotations and reflection.)

It is easy to confuse DI with IoC. DI is one way to follow the IoC concept, but others include the Template pattern and the Service Locator pattern. DI seems to lead to better code in most cases.

Depending on the DI framework used, you could inject objects into:

1 Class fields (should not be private fields in this case and is frowned upon since this type of injection is impossible to test),
2 Setters, or
3 Constructors.

Injection into constructors is currently (2018) a best practice.

Example of DI

Suppose you have a class (say `Car`) that *uses-a* (stores a reference in a field) an object of class B (say `DriverImpl`). If you follow good design principles, class A will use an interface reference for class B's object (in this example, `Driver` is an interface and `DriverImpl` is an implementation of it):

```
public class Car {
  private Driver driver = null;
  public Car () { this.driver = new DriverImpl(); }
  public drive() { if ( driver.isReady() ) ... }
}
```

Notice the coupling is still there. Using an interface didn't help; we still needed to know the name of the actual class. Not only that, but it would be difficult to unit test this class, since you need a working `DriverImpl` class too. And look at the signature of the constructor: you'd never guess the dependency.

A *factory pattern* (discussed previously) moves the responsibilities for knowing which class implements `Driver`, and actually creating the object, to another class (the factory). Here's the same example using a factory pattern:

```
public class Car {
  private Driver driver = null;
  public Car ( DriverFactory df) {
    this.driver = DriverFactory.getDriver(); ... }
  public drive() { if ( driver.isReady() ) ... }
}
```

This is a little better. If the implementation class changes, `Car` does not need any changes, but of course `DriverFactory` will.

With the above, class `Car` was still responsible for creating the dependent class `Driver`. What we really want to do is completely remove the responsibility for creating the dependent objects from our classes that use them. Suppose instead we have this:

```
public class Car {
  private Driver driver = null;
  public Car (Driver driver) {
    this.driver = driver;
     ...
  }
  public void drive() {
    if ( driver.isReady() )
      ...
  }
}
```

This is a much nicer `Car` class! It has no need to worry about implementation of `Driver`, so that can be changed anytime and class `Car` don't care. This version of `Car` makes explicit the dependency in the constructor's signature, and so is less

confusing. It is also easier to unit-test, since none of `Car`'s methods need to create any dependent objects. (You can test `Car` with a mock `Driver`.)

This is the design pattern DI. It is possible to have a version of the DI pattern that uses a `setDriver` method instead of a constructor argument (used by the Spring DI framework), but most people prefer using constructor arguments.

Of course, the `Driver` object must be instantiated someplace so we have only moved the code, not eliminated it. But we have still gained: the class that uses `Driver` (that is, `Car`) no longer has the additional responsibility of locating a suitable implementation class and constructing an object from it; the dependency of `Driver` is now *injected* into `Car`, and `Car` has one less responsibility.

This technique can also be applied to whatever class is using `Car`. Ultimately, `main` or some such method must actually instantiate all the dependent objects. Can't the dependency be removed completely instead of moved to other code?

> Related to the factory pattern is the *service locator* pattern. The difference is there is one uber-factory (the locator) that knows how to instantiate anything your code requires, rather than one factory per item needed. JDNI is a service locator built into Jakarta EE. When class `Car` needs a `Driver` object, it asks the locator for one, something like this:
>
> ```
> Foo f = (Foo) ctx.lookup( "Foo" );
> ```
>
> Where `ctx` is the service locator object. Usually these are configured with some XML config file or with annotations. With XML, if you later change the implementation you need to update the config file. I prefer annotations.

Using IoC (*Inversion of* Control) with a DI framework (sometimes called an *DI container*) makes all this easier; the framework provides all the objects on request (at runtime) based on its configuration. All the programmer needs to do is to tell the framework which type of object is wanted. (Notice how similar this to the Service Locator pattern. Instead of referencing the framework from within `Car` (service locator), some other code references the framework to obtain a `Car` reference, and the framework notes the dependency and handles constructor arguments.)

> In a sense, the DI container or framework takes the place of a main method: it creates objects as needed and invokes their methods as needed. The programmer's job is now to annotate classes that might need injecting and write class methods that use those injected objects. The framework does the rest.

With a DI framework, if some code needed a `Car` object, it could just ask the framework for it. The framework can consult its config files (or uses reflection),

dynamically locates and loads the needed dependent class (`Driver`), create an instance of it, and then invokes `Car`'s constructor with the right arguments, and return a reference to that `Car` object.

For another example, see a beginners guide to Dependency Injection.

DI Containers (Frameworks)

The DI pattern eliminates these compile-time dependencies altogether. Instead, the implementation is found at runtime by code that consults configuration data, just like the factory pattern, or uses reflection. Java SE and EE support this, but Oracle's Java SE doesn't provide an implementation; you'll need to add a DI framework to your CLASSPATH (or possibly your module path for Java 9 and higher).

Some frameworks or containers act like factories, so the client passes an interface (rather than a class) and the framework determines the right class to use (possibly configuring the object along the way). Such frameworks often use Java reflection to locate classes by name. Using interfaces this way is a best practice.

Jakarta EE goes further; its JRE (a *container*) can not only locate and construct objects but can also set fields to values based on annotations. To me, this is dark Jakarta EE voodoo:

```
class Foo {
  ...
  @someAnnotation
  private EntityManager em;  // set magically!
  em.doSomething();
  ...
}
```

Note that injecting references into private fields is frowned upon and Java since 2018 complains when you try (and will not allow it at all at some point).

Some popular frameworks for this are Spring (one part of it anyway), Google's Guice, and the simple picocontainer. (See tutorials.jenkov.com.) Jakarta EE also supports this officially; see jakarta.inject for a good overview of DI.

Qu: *This seems like a lot of work, why bother?* Ans: You shouldn't in simple cases; using an interface and/or factory pattern is good enough. But complex (enterprise) code often has many dependent classes (those that provide services to other classes). Those dependencies may depend on others in complex ways that change over time. You don't want to have to redo code everywhere when a change occurs.

Using a DI framework will automate it all, working out the dependencies and instantiating them in a proper order. **But don't overuse DI!** It is a mistake to try to inject foundational (trivial) class objects, or objects that are very unlikely to ever change.

> A configuration file approach to DI frameworks provides the ability to change the class without recompiling, providing maximum flexibility. A problem with XML-based configuration of DI frameworks is that such files can be hard to read and are not checked by a compiler (a mistake can have the framework return the wrong type of object at runtime). Some frameworks allow the choice of XML configuration, or code configuration (you have code in main that registers what classes to return when asked). You need to decide if you really need that kind of flexibility.
>
> Note, you don't need to use any framework for DI. (You don't need to use DI at all, if not needed.) A Beginner's guide to Dependency Injection shows a good example of DI with Java property file-based configuration and Reflection (and no framework at all).

Here's an example with picocontainer (using code-based configuration):

```
Class Foo {
MutablePicoContainer pico = new DefaultPicoContainer();
...
public static void main ( String[] args ) {
   pico.addComponent( Driver.class );
   pico.addComponent( Car.class );
   ...  // rest of main
}
public void drive () {
  Car car = (Car) pico.getComponent(Car.class);
   ...  // rest of method
}
...  // rest of class
}
```

> If `Driver` takes non-constant constructor arguments (e.g., the name of the driver), you need a fancy framework such as Spring, and the complexity of configuration is often not worth the effort. But such classes (many constructor arguments) are generally inappropriate for the DI pattern.

Model-View-Controller (MVC) — Architectural Pattern

MVC is a design pattern for structuring large user interfaces; swing uses this. This is an architectural design pattern that applies to the user interface layer. The idea is to split the code into three parts:

- The ***model*** is the code that deals with the application's data. It also handles persistence (that is, the database). Rather than scatter SQL statements throughout your code, you create classes with methods to do the updates. These classes are the only ones that need to deal with the database directly. This separation allows you to fine-tune the SQL, change the database from (say) MySQL to PostgreSQL, and not have any of the other code affected. The model has methods to return data (usually requested by the view) and to change data (usually requested by the controller). (Java sometimes uses the term *Document* for a class that plays the role of a model.)

  The model has no user interface. It should have methods that allow the controller to update (change the data in) the model, and for a view to fetch some or all of the data. The model uses *events* to notify *listeners* (the view) that the model changed.

  When you design the model, you design an API for the rest of your view and controller code to use. The details of the data structures used and the persistence details are hidden from the rest of your code. (Internally, the model generally handles the persistence separately from the rest of the model code, sort of a sub-module.)

  Finally, keep in mind that in the model of a GUI component such as a `JTable`, or `JTextArea`, there are often two types of data: state of the component (such as position, isEnabled, isPressed, etc.) and the application data (the contents of a cell in a `JTable`, or the text in a `JTextArea`). Components such as `JButtons` only have the first kind; the model should not store the caption of a button. (In rare cases, these do change and thus should be part of the model.)

- The ***view*** is the user interface. This is typically written using a GUI framework such as AWT, Swing, JavaFX, QT5, or something else, for stand-alone apps. For distributed apps, a web browser is often used with a combination of HTML, CSS, JavaScript, and some programming code (in Java, PHP, or some other language supported by the web server).

  > The best source for web-based technology documentation is developer.mozilla.org. Despite the Mozilla domain, this site is supported by many vendors as the one best source of information.

  The code in the view is typically used to fetch needed data from the model so it can be displayed, validate user input (before passing it to a controller), and update the display when necessary (for example, a session time-out, or the model has been updated). By keeping this code separate you can easily support multiple user interfaces and not have to make changes to the rest of the code: a web browser-based UI, an executable program run on the client, or as a smartphone app.

A typical application will have several screens or views. You can of course make a separate class for each if desired, but the whole collection of user interface code is called the view.

Modern view code often uses HTML templates with placeholders for the model data (and more). To display a view, the model code, processes the template, inserting the mode data as needed, then sends the resulting HTML (and CSS) to a web browser).

- The ***controller*** is everything else, the "business logic" of the application. When the user clicks on something, the view tells the controller. (In a typical web application, the controller receives a GET or POST HTTP message.) The controller in turn will decide what to do next. It may send a message to the view code to show a different web page (or otherwise update the view) or send a message to the model (with the data supplied by the user in a form). The controller may also send an email to someone, send a message to a business partner (for example, order another meal when someone makes a reservation), generate log messages, or some combination of all of these. (Note that if the model gets updated, either the model or the controller needs to tell the view in case it needs updating too.)

  Like with views, applications may have several controllers, one for each possible action. These can be different methods of one class or one controller method per class. Either way, the whole collection of controllers is "the controller". Note if you have multiple controller methods, some sort of *dispatcher* method is used by the view to invoke the right one.

  MVC was first invented for Smalltalk language. In this original definition, it consists of three kinds of objects: The model is the application object, the view is its screen presentation, and the controller defines the way the user interface reacts to user input. So in that definition, most of the application code is part of the model, not the controller.

  In some modern uses, the controller handles most of the application logic. It is best to have simple controller method(s), and handle all application tasks as *service* methods invoked by the controller.

  The different modules in MVC generally communicate via events. When the model changes, it "fires" an event. When the user has interacted with the view, the view fires an event. The other parts of your code either ignore these notifications or take some action in response. (MVC need not use events at all, but instead can communicate solely via method calls. For example, if the user

clicks a submit butten in the view, the view code could just invoke a method in the controller in response.)

For example, consider an eBay auction: The current view shows the item, highest bid, and time left. If another user makes a new bid, the model will be updated with that information, and an event fired. The view can refresh the "current bid" info in response to that event.

(From A Swing Architecture Overview):

Models must be able to notify any interested parties (such as views) when their data or value changes. Swing models use the *JavaBeans Event model* for the implementation of this notification. There are two approaches for this notification used in Swing, and in general for MVC:

Send a ***lightweight notification*** that the state (model) has "changed" and require the listener to respond by sending a query back to the model to find out what has changed. The advantage of this approach is that a single event instance can be used for all notifications from a particular model — which is highly desirable when the notifications tend to be high in frequency (such as when a `JScrollBar` is dragged).

Send a ***stateful notification*** that describes more precisely how the model has changed. This alternative requires a different event instance for each notification. It is desirable when a generic notification doesn't provide the listener with enough information to determine efficiently what has changed by querying the model (such as when a column of cells change their values in a `JTable`).

Lightweight notifications use `ChangeEvents` and a `ChangeListener`. `JButtons` uses these, for example. Some stateful notifications include `TableModelEvent` (used with `JTables`) and `DocumentEvents` (used with `JTextAreas`). Naturally, you can invent your own events.

There are many variations on MVC, to suit a particular situation. For example, you might combine the controller and view, or split the model into several classes (so the view is responsible for session data such as a shopping cart but inventory is handled differently). Swing combines the controller and view into a single component called the *UI delegate*. This is because there is a high degree of coupling between the two so it makes sense to put that functionality in one place.

Over time, some of these variants have become sufficiently popular to have their own acronyms, such as MVP and MVVM. See this short blog post at vtsen.hashnode that explains the differences nicely.

Another possibility is to split the controller into the application flow control (just respond to user events from the view and decide what view to show next, the original meaning of controller). The possibly complex logic of your app can be put into a separate module, in the model, or even the view (rare).

MVC is flexible about the responsibilities of each module. For example, you can have a simple model API and have all the work done in the controller. Or have a high-level API and have all the work done in the model. You can pick whatever design seems simpler. As a rule of thumb, try to keep the complexity of each module similar. It would be a shame if the programmer responsible for the controller is done in a week and the one responsible for the model takes months.

MVC provides several benefits: the project is usually easier to develop, especially in a team (a UI expert can develop the view and a DB expert the model), and the application will *scale* up (support many users at once), possibly by using a cluster of web servers for the view. Even small applications can usefully apply MVC. (You can think of a web page as using HTML for the model, CSS for the view, and JavaScript for the controller.)

MVC does add some complexity though; you will have many more source code files than if you did it all in one. You may need to use fancy database features (e.g., stored procedures) for performance (and should use them for security in any case).

Although it is sometimes applied to the overall enterprise application architecture, MVC is really not the same thing as a multi-tiered application. But don't worry about the differences; there is a lot of overlapping definitions floating around the Internet in any case.

State Pattern

This design pattern is similar to the strategy pattern. It is used when some methods (the behavior) need to change depending on the state of the object. A naive approach is to create an enum (or a string) field to keep track of the current state. Then in state-dependent methods, use a `switch` statement to do the right thing depending on the state. This is a problem since if new states are created or removed, all methods need to have their `switch` statements updated with new branches.

An object `obj` using the state pattern delegates any state-dependent methods to a state object that `obj` keeps a reference to. When a method changes to a different state, it just needs to update the state reference in `obj`. No switch statements needed! (*Show AlertDemo.java*.)

Modern Enterprise Application Architecture

Just as there are design patterns for code, there are software architectural design patterns for designing the overall structure of the application. There are five to ten commonly used patterns (it depends on who you ask), and not all applications are complex enough to need the overhead of any of these designs. (*Show the resources for this.*) Examples of such patterns include layered, client-server, micro-services, and others.

Classic enterprise application architecture is to split the code into three different layers, or *tiers*: the *presentation layer* (MVC applies here), the data layer (persistence), and everything else in the middle layer (the business logic). When the user interacts with the user interface, a controller takes in the data from the view (form data) and will invoke *services* in the business logic layer. The results of those method calls will be used by the controller to decide which view to show next, and to populate the view's model with the data it needs to show. The controller and services share data by exchanging DTOs, or *data transfer objects*. These DTOs are dumb containers for data, but using them reduces coupling between the two layers. Also note the services are methods of objects and are known to the controllers via interfaces. (There is a similar decoupling between the business logic (service) classes and the persistence layer classes.) This three tier structure allows you to change any layer's code without affecting the other two layers, for example changing the user interface or changing from files to a database.

Enterprise applications need to scale up. They run in datacenters containing many racks of physical servers and storage units, all networked together somehow. Until a few years ago, the dominant design pattern was to have many applications run in the one datacenter, with networking, storage, and virtual machines (VMs) allocated as needed to each application. (The physical servers ran *hypervisor* software, which in turn ran the VMs.) This design is still used.

A software engineer determines the requirements for some enterprise application: how much storage is needed and of what type, network requirements, and (if a cluster is needed) how many VMs to allocate.

It's more complex than that; you also need to worry about a *load balancer* for each application's cluster, ways to add or remove storage or VMs to it, controlling access (networking) to the VMs in the cluster, monitoring all of that, and management of the app including deployment of upgrades (and downgrades when you mess something up).

The infrastructure must also be monitored and managed. If the load on a physical server is too high, somehow you need to know that and be able to move applications (VMs) from one server to another without disruption (or at least, not too much disruption). All of that requires indirect networking at the very least, and

with VMs migrating from place to place, a way of finding them is needed (*service discovery*, often some sort of registry is used). Finally, **if different requests of the same session to your application can go to different VMs, or if VMs can move to different physical servers, you cannot use memory to hold any state.**

If an organization hosts only their own applications in their datacenter, they need software to manage and run the whole thing. Such software includes vSphere from VMware and the open source OpenStack.

Rather than run their own datacenters, today companies have built very large datacenters that host customer's applications, for a fee. This sort of ***multi-tenant*** datacenter (or rather a group of them in different geolocations) are known as public clouds.

You must design your applications for such environments: no state in memory, ability to run load-balanced clusters that can grow, shrink, and move, and so on. For example, shopping carts cannot be stored in memory but use fast but persistent and shared memory technology, such as Redis or Memcached, or the modern fork of Redis, Valkey.

Cloud vendors provide many storage offerings, but if (for example) you build your application around Amazon's S3 storage, you cannot easily move your application to a different vendor's cloud. Many applications opt for simple storage, monitoring, and management from the cloud vendor, and run their own database and other software in the VMs. That keeps things portable.

> An alternate architectural pattern was invented by Microsoft. Known as *virtual actor* pattern, it treats the whole system in a more familiar, OOP way. Microsoft provides a popular .NET framework for this, Orleans.

The section started out with "until a few years ago". What has changed? The answer is *containers*, especially Docker.

Containers

Virtual Machines (VMs) provide good isolation for multiple applications running on the same server hardware. But VMs are not perfect. They take a long time to boot up, must be all kept up to date with patches, and the redundant software takes lots of unnecessary storage and memory (and hence, money). Developers need to know details such as setting up security and networking for the VM's operating system.

Datacenters today are shifting away from VMs. Instead, they run ***containers*** on the physical servers. Containers are self-contained execution environments (not the OS though), each with its own private resources such as CPU, memory, block I/O, and NICs (network connections). Applications designed from the start to run as one or more containers are often called *cloud native*.

Compared to a VM, a container is very lightweight. The physical servers generally run the Linux OS instead of some hypervisor. (On a Windows OS, a Linux VM may be needed to run the containers.) An OS has the ability to isolate a set of processes from other such sets. Each set of processes, together with a private filesystem, private users, separate networking, and resource limits, is a *container*. Because they are just some processes, containers are fast to start. There is no redundant storage wasted for the OS. You do not run a separate OS for each.

While many operating systems supported containers in the past, creating, deploying, and managing them was a pain. Docker changed that by defining a Docker (or container) *image file* (usually just called an *image*), which is composed of several immutable sets of files, overlaid on top of each other, with only the topmost layer (at most) writable by the running container This ephemeral layer is called the *container layer*. A user can easily define a bunch of layers they want to comprise their container and build it easily. Such image files contain setup and configuration information as well.

Typically, a container daemon (or *container runtime*) is told to create containers from these image files. This clever design means that each container can be self-contained with all the DLLs, services, and files it needs. (Of course, this means each container image file has a redundant copy of the DLLs used.)

That topmost writable layer, if present, is ephemeral (and is often called the *container layer*). Any additions or changes to files done by the processes inside a running container confined to that layer, and are lost once the container stops. For persistent storage, you need to attach (mount) a *storage volume* (commonly, just "volume") to write to, or use a remote storage service (the sort provided by databases and/or clouds). You need to keep this in mind as you develop software for containers.

> You can build a Docker container image a step at a time while running that container: start your container, then add/change whatever files you wish. When you stop the container, the state is not reset immediately; the changed container layer is still present. You can find the just-updated container and do a *commit* operation to update the corresponding container image file. This is done by copying the ephemeral container layer as a new read-only layer, and modifying the image file to include that new layer on top. The next time you create a container from that image, a new read-write container layer is used. You can repeat this many times to build a container image file a bit at a time. In the end you may have lots of layers doing this!

In addition to having tools to create, delete, update, start, and stop (and generally manage) Docker image files, Docker maintains a repository of popular prebuilt images that are ready to use. Some of these are very bare-bones images, meant to be the base building block of a more complex image. (Your organization can also

use private repositories of images, known as registries.) Docker provides simple tools to start containers from the downloaded or built images, and to manage all running containers. All the lowest level, the virtualization work is handled by the Docker daemon (the container runtime) so programmers need not worry about it. (Docker Desktop is a nice GUI for working with Docker.)

Delivering an application using a container is called a *containerized application*. Such applications are self-contained and can work on nearly any system without porting! Java works very well with containers. A Java containerized application does not need to have a JRE installed on the host as it is bundled with the container.

> Updates are an issue with containers. Normally, you update your OS and DLLs with a package manager, and all apps and services pick up the changes the next time they start. (After testing, of course.) But doing that has no affect on containers! You will need to download updated layers, then update your container image file to use the new layers. In this sense, containers are like statically linked executables; you must recompile those to pick up any updates to libraries.
>
> The best way to handle updates is to have the base image layer contain most or all of the DLLs required. Then when there are security or other updates to those, you pull just an updated base layer image from the Docker (or other) hub. Then you rebuild your images and retest them. Finally, you deploy the new containers. They same is true of other intermediate layers with updates.

Besides portable applications (they only need a container runtime), containers are also useful for running untrusted code and for running tests. (Those web sites that allow you to upload code and run it generally use containers to do so.)

Sometimes a service is composed of several containers working together and which depend on either other. A ***container orchestration*** tool is used to manage these sets of containers (called a ***pod***) and to monitor and manage them. A complex but common tool for this today is called ***Kubernetes*** (often spelled "k8s"). (For example, an application service using several web servers with a load balancer, application (Java) servers (also with a load balancer), a database server(s), a log server, and maybe some storage volumes, might all form a single pod.) An orchestration tool will take care of setting up all the networks, starting all the containers, adding more as demand grows, replacing crashed containers, hooking up networking between the containers, and more. This tool uses a configuration file to define the pod and the setup required.

Like datacenters of VMs, datacenters of containers have the same management and monitoring needs. Only they use different software for this, often called *container orchestration engines* (*COEs*) such as Docker Swarm, Kubernetes, and Mesos.

Don't let the complexity put you off. You don't need orchestration to begin with. You can get started easily by installing Docker (Docker Desktop on Windows) and using a pre-built container image for plain JDK, or JDK with Maven (or Gradle), or Jakarta EE (renamed to just Jakarta) with or without Spring), etc. Create and test a Java app or service in the they way you've always done, commit the change to the container image, and you're done! (In fact, this is exactly the workflow used for CI.) Anyone with a compatible container runtime and easily run your containerized application if you share your image. You can also try a pre-built container for Java from Quarkus.io or others.

Of course, Docker isn't the only container system out there, there are lots. But Docker is the most popular and has contributed most of their stuff to the Open Container Initiative, which is striving to make such containers portable and not dependent on Linux.

Linux is usually required one way or another. You can use Docker on Windows by running a Linux VM that in turn runs all the containers. (In Windows 11, a Linux VM is no longer needed; I believe Docker runs on WSL2 in Windows 11.)

> Docker is an excellent product, but it does have two issues. It requires a daemon to be running all the time, with root (administrator) privileges. And the processes launched by that daemon (all your containers) are also running as root. This can have security implications.
>
> There is a modern replacement known as Podman (and Podman-Desktop). Podman (for Pod MANager) uses the Linux `init` daemon to launch containers, so no addition daemon is needed and the resulting containers run as the regular user who started them. On Windows, Podman currently requires a Linux VM.

Containers for the build environment — It is not unusual today to have a container with all your build tools and development environment, including all testing and code review tools. To work on software, you would launch your IDE in this container. The build output is tested and then used to build the containerized app (a separate container). This method can ensure all developers use the exact same development environment, and that this environment can be recreated exactly, years later if needed, to update old code. It also allows you to carry your development environment container on a flash drive so you can use it anywhere. (Of course, not all teams require everyone to use the same IDE.)

This method of development is known as a ***multi-stage build***: you create an intermediate container with all the required tools to compile or produce your final artifacts (i.e., the final executable). Then, you copy only the resulting artifacts to the final container image, without the additional development dependencies, temporary build files, etc. (From itnext.io/software-development-in-containers-a-

cookbook, accessed 12/2021.) You can also post-process the artifacts to strip any non-determinism from them. (Although after testing, you may want to add some back in for security; then more testing is needed.) A well-crafted multistage build can reduce the attack surface, decreasing vulnerabilities of the final container. It is thus safer and reduces image size.

> As of 2021, Windows containers cannot natively run GUI applications. However, Linux containers easily do so. Thus on Windows, you may need a Linux VM to run your development environment in a container. (You may not need that with Windows 11 or newer.) An alternative is to use a virtual machine (not a container) for your development environment. Vagrant is often used to manage VMs, although recently (2025) the license changed for vagrant so you might wish to find an alternative.

CI/CD setups make such work-flows nearly painless: you fire up your containerized IDE, work on code, and do a push to the shared repo when ready. Every other step (except for human code review) is or can be automated. Learning how to do that does mean learning yet another skill, but it is worth doing. (Rather than learn GitHub Actions or some other language for this, many developers prefer writing Linux shell scripts to build and test in their CI/CD pipeline.)

> There are some additional steps, such as building the application container, testing that (as opposed to building and testing the app directly which you still do), cleaning and minimizing the app's container image, digitally signing it, and adding it to a container registry (a repo of container images).

Microservices Architectural Pattern

Because containers are much faster and less resource intensive, they have sparked a new (not really; see *service-oriented architecture*) design known as ***micro-services***. Instead of constructing your application as a single piece of code, you divide it into many very small services that communicate using some API (commonly, HTTPS RESTful API). It is not required, but it is common for each micro service to run in a container. Such a design does take more network bandwidth, but allows one to have a cluster for any bottleneck part of the app. You can think of this as creating a multi-threaded application, but one where each thread runs in a container and communicates using some API. A "modular" design can be much easier for a team to build. Such distributed applications can have less than a dozen micro-services, or more than 150. (When the number gets large, managing the dependencies between can become challenging.)

Such a design comes with various challenges, including trying to debug the application as a whole, monitoring and controlling the application, and other tasks require new tools and techniques.

In 2024, microservices for cloud applications became so popular that the term API generally now refers to the HTTPS networking set of URLs your application uses. *Tracing* an application similarly now means tracing the flow from a user interaction through all the various microservices.

Some people still use these terms in their original sense, so be careful of context when you talk about this stuff.

Microservices today (2025) have fallen out of favor with some developers. The design is difficult to get right, sharing state becomes nearly impossible, trouble-shooting is much harder, the APIs can grow into something like spaghetti code making the service/application hard to maintain and grow. Security is also an increased concern. Testing your code requires multiple services that communicate over a network, rather than a single executable.

In summary, today enterprise applications must be able to run within a modern datacenter, and as a software developer you must be aware of the operational issues of clusters of VMs and/or containers, typically running on the Linux OS. A common technique for cloud-based applications is to split a complex application into many tiny services, each easy to create and test. This is known as microservices. Such a design includes many developer challenges not required of traditional "monolithic" applications. Because there are so many possible platforms and public and private clouds, you should know these general ideas and then research the specific infrastructure and APIs to be used by your project.

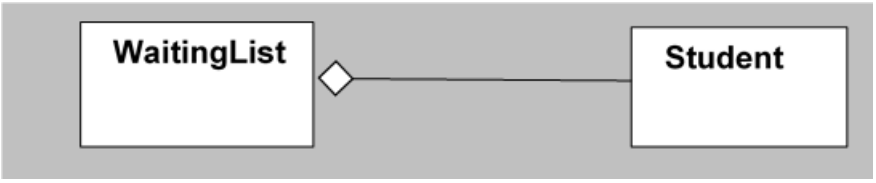
If you want to learn on your own, I suggest running through the Docker tutorial to create and run a container or two (*Hello world* and your preferred development environment would be two good choices), then learn the simplest COE for Docker, Docker Swarm. Next, learn Kubernetes. After that, find some Amazon Web Services (AWS) tutorials for the sixty-odd services they offer for cloud native applications and services. Along the way, learn about networking and security. Learning some Linux command line stuff wouldn't hurt you either!

Basic UML — Documenting Requirements and Designs

Today it is common to design with pictures and diagrams and less pseudocode. Traditional flowcharts (see for example this flowchart) don't work well with OOP so other types of diagrams are used. For example, each class could be a box with the name of the class at the top and a list of the class members inside. The boxes might be connected with lines to show associations (relationships or collaborations) between the classes (really objects). Many types of diagrams are possible to show various things, such as class diagrams, object diagrams, sequence diagrams, state diagrams, etc. (You rarely use all types for a particular project.)

Originally every OO book author invented their own graphical notation. Since they all do the same job, it was sensible to merge them all into a single notation. Surprisingly, this is exactly what happened. The standard is called the ***Unified Modeling Language*** or UML.

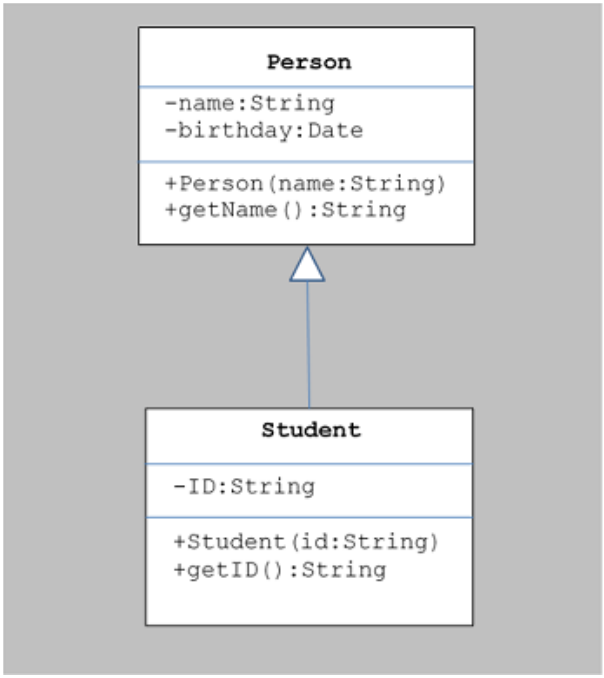
Many books and on-line resources describe UML, and software (both free and commercial) that allows you to create UML diagrams (e.g., **violet**, ArgoUML, nClass, StarUML (which I personally like so much I bought the commercial version), and IDE add-ons). (***Show UML PDF guide***.) A simplified class diagram:



The (hollow) diamond line on a solid line shows that a `WaitingList` object contains (or *aggregates*) one or more `Student` objects. A solid diamond show *composition*, which for Java is about the same thing. (UML is not just used for Java.)

Here's a list of basic UML lines for use in a class diagram:

- Solid line with hollow triangle arrowhead shows inheritance (is-a).
- Dashed line with hollow triangle arrowhead shows interface implementation. (Show interfaces by using *guillemets*, i.e., French quotes around the name <<name>>. Show abstract classes using italics.)
- Dashed line with ">" (open arrowhead) shows dependency (uses-a).
- Solid line with hollow diamond shows aggregation (has-a).
- Solid line with no arrows shows association, with a ">" open arrowhead it is a directed association.
- Dashed line to a dashed box shows a comment.



Each box represents a class (or interface), with the class name at the top, then a list of (public) fields in the middle (if any), and the (public) methods at the bottom:

A leading plus shows a public member, minus shows a private one. Notice how the inherited members aren't shown, unless they are overridden.

> Any part of a class except its name can be omitted from a UML diagram. The idea of such a diagram is to show something specific; needless detail should be omitted whenever possible.

The lines many have the multiplicity shown if desired, at each end; if omitted it is assumed "1". This multiplicity refers to how many objects of one class are involved in the relationship:

* (any number), 0..* (same), 1..* (one or more), 0..1 (either zero or one), *n* (exactly *n*).

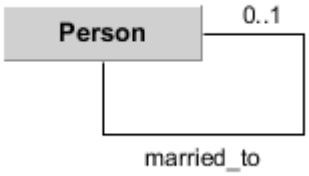
Finally, the association lines can be labeled (at either or both ends), to show the association details:

[student]----registers for----------[course]

Qu: what multiplicity numbers would you put at each end? Ans: "1" at the student end for sure. The other end depends on your design: maybe "1", maybe "1..*" if your design calls for a method that allows a student to register for several courses at once.

Don't show foundation classes (i.e., basic classes such as String, Date, Point, Line, etc.) in your UML diagrams.

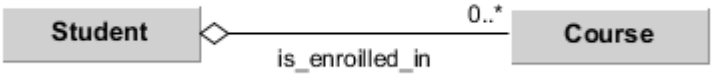
Example: "Person is married to" Association



This is a general association, between one Person object and another (or none). This sort of self-association is called *reflexive* or sometimes *recursive*. Here's the code for this:

```
public class Person {
  private Person marriedTo;
  public Person getMarriedTo() { return marriedTo; }
  public void setMarriedTo( Person spouse ) {
    this.marriedTo = spouse;
  }
}
```

Example: "Student enrolls in Courses" Association



Here, the association is that a student "has-a" course. Notice where the hollow diamond is located; we could have used the association "a course has students" instead. Here's the code for class Student (many parts of a production quality class Student have been omitted):

```
public class Student {
  private List<Course> courses = new ArrayList<>();
  public List<Course> getCourses() {return courses;}
  public void enrollIn( Course c ) {
    courses.add( c );  // checking, etc. omitted
  }
  public void withdraw( Course c ) {
    courses.remove( c );  // checking, etc. omitted
  }
}
```

Try modeling and then implementing the association *a person may have many friends*.

There is software (such as IDE add-ons) that can generate UML automatically from code and even update the code when you edit the UML!

The Five Fundamental Principles of Object-Oriented Design

The acronym SOLID (see the SOLID Wikipedia article) is used to describe five basic principles of OOD:

- **Single Responsibility Principle (SRP)**: Any class should have a single responsibility. This applies to methods as well. Any class or method with multiple responsibilities will likely have multiple reasons to change in the future, and also be much harder to test. (It is better to have many tiny classes and methods, than fewer larger ones! (Q: why? A: small, focused methods and classes are much easier to understand, test, debug, and refactor.)
- **Open/Closed Principle (OCP)**: Classes should be open to being extended, but closed to being modified. That means you shouldn't need to modify code to extend it; proper use of well-defined interfaces and inheritance can accomplish this without modifying existing classes.
- **Liskov Substitution Principle (LSP)**: Objects of one type should be replaceable with instances of subtypes, without altering the correctness of that program.
- **Interface Segregation Principle (ISP)**: Many client-specific interfaces are better than one general-purpose interface, even for a single class.
- **Dependency Inversion Principle (DIP)**: High-level modules should not depend on low-level modules, or vice-versa; both should depend on abstractions. (Also, abstractions should not depend upon details; details should depend upon abstractions.) Even when you have a single object of a single class, it is usually a good idea to make an interface. (Q: why? A: you can later switch implementations of the interface for specific situations, and you can use "mock" objects easily for testing.)

One other principle is worth memorizing: **The Principle of Least Surprise (**or **astonishment):** When facing a design choice, do the least surprising thing. This means all your classes should work in similar ways.

Lecture 16 — Multi-threading (Concurrency)

> "Some people, when confronted with a problem, think, "I know, I'll use threads," and then two they hav erpoblesms." — Ned Batchelder

Multi-tasking

In the beginning, computers ran in ***batch mode***: a single program with total access to all resources. Later, a multi-tasking operating system enabled multiple (independent) jobs to run (more or less) simultaneously, even though there was only a single CPU. The jobs take turns running some of their code at one time. This type of interleaving of jobs is called ***concurrency***.

Multitasking is performing two or more tasks (or *jobs*) at roughly the same time. Nearly all operating systems are capable of multitasking by using one of two multitasking techniques: process-based multitasking and thread-based multitasking.

Each running job was (and sometimes still is) called a ***process***, which has *state* (a collection of data and handles to system resources such as files and devices) and a single sequence of instructions. It turns out that many tasks are easier to write, debug, and manage if organized as multiple copies of a process (e.g., a web server). Processes are sometimes called tasks, threads, or jobs. (Although there are some distinctions between these concepts, we will ignore them for now.)

Time slice, (context) switching, and *scheduling* are operating system ("OS") terms related to multi-tasking. The amount of time a task is allowed to use the CPU exclusively is called a ***time slice***, typically 10ms (a hundredth of a second) but can vary depending on your OS and other factors from 1ms to more than 120ms.

When the OS notices the time slice for some task has expired, it can ***preempt*** the task. Then it must decide which task to run next (***scheduling***), and restore the selected task so it can run from where it last left off (***context switching***). To switch between tasks, the OS must copy all *register* values to memory, and restore the values from the task it is about to run.

Newer computers have multiple CPUs (called *SMP* or Symmetric Multi-Processing, or ***multi-core***). As many processes can run simultaneously as a computer has CPUs/cores.

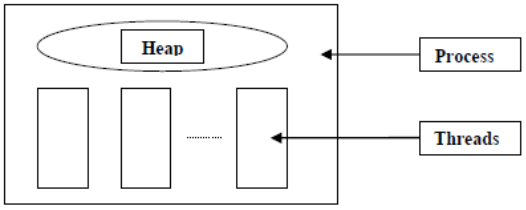
Running multiple tasks at the same time on different hardware is a type of concurrency known as *parallelism*.

Each core usually has its own RAM *cache*, so updates to shared variables may or may not be visible in other threads (and changes may be reordered), unless you take explicit steps to handle that.

The problems with multi-processing are that **creating a new process is slow and sharing data and resources between them is difficult.** It is possible to set up a region of shared memory for two or more processes, but details differ between OSes and shared memory is rarely used. Sharing files (or anything) is also fraught with peril. A DBMS can be used to share data safely, but slowly. Processes can also use some form of message passing, e.g., pipes. All these techniques are known as *interprocess communication*, or IPC.

To address the need for fast and easy ways to start multiple tasks that can share access to data, processes are now allowed to contain multiple (independent) sequences of instructions, known as *threads* (or sometimes *tasks*).

> Process-based multitasking is running two or more programs ("processes") concurrently. Thread-based multitasking is having a single program perform two tasks concurrently. For example, a word processing program can check the spelling of words in a document while you still can write the document.



> A ***thread*** is a sequence of steps executed one at a time. A multi-threaded program has several (up to thousands) of such threads all running at the same time (concurrently).
>
> Most server programs today are multi-threaded: web, FTP, DB, etc. Usually each incoming request is handled by a separate thread. For the client-side, threads are used to provide a responsive user interface and for timers. (Java includes timer classes that do this automatically for you.) Often a thread is considered a mini or *light-weight* process.

The process is still used as a container for the state (the *heap* contents, and also OS resources such as priority, user ID, current directory, open file handles, network sockets, etc.) that all the threads in the process share. And every process contains at least one thread. However, **each thread contains its own stack**, used to hold the method-local variables. In some OSes, threads can also contain private (per-thread) "global" objects and possibly other state (and sometimes an extra stack for the kernel's use).

> Threads of one process all share the same address space, in particular the same heap (where objects are). This makes switching from one thread to another much faster than switching from one process to another, and allows an easy and fast way for the different threads to share information (they can

all access the same objects). This sharing is also the source of all problems with multi-threading.

Summary: *The threads of a single process share memory: objects (on the heap) but not variables local to methods (on the stack). This is because each thread has its own stack. Sharing data is useful, but can lead to problems. The different threads run concurrently (interleaved) no matter if you have one CPU core or many.*

Qu: have you ever written a multi-threaded program? Ans: Yes, all Java programs are multithreaded. The programs we've written so far run in a thread called **`main`**, the *garbage collector* runs in a different thread, and the AWT event handling (the *AWT Event Handling Thread*, or *Event Dispatch Thread*) runs in another thread. (Qu: why doesn't a GUI program end when `main` returns? Ans: A Java program (the JVM process) ends only when all (user) threads have terminated.)

Using threads can simplify program design especially for servers (create one thread per client request), but also for applications (responsive GUIs, fetching data from the Internet while other stuff goes on, using timers, etc., are all easier using threads). Without threads, the programmer must manage all the tasks in a single thread, which is much harder. (It can be more efficient in some cases.)

(*Show Threads.java demo*.**)**

In addition to simplifying code, **multi-threaded programs usually run faster**. This is because when one task is blocked and can't continue yet (e.g., waiting for input from user or from a disk or network), other tasks can continue to run, rather than force the whole program to wait.

As mentioned above, **modern CPUs have multiple cores**. Only a multi-threaded program can take advantage of additional cores. (Of course, other processes could run on other cores at the same time as your process).

It seems obvious that a multi-threaded program will run faster if you have more cores than if you have fewer cores. However, there is a limit to this speedup. Known as **Amdahl's law**, the max speedup is limited by the slowest sequential sequence (slowest thread). Some tasks will even run slower if designed to run as multiple threads! (For example, calculating Fibonacci numbers is fastest using a single thread.) See Quora.com for a good explanation.

(Gustafson argued that Amdahl was too pessimistic for massively parallel machines and created a modified, more optimistic law.)

While multi-threading makes most programming tasks easier and faster, there are some pitfalls you need to watch for. And since all Java programs are multi-threaded (except Java ME), you can't ignore this as some rare issue. AWT,

Swing, and JavaFX are multi-threaded even if you don't create additional threads manually, so all GUI applications are multi-threaded. Most Java enterprise frameworks are multi-threaded too.

The pitfalls all arise from sharing *mutable state* between threads. Threads communicate primarily by sharing access to objects' fields. This is efficient, but makes two kinds of errors possible:

Thread interference errors occur when different threads access shared data using a sequence of steps. As the OS preempts the different threads, these steps may run in an interleaved fashion. This can cause data loss and/or corruption. **If the threads don't ever share data, then each one is independent of the others and no interference is possible. There are also no problems possible if the shared data is immutable.**

> Independent threads can be run ***asynchronously*** since any interleaving of the sequences won't matter. This is the default assumption in Java (and indeed, in most programming and even in hardware design). Sometimes however, one thread must wait until another thread has completed some action. In this case, you must run the threads ***synchronously***. Java includes support for doing this.

The other type of errors possible are called ***memory consistency*** errors, and are related to modern hardware. **Each CPU/core today contains a RAM (main memory) cache plus *registers* that hold data**. When one thread changes shared data, that change may only be saved in a register or in a local cache and may not be "flushed" to RAM right away. So if another thread tries to read that same data from RAM, it will fetch the (old) value and not see the change. If the second process also updates the value, both threads may use the wrong values. This problem is called a ***visibility*** issue, specifically that of ***stale data***. Corruption of shared data is possible too.

> **This explanation is very simplistic.** Caching and memory systems on modern hardware are incredibly complicated. There are multiple caches on any system, such as L0 (Intel uses this), L1d and L1i (for data and instructions) caches that are per-core, L2 and sometimes L3 (and rarely, L4) caches that are shared between cores. Keeping shared information consistent in all these caches requires they talk to each other using a *cache coherence* protocol (e.g., MESI). This architecture may require programmers to be aware of caching and include cache flushing and memory barriers (or fences) to be included in your code, when writing close-to-the-hardware or highly optimized code.

Further, modern CPUs can reorder instructions from the order provided in the code, even with single threaded programs. And so can the compiler. (And even this description is still somewhat simplistic!)

Let's not even consider code running on GPUs or servers using NUMA. There are security issues as well (e.g., false sharing of data in cache lines between processes.)

Fortunately, the simple explanation provided above suffices for most Java (and other) software, for understanding what you (the programmer) must do to make your multi-threaded programs work correctly.

To better understand memory systems, I suggest reading the following:

What every programmer should know about memory, Part 1,

What every programmer should know about memory part 2 (caching)

Written in 2007 by Ulrich Drepper, maybe 95% still applies today. These are classic references, but the articles are long and a bit dry to read. More modern articles are:

Understanding the Processor

Chapter 7 of the book *Rust Atomics and Locks: Low-Level Concurrency in Practice,* by Mara Bos, and published in 2023 by O'Reilly. (The book is about Rust programming language but his chapter is about the hardware and not so much about any particular programming language.

Modern Microprocessors

A relatively short and easily read article by Jason Robert Carey Patterson, written in 2001 but updated in 2016, it provides an excellent overview of modern hardware (including memory).

Code that is written to operate safely in a multi-threaded environment is called ***thread safe***. **In most cases, it is perfectly safe to ignore threading issues**, such as a non-GUI application with no state (that is, only local variables).

Consider:

```
List<String> list = new ArrayList<>();
```

If this statement is inside a method, then `list` is only accessible by that one thread and the `ArrayList` object doesn't need to be thread safe. But if `list` is an instance or class variable, it is shareable (visible in other threads). If your code never creates other threads that access `list`, then the code is correct but is not thread safe. One solution, only allowing objects to be accessed by a single thread, is called ***thread containment*** and is discussed below. This technique is used by

swing. (Another technique is to make `list` immutable, but that isn't always feasible.)

The default assumption in Java is that thread safety isn't needed. This make simple programs simple and efficient. **It is up to the programmer to decide when thread safety is important, and take steps.** (*Show java.util.Collections.synchronized* methods, and java.util.concurrent.Concurrent**.)

Even thread safe programs can have problems in addition to visibility and safety, known as *liveness* and *performance*. ***Liveness*** refers to problems when the program never completes (e.g., *deadlock*). *Performance* refers to having a program complete a task in a timely manner. Other liveness issues include *livelock* and *thread starvation*. Both of these types of issues are caused by the solutions used to provide thread safety, and you must address them in your code.

Summary: *Threads may make programming easier and may make programs run faster, but sharing mutable data between threads can lead to thread interference and memory consistency issues that must be addressed. The solutions to these issues provide thread safety, but those solutions can cause other issues such as liveness and poor performance.*

MapReduce (Fork/Join)

[*Adapted from arstechnica.com/open-source/news/2010/01/googles-mapreduce-patent-what-does-it-mean-for-hadoop.ars*]

"Map" and "reduce" are functional programming primitives that have been used in software development for decades. A "map" operation allows you to apply a function to every item in a sequence, returning a sequence of equal size but with the processed values in place of the originals. A "reduce" operation accumulates the contents of a sequence into a single return value by performing a function that combines each item in the sequence with the return value of the previous iteration. These are used for "divide and conquer" algorithms. (We covered map and reduce earlier in the course, in *aggregate stream operations*.)

The idea is to split ("divide") the data space to be processed by an algorithm into smaller, independent chunks. That is the "map" phase. In turn, once a set of chunks has been processed, partial results can be collected to form the result ("conquer"). This is the "reduce" phase.

A trivial example would be a huge set of integers for which you would like to compute the sum. Since addition is commutative, you can split the set into smaller subsets, where concurrent threads compute partial sums (one thread per subset). The partial sums can then be added to compute the total sum. Because threads can operate independently on different areas of an array for this algorithm, you will see

a clear performance boost on multicore architectures, compared to a mono-thread algorithm that need to iterate over each integer in the collection.

Google's MapReduce framework ("BigTable") is based on those concepts. A series of data elements is processed in a map operation, then combined at the end with a reduce operation to produce the finished output. The advantage of partitioning a workload this way is that it's extremely conducive to parallelization. Each discrete unit of data in the series can be processed individually and combined at the end, making it possible to spread the workload across multiple processors or computers. It's an elegant approach to scalable concurrency, one that offers efficiency regardless of whether your environment is a single multicore processor or a massive grid in a data center.

Google published a paper in 2004 that described how it uses MapReduce. The paper attracted considerable interest and paved the way for the MapReduce pattern to become a common technique for parallelization. One of the most well-known third-party implementations of MapReduce for distributed computing is *Hadoop*, an open source Apache project now used by Yahoo, Amazon, IBM, Facebook, Rackspace, Hulu, the New York Times, and a growing number of other companies. Other MapReduce software includes CouchDB and Nokia's QtConcurrent framework.

Map-Reduce has limitations, because not all map operations are thread-safe.

> Recently, Google was awarded Patent #7,650,331 for MapReduce; it is not clear what might happen if they choose to enforce it. That is unlikely though.
>
> Google has since moved beyond MapReduce, and now (2014) uses Spanner and F1, which scale up as well, but support full SQL and ACID transactions, so the limitations of MapReduce are gone.

Java 7 included some support for this. **`ForkJoinTask`** objects support the creation of subtasks, plus waiting for the subtasks to complete. (Consult the Java docs for specifics, also this fork/join article on Oracle's site.) The `Arrays.parallelSort` method use fork/join internally to take advantage of multiple cores.

Java 8 added significant support for map-reduce, as part of the new streams ("aggregate operations") facilities. It is simple to use: just use a `parallelStream` instead of a normal `stream`. That's all! The system will use map-reduce if possible (that may require extra steps to make happen). Additionally, the `java.utils.Arrays` class has `parallelSort` methods that use fork/join.

GPU Programming

Graphic Processing Units (GPUs) are special types of CPUs, designed to handle the task of rendering millions of tiny triangles thirty times a second. To accomplish this, GPUs are designed to support multithreading, hundreds to thousands of them at once. It turns out that such a design is very useful or a variety of other applications, and GPU programming is becoming more and more popular. You need not worry about GPU programming for our course; the main languages are C and C++, and even then they must use a GPU language too.

A typical GPU contains multiple cores (called MPs). A GPU program (confusingly called a *kernel*) is composed of one or more *threadblocks*, each of which can contain up to 1,024 threads. The threadblocks are put on a queue and dispatched to an MP when one is available. There, the threadblock runs without pause until all its threads terminate.

GPUs also support an additional form of parallelism. For many types of problems, the same code needs to be run on different subsets of the data. (This is sometimes known as SIMD computing.) A thread can be replicated easily in a group called a *warp*, which is typically 32 threads max.

GPUs have no notion of peripheral devices, take special effort to load in the code (kernel) and data, and to pull the results out. Such programming usually uses the OpenCL standard (and library) with C or C++.

Using Java Threads — Thread Creation, Termination, and Thread Types

Early Java had limit ways to work with threads, and those methods were broken! Multithreading simply was not well-understood in the 1990s. Over time, the Java language added additional (and correct) ways to work with threads, and deprecated the broken bits. Every few versions, Java added another new framework (set of classes and interfaces) to work with threads. But old code is often not rewritten from scratch to use the newer ways.

The latest framework was added in Java 21, virtual threads. But entry-level programmers (such as recent graduates) are likely to be assigned old code to maintain; websites including StackOverflow will often show older code. So rather than simplify these notes, I have just added to them over time with the newer methods. I have noted the Java version in which each feature has been introduced.

If you are learning Java by writing new code, or are lucky to be assigned a *greenfield* project (no prior code), I suggest you skip to the end of this section and only worry about older methods if you need to.

In Java, each thread is represented by a `Thread` object (naturally!). These don't do anything by default. To create additional `Thread` objects that do something

useful, either extend class `Thread`, or (better) create an object of any class that implements the `Runnable` interface (often an anonymous class is used) and then pass your `Runnable` object to the `Thread` constructor. The `Runnable` way is more flexible and is used more in practice. Here are examples of both ways:

```
class Foo extends Thread
{  @Override
   public void run ()
   { ... }  // Thread ends when this method returns
}
Foo f = new Foo();
f.start();
```

Using a `Runnable` instead:

```
class Foo implements Runnable
{  @Override
   public void run () { ... }
}
Thread t = new Thread( new Foo() );
t.start();
```

```
Thread t = new Thread( new Runnable(){
   @Override
   public void run () { ... }
});
t.start();
```

Or using a Lambda:

```
Thread t = new Thread( () -> { ... }; );
t.start();
```

A newly created `Thread` doesn't run immediately. Start a `Thread` by calling its **`start()`** method. Once a `Thread` is started, it executes the instructions in its **`run`** method. **The `Thread` is *terminated* when `run` is exited for any reason** (`return`, fall off the end, or an uncaught exception). Of course, other methods can be invoked from `run`.

Once terminated, a `Thread` can't be re-started or reused. However, many different `Thread` objects can be created from the same `Runnable`.

(Show PServer1.java.) Note in this example the class implements `Runnable` and a `Thread` is created from it in the class constructor. No reference to the new `Thread` object appears to be saved.

Suppose instead you have a `Thread t`, and later, while the `Thread` is running, do "`t = null;`". Qu: Is the `Thread` garbage collected and/or stopped? Ans: No, since there is still a reference to the thread kept internally by the JVM to keep track of all live threads in a process.

Starting a Thread in its constructor is not a good idea since the new thread may start executing before the `main` thread (running the constructor) does, and thus the new thread may access properties that haven't been initialized yet! If you do this, be sure the call to `start` the new thread is the last step in the constructor, and that no other classes extend this class (declare it `final`).

Another problem with PServer1 is that the `run` method must be `public`, yet it would not be desirable to allow any other thread to invoke the `run` method. The solution to this is to use *inner classes*. An anonymous inner class or a `private static` inner class can be used to prevent the outside world from invoking `run`. **(Show PServer2.java**.)

Every thread has a unique ID number. You can set and get a name for your threads too. Threads also have a priority (discussed below) and other properties. None of these features are directly used much.

> Java threads are actually just wrappers around your platform's operating system's threads. This imposes significant issues: a limited number of threads, very slow thread creation, and features that work differently (or not at all) depending on your platform. For example, thread priorities. Such issues are discussed below.

Thread Types — User and Daemon

When you use the above code, you have created a *user* thread. These are the normal threads that you commonly use. When you start the JVM, a single user thread called *main* is created.

The JVM will shut down automatically when the last user thread terminates.

Sometimes, you want to start a background thread and forget about it. Examples include threads that display a clock, or play background music. If these were user threads, you would have to manually stop them. That takes more code and is a pain. Instead, Java provides ***daemon*** (not *demon*) threads for such purposes. (A *daemon* is "an attendant spirit", while a *demon* is "an evil being".) The garbage collector is typically a daemon thread.

A running daemon thread won't prevent the JVM from shutting down. Never use a daemon thread that modifies state; it may be killed at any time, even in the middle of some operation!

Before a thread is started, you can invoke `setDaemon(boolean)` to change its type. **Once the thread is running, you can't change the type.**

Thread States

A thread in Java can be in one of six different states. The state determines what the thread can do, what an interrupt will do, and what locks and other resources the thread can hold. Concepts such as locks and interrupts will be discussed later, but Java programmers should know what the states are:

- **New** — Created but not yet started. Such a thread cannot do anything except start.
- **Runnable** — A started thread. Note, all runnable threads take turns running.
- **Blocked** — When a thread attempts to acquire a *lock* that isn't available, it must wait until the lock becomes available. Such a thread is *blocked* and won't run. Once unblocked, the thread becomes runnable again.
- **Waiting** — When a thread needs another thread to complete some action, it becomes inactive until notified. **In this waiting state, the thread gives up its locks.** Once notified, the thread enters the *runnable* state again if the locks it held are available, or the *blocked* state if they aren't.

  > A blocked thread waits for some ***condition***. Until Java 5, each *lock* had one condition, causing many students to confuse the two concepts. Modern Java allows one to create as many conditions per lock as desired. (Conditions are sometimes called *condition variables*.) The idea is instead of waiting on the lock itself, a thread can wait on a condition. For example, you might create a condition *queueNotEmpty* and have a thread that reads from a queue wait on that. The thread that adds items to the queue notifies threads waiting on that condition.
  >
  > Even though using one condition per lock is common, it can help readability to give the locks and conditions separate and descriptive names, such as `accountLock` (to lock a stock broker's customers' accounts) and `metReservePrice` (a condition a thread must wait for, before completing a transaction). You can imagine a single account with multiple stock trades pending, each awaiting a different condition.

- **Timed Waiting** — Some of the methods that cause a thread to enter the waiting state have an optional *time-out* parameter, which will cause the thread to become runnable even if the action it is waiting for never happens.
- **Terminated** — When the `run` method exits for any reason, it terminates. Such a thread may not be garbage collected (at least, not right away), but can never run again. (That is, you can't call `start()` on it.)

Pausing a Thread for Some Amount of Time — Thread.sleep

A thread can be paused with **`Thread.sleep(`*`milliseconds`*`)`** or `sleep(long` *`millis`*`, long` *`nanos`*`)` methods. (Show **HoopsApp.java**.) (It can also be paused with a version of `Object.wait` that takes a time-out argument.)

> If you've forgotten SI (standard international) unit prefixes, milliseconds are a 1,000th of a second, microseconds are a millionth of a second, and nanoseconds are billionths of a second.

A sleeping thread may awaken early. What happens when a sleeping thread gets *interrupted*? The `sleep` method is aborted with an **`InterruptedException`**.

After waking up, a thread might have to wait if other threads are running. Note that while sleeping, a thread doesn't release any locks (discussed later).

`Thread.sleep` can be handy for a *timer* thread that does some work, then waits for a while by sleeping in a loop. When interrupted, the exception that gets thrown aborts the thread. Use code something like this (a common code pattern):

```
public void run ()
{  try {
      for ( ; ; )
      {  while ( ! work_to_do )
         {  sleep( LONG_TIME ); }
         do_work;
      }
   } catch ( InterruptedException e )
   { clean_up;  }
}
```

Sleep duration is not very accurate on some platforms (especially older or cheap ones). Once a sleeping thread wakes up, it may not run right away. If you plan on implementing your own timer threads, it is usually best to sleep for no more than 10 milliseconds and use a loop to ensure you pause long enough:

```
long now = 0,
 stopTime = System.currentTimeMillis()+desiredInterval;
int duration = 10;
while ( now < stopTime )
{  sleep( duration );
   now = System.currentTimeMillis();
   if ( duration > stopTime - now )
      duration = stopTime - now;
}
```

(Using `System.nanoTime()` is usually more accurate.)

`Thread.sleep` can also take a `Duration` object (new in Java 21). Note that `Thread.join` can also take a timeout value as a `Duration`. (New in Java 24 is that the method `Process.waitFor` can take a `Duration` argument too.)

Using Java Timer Classes

"Old school" timers were just `Threads` that would sleep, then wake up and do something, in a loop. To sleep with high precision, use a loop similar to this:

```
void pause ( final int duration )
{  int resolution = 10;  // 10 ms clock resolution
   for (int time=0; time<=duration; time+=resolution)
      try { Thread.sleep( resolution );
      } catch ( InterruptedException e ) {}
}
```

The empty catch clause is commonly seen, but not a good idea. Either handle the exception, or declare your method throws it. One way to handle it might be to set the thread's *interrupted* flag (which is not set when this exception is caught):

```
...
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
}
```

Now the flag can be tested by the calling (probably `run`) method.

Using a thread as a timer was so common in early Java that different groups of developers added timer classes in later Java versions: `jakarta.swing.Timer` and later `java.util.Timer`. (They work completely differently.) Be careful when using wildcard import statements with both these packages!

If you are going to run your application on Windows, be aware that the clock resolution will be only about 54 ms (a frame rate of around 18 fps) in Windows 7 and earlier. The default timer interval or resolution in Windows 10 (in 2021) is 15.625 ms (1,000 ms divided by 64). That means that if your program tries to `sleep(1)` (pause for 1 millisecond) at some random time, it will probably be woken-up sometime between 1.0 ms and 16.625 ms in the future, whenever the next interrupt fires (or the one after that if the next interrupt is too soon). What's really bad is that this resolution can be changed by any program, even though it is a global value in Windows. So you cannot count on this!

(On my PC with no other programs running, measurements show the interval (granularity) is indeed about 15.5 ms. Running my web browser at the same time changes this to about 2.5 ms. Running the Zoom client changes it to about 1 ms.)

It is easy to see this effect on most MS Windows systems if you run a short program (`TimerRes.java`) that retrieves the system clock using `currentTimeMillis()` in a tight loop. Have the program count the number of times the value returned by `currentTimeMillis()` changes in one minute and divide that by 60. You'll see that the value changes from about 65 times a second to 999 times a second, depending on what else is running!

It's worse for Windows 11! MS enables by default ProcessPowerThrottling which can lower the resolution of non-foreground tasks (i.e., if you minimize the window, or have another app's window(s) on top).

Some systems (Mac or Intel systems running a non-Windows OS) have higher (and more stable) system clock resolutions.

Swing timers are easy to use to update Swing GUIs. Here's an example of using the Swing `Timer` (Beep the time every 10 sec):

```
Timer t = new Timer(10_000, new ActionListener(){
 public void actionPerformed ( ActionEvent e ) {
   System.out.println( "At the tone, the time "
     + "will be " + new Date() );
   Toolkit.getDefaultToolkit().beep();
 }
});
t.start();
```

Note the action is done on the event handling thread, so it must execute quickly or your GUI will appear unresponsive. Also, nothing happens if the event handling thread isn't running, so this demo only works if there is a GUI created. However, it is simple to use. In the above code, the `jakarta.swing.Timer` sends `ActionEvents` to the registered listener (here an anonymous inner class) every second. The work will be done on the AWT event handling thread so you can safely update your Swing GUI. However, the task performed must execute quickly or your user interface will suffer.

Show animation using timer (SSJava) and threads (Bouncing Hoops).

The more general `java.util.Timer` has more features and control, doesn't require the GUI be running, and doesn't use the event handling thread (so your tasks can take as long as necessary). Here's an example:

```
public static void main ( String [] args ) {
  Timer t = new Timer();  // java.util.Timer
  TimerTask task = new TimerTask() {
    @Override
    public void run () {
      System.out.println( "At the tone, the time "
        + "will be " + new Date() );
      Toolkit.getDefaultToolkit().beep();
    }
  };
  t.scheduleAtFixedRate( task, 0, 10_000 );
}
```

Finally, the `concurrent` package (added in Java 5) has a more featureful class which can be used as a replacement for `java.util.Timer`. It is called **`ScheduledThreadPoolExecutor`**. Here's a demo:

```
public static void main ( String [] args ) {
  ScheduledThreadPoolExecutor timer =
    new ScheduledThreadPoolExecutor( 1 );
  Runnable task = new Runnable() {
    @Override
    public void run () {
      System.out.println( "At the tone, the time "
        + "will be " + new Date() );
      Toolkit.getDefaultToolkit().beep();
    }
  };
  timer.scheduleAtFixedRate( task, 0, 10,
     TimeUnit.SECONDS);
}
```

This code creates a pool of one thread. That thread runs a single task every 10 seconds (after an initial delay of zero seconds). The `scheduleAtFixedRate` method returns an object, a `Future`, that can be used to cancel the task.

A `ScheduledThreadPoolExecutor` work better than a `java.util.Timer` when you have multiple "worker" threads that can execute the tasks, or when tasks need to return a value.

The type of thread the timer classes use, *daemon* or *user*, varies. Swing timers apparently use one *daemon* thread and that will not prevent a program from terminating. Also, as events are run on the EDT. So if no GUI

component is displayed, that thread does not start and none of your tasks run!

The `java.util.Timer` class uses a *user* thread by default, but you can force it to use a *daemon* thread instead.

Waiting for a Thread to Terminate — Thread.join

One Thread (`t1`) can wait for another (`t2`) to finish before continuing by having `t1` run the code `t2.join()`. Here's an example:

```
class Foo { private bkgrndThread bt = ...;
   Foo() { bt.start(); ... }
   doWork () { bt.join();  showMedia(); }
}
```

Qu: What would happen if `wait` or `join` were used in an AWT Component event handler such as `actionPerformed`? Ans: the event dispatch thread pauses, locking up your GUI. (*Demo Oops.java.*)

Stopping a Thread — Thread Cancellation

It is very dangerous to stop (or *terminate*) a thread from the outside; there's no way to know what the thread is currently doing! (Imagine stopping a thread in the middle of a money transfer.) Originally Java had a method for this but it is deprecated. (Show Java 8 doc for Thread.stop/suspend/resume deprecation; note some deprecated methods have been removed since Java 11.)

The correct technique is to tell a thread that you want it to stop. This is done in any number of ways, such as by setting a `boolean` flag (a *field*) to `true`. The thread should periodically check this flag, when it is safe to do so. When it sees the flag is set, the thread should stop as soon as it safely can. (See **Threads.java**.)

Summary: *Create a thread by passing a Runnable to a Thread constructor. This creates a user thread. Use setDaemon(true) to create a daemon thread. Then use Thread.start to have it execute its run method. One thread can wait for another using join. A thread can pause using sleep. It is dangerous for one thread to pause or stop ("cancel") another, so instead have one thread set a flag the other checks periodically.*

Multi-Threading Issues — Visibility, Safety, Liveness (and Performance)

Threads that don't share access to common resources (independent, asynchronous threads) have no issues and do not need to be thread-safe. It is also fine to share immutable (read-only) objects. A lot of designs do this, including multi-threaded web servers (a main listening thread, which starts a new worker thread with (read-only) connection information).

Those that do share access to writable objects (which is common) have a number of potential problems, **all ultimately caused by *thread interference* and the *memory consistency* issues of modern computer hardware.**

Visibility Issues

If two or more threads share access to some object's field (or to files or some other resources, but don't worry about that here), it is possible for one thread to update that field and other threads may never see the new value, only the old one. This is called a *visibility* issue.

Non-volatile `longs` and `doubles` can't be read/written in a single, *atomic* (indivisible as far as other threads are concerned) operation. It is thus possible for the data to get corrupted. (Half is updated, then the whole is read by another thread.) Use `volatile long` (or `double`) to have reads and writes be atomic (as 32-bit primitives inherently are).

If your data is read only (immutable), there is no problem with sharing access. So using only immutable (or constant primitive) data will solve the problem.

Sadly, in real applications you may need to share mutable data. Several techniques exist to handle this problem, including using `volatile`, using atomic variables, and the use of synchronize blocks.

In the absence of synchronization, some code can be optimized in a way that causes visibility and liveness problems. For example:

```
while ( ! done ) ++i;
```

(where `done` is set by another thread) might (and in Sun's *HotSpot* JVM will) be *hoisted* (a type of optimization). The resulting byte code does this:

```
if ( ! done ) while ( true ) ++i;
```

Thread Safety Issues

A *thread safe* object (or any shared data) is one that can safely be accessed by more than one thread at a time. Because of the problems of thread interference and of memory consistency, objects aren't thread-safe by default.

There are four ways to address this issue (besides ignoring it and living with the crashes and lock-ups): *use immutable objects, thread containment, use atomic operations only,* and *synchronize* (coordinate) access. Of these four, the last (using synchronized code) is the hardest and can cause liveness problems.

The thread safety of a class is vital information that **should be put into the Java doc comments** for any reusable classes, and any special thread safety issues should be documented on the methods that need them.

It is recommended that you *label* classes in the doc comments with one of: *immutable*, *thread safe*, *conditionally thread safe* (some methods will require external synchronization), and *not thread safe*. In his book *Java Concurrency in Practice*, Goetz recommends creating annotations for these categories and using them to label your classes and methods.

Race Condition or Race Hazard

The main problem with sharing access to mutable objects (aside from the visibility issues discussed above) is that **reading, updating, and writing an object's fields require multiple steps**. But the sequence of steps done by a thread can be interrupted at any point and a different thread may access the data before the first thread resumes and finishes. This interleaving of instruction sequences is known as a ***race condition*** or *race hazard*. (This problem is exactly why database systems use transactions.)

A ***race condition*** is present when the correctness of a program depends on the relative timing of (or the order of interleaving of) multiple threads by the runtime.

Example: Two or more threads incrementing a shared counter in the order T1: read current val; T2: read current val; T2: increment; T1: increment:

T1:	inst 1; (read val)	T2:	inst 1; (read val)
	inst 2; (incr)		inst 2; (incr)
	inst 3; (store val)		inst 3; (store val)

Java guarantees certain operations are *atomic*, including reading or writing an `int`. Sadly, most applications need to read-modify-write, requiring multiple operations, so the sequence of instructions as a whole is not atomic. Modern Java includes new classes with additional atomic operations.

Potentially interfering code fragments (often whole method bodies) are known as ***critical sections*** or ***critical regions***. As with database transactions, a critical region must appear to be a single (or *atomic*) operation as far as the other threads are concerned. Any code that accesses the same shared data is a critical region for that data.

It is up to the developer to ensure that only a single `Thread` is executing statements from any of some shared resource's critical regions at any time.

Consider the example above again. If T2 can't access the value while T1 is in the middle of its sequence of (three) steps, and vice-versa, then no corruption can occur. This requires some sort of synchronization of the threads.

Only after one thread has finished all the statements in a critical region for some piece of data is another thread allowed to start executing code from a (the same or

another) critical region for that piece of data. If the JVM pauses T1 in the middle, T2 must be blocked (prevented) from running code in a critical region (but can run other code). Eventually T1 will finish and T2 will have a chance to use the data.

Every shared data resource is a potential trouble spot. Access to each shared resource requires a separate set of critical regions. That is, a given shared resource may be accessed/modified from several different places (different methods), and each such block of code is a critical region for that shared resource. Worse, sometimes a group of resources must be updated atomically. (Example: Transferring money from checking to savings is not atomic since the system must decrement one account balance variable and increment the other variable. T2 must be prevented from accessing either account until T1 is finished updating both.) **(**Show **Bank.java.)**

While no two threads can be allowed to access the same resource at the same time, a given `Thread` or `Thread`s may execute code from two (or more) critical sections at the same time, if they are critical sections for two different resources.

Qu: Is there a problem if many threads only read a shared variable? Ans: No.

Qu: Is there a problem if only one thread is writing and all others are just reading? Ans: Maybe, if the write is not ***atomic***. Corruption can occur if one `Thread` writes while another reads, if the read operation takes two or more cycles (this is a *non-atomic* read operation).

Updating a single `int` is atomic, but updating a `long`, `double`, or `Object` is not. It might happen that when a `long` is read, the first 32 bits are the old value but the next 32 bits are from the new value. This usually results in garbage.

> The `java.util.Collections` class has methods that can return a thread-safe version of a collection. It does this by wrapping a collection, and the wrapper's methods are all synchronized; almost certainly by locking the data in each method, and releasing the lock when that method exits. This won't work if you have other references to the collection and use them. Consider using the newer `java.concurrent.Concurrent*` classes and interfaces.

Liveness and Performance Issues

Thread liveness is the complement of thread safety. In a nutshell, thread safety means that nothing bad (data corruption) can happen, while **thread liveness means something good will eventually happen**. If not carefully designed, a thread can get stuck waiting for events that never occur. Other threads may in turn wait for this thread and eventually all threads end up blocked! This situation (where each thread is waiting for the others to finish) is called ***deadlock***.

A thread should never pause an application by blocking on some external event (such as user input) while within a critical section (that is, while holding locks). This type of bad code is common in some operating systems, which is why they "lock up".

One common way to deal with this problem is to use the *observer pattern* (used to load and display `Images`). A second thread (the observer) does the waiting, and notifies the first thread only when the event has occurred.

Livelock is similar to deadlock except the threads aren't blocked; instead they keep trying some operation (such as accessing a shared resource) that always fails, usually in a loop. Imagine a DB transaction that always fails and gets rolled-back. Now assume the thread keeps trying until the transaction succeeds. While the thread isn't actually blocked, it won't do any useful work either!

Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads that hog the resource for (very) long periods of time. Starvation can also happen when there are a lot of high-priority threads competing with some low-priority threads; the low priority ones may never run and are "starved".

Performance is related to liveness, in that there can be a lot of overhead with a multi-threaded program. If not carefully designed, a program may not work quickly enough to satisfy performance requirements.

***Summary**: Threads that access a shared object or other data require careful design to avoid the issues of data visibility, race conditions, and liveness. Any sequence of statements that access or modify shared data form a critical region, and those must execute atomically (synchronized) insofar as other threads are concerned.*

Solutions to Multi-Threading Issues:

Using Immutable Data

One way to ensure thread safety is to **make objects immutable**. Sometimes however such an object needs to be replaced with a new one, and the reference to it must be shared and updated. The reference must be made thread safe using a different technique.

Thread Containment

Another method (used by swing) is called ***thread containment***. With this method you don't worry about thread safe objects at all! Instead **ensure all objects are accessed by a single thread only**. The swing GUI uses this method; all GUI component updates (changes to the GUI) must be performed by the AWT event handling thread only (also known as the ***event dispatch thread***).

Use **`jakarta.swing.SwingUtilities.invokeAndWait()`** and **`invokeLater()`** (or the equivalent methods `java.awt.EventQueue.invokeAndWait()` and `java.awt.EventQueue.invokeLater()`) to execute the code in some `Runnable` on the event dispatch thread. The `invokeLater` method adds an event to run the code to the end of the queue. The `invokeAndWait` method is the same except it blocks the current thread until after the update is done (so don't call this method from a GUI event handler or you will lock up your app's GUI).

The problem with these methods is that you shouldn't try to do long running tasks on the event dispatch thread (EDT) or the user interface will appear choppy or even lock up. The correct technique is to do lengthy updates from a new thread, but you still must have the GUI update done on the event dispatch thread.

In Java 6, the class **`SwingWorker`** was added to make this task simpler.

Declaring Data volatile

Although the common case of a `Thread` critical section doing a "get-modify-set" sequence requires the use of `synchronized` blocks (discussed below), there are simpler cases when the overhead and headaches can be avoided. If access is atomic from all threads, you only need to worry about visibility issues. Consider:

```
class Thermostat
{  int currentTemp;  // set by another Thread.
   ...
public void run ()
{  for ( ;; )
   {  if ( currentTemp < lowThreshold )
         setHeater( true );  // Turn up the heat.
      Thread.sleep( 1000 * 60 * 3 );  // Wait 3 min.
}  }
}
```

This `Thread` only reads `currentTemp`, while some other `Thread` only sets it. Also note the `currentTemp` is an `int`, so no corruption is possible (it is atomic to get/set an `int`). However, this code may not work! The JVM may think `currentTemp` is unchanged inside `run` and never check the RAM location, not realizing that some other code might change this variable. (This assumption is never made by the compiler when a `synchronized` block is used.)

The solution is to declare `currentTemp` as **`volatile`**, which forces the JVM to fetch the value of the variable from RAM every time through the loop. This avoids visibility (stale data) problems.

> A `volatile` object reference means the reference is declared `volatile`, not the fields of the object. That is rarely useful.

Atomic Variables (since Java 5)

Often multiple threads need access to a single shared variable and only require simple operations, such as incrementing a counter or updating a value. As of Java 5, all `volatile` primitive and reference variables are read and written atomically and thus inherently thread-safe. But incrementing a variable is **not** atomic! (That's because the data must be read into the CPU, then worked on, then the updated value must be written back to memory. Other threads might modify the value after your thread reads it but before it writes it, so one update may be lost.)

Another way to ensure thread safety is to **ensure all operations are *atomic***. In the past, the only choice was to use `synchronized` blocks to make some object's getter and setter methods atomic, which slows down programs significantly. But such objects are thread-safe and (relatively) easy to use.

Today, all modern processors have instructions for updating some data in a way that can either detect or prevent concurrent access from other processors. These instructions are called ***compare-and-swap* or CAS**. Java now includes a whole slew of new thread-safe classes that contain (atomic) methods for common operations. These use CAS if available on your hardware and are much more efficient than using objects with `synchronized` methods!

> A CAS operation includes three parameters: a memory location, the expected old value, and a new value. The processor will update the location to the new value **if** the value that is there matches the expected old value; otherwise it will do nothing. After the atomic update, it will return the value that was at that location prior to the CAS instruction. The check and update form a single atomic operation.

> An example way to use CAS for synchronization is as follows:

```
public int increment () {
   int oldValue = value.getValue();
   int newValue = oldValue + 1;
   while (value.compareAndSwap(oldValue,newValue)!=
     oldValue)
      oldValue = value.getValue(); //repeat until done
```

```
    return newValue;
}
```

First we read a value from the `value`, then perform a multi-step computation to derive a new value (this example is just increasing by one), and then use CAS to change the value of `value` from `oldValue` to the `newValue`. The CAS succeeds if the value at address has not been changed in the meantime. If another thread did modify the variable at the same time, the CAS operation will fail, but detect it and retry it in a `while` loop.

The best thing about CAS is that it is implemented in hardware and is extremely efficient. If 100 threads execute this `increment()` method at the same time, in the worst case each thread will have to retry at worst 99 times before the increment is complete. (By comparison, acquiring/releasing a lock can take thousands of instructions per thread.)

You can find these classes in the `java.util.concurrent.atomic` package: `AtomicInteger`, `AtomicLong`, `AtomicReference`, `AtomicBoolean`, array forms of atomic integer (`AtomicIntegerArray`), and various atomic reference classes. (Different CPUs may support other types with CAS but Java cannot use them.)

With `AtomicIntegerArray`, `AtomicLongArray`, and `AtomicReferenceArray`, you cannot access the entire array atomically. (You still need an `AtomicReference` for that). But you can access each of the elements in the array atomically.

`AtomicInteger[]` would do the job as well, but it might be unwanted because it would need many allocations, one for each `AtomicInteger` object. However, `AtomicIntegerArray` is more likely to cause a performance issue called *false sharing* (see box below), which can be orders of magnitude slower than the additional allocations of `AtomicInterger[]`. If performance matters, you'll have to test your application to see which way works best.

False Sharing is a problem causes by hardware caches shared by multiple threads. Access to the cache is not by a byte or by a 4 (or 8) byte word. The width of a cache entry is called a *cache line***.** If any part of the cache line is updated, all of it must be reloaded from main memory.

False sharing occurs when threads running on different processors modify variables that reside on the same cache line. This is called false sharing because each thread is not actually sharing access to the same variable. So when two unrelated items share a cache line, the second must be updated

when the first is, even if there is no logical reason for it. This can degrade performance significantly.

Using the atomic package is easy. Here's an example of an integer counter with some atomic methods:

```
import java.util.concurrent.atomic.*;
class Counter {
   private AtomicInteger value = new AtomicInteger(0);
   public int increment () {
     return value.getAndIncrement();
   }
   public int get () { return value.get(); }
```

You probably don't even need a special class `Counter` in this case; it's a useless wrapper class. (Show `AtomicInteger` Java doc.)

Here's another example. Assume a server allows students to enroll at HCC on the Web, and that multiple students may register simultaneously (each from a different thread). How do you assign unique Student ID numbers? This sounds easy, just have a "`nextID`" `static` field. But access must be atomic and visible. Not so easy now! Or is it? Since Java 5 we can do this:

```
class Student {
 private static AtomicLong nextID =
  new AtomicLong(10000);  // or read from file or DB
 public long assignID() {
  return nextID.getAndIncrement(); //look ma no locks
 }
}
```

Maintaining a single count or sum that is updated by possibly many threads is a common problem. While solved by using atomic variables, the guarantees that gives come with a cost.

In some cases, applications can sacrifice 100% accuracy for performance. Java 8 introduces scalable updatable variable support through a small set of new classes: `DoubleAccumulator`, `DoubleAdder`, `LongAccumulator`, and `LongAdder`. These classes internally employ contention-reduction techniques that provide huge throughput improvements as compared to Atomic* variables. This is made possible by relaxing atomicity guarantees in a way that is acceptable in many applications.

Using Synchronization and Locks

The final method to ensure thread safety is to coordinate the activities of multiple threads that share access to some object. This is known as ***synchronization***, and is

one of the oldest supported mechanisms in Java. (See later for newer frameworks that are a lot easier to use.)

Synchronization is done by creating a *lock* object (locks are sometimes known as *monitors*) to protect (*guard*) each mutable resource (or set of resources) that has shared access. Then any code you write that access the shared resource, from any thread, must do these steps:

acquire the lock	**`synchronize( lock ) {`**
critical region	*critical region (do stuff with the shared resource)*
release the lock	**`}`**

The code that accesses the object(s) appears atomic to other threads, since only one thread can hold a given lock at a time. In addition, **using synchronized blocks works like volatile, preventing visibility problems.**

> Mutable thread-safe classes such as `Vector` use synchronization for each public method all the time, even if not multi-threading. But the locking code, while simple to use, takes a long time to run. The other methods for ensuring thread safety should be used instead, if reasonable.

When a synchronized block exits for any reason, the `Thread` will *release* the lock. (In some languages, you must explicitly release locks, but not in Java!)

When extending a class and overriding a `synchronized` method, you are not required to have the new method synchronized, unless it needs to be.

Until recently, Java didn't have special lock classes. Instead, **any object can be used as a lock, even the mutable object itself (which is common).** (The Java documentation calls this a *monitor lock* or just a *monitor*.)

Here's an example of a thread-safe integer counter using synchronized blocks:

```
public class Counter {
   private int counter = 0;
   public int get () {
      synchronized( this ) {
         return counter;
   }  }
   public void increment () {
      increment( 1 );
   }
   public void increment ( int amount ) {
      synchronized( this ) {
         counter += amount;
}  }  }
```

In the code above, the whole method bodies are *critical regions*. In such cases, **Java allows you to use `synchronized` keyword as a modifier on the method; the object itself (`this`) is then used as the lock automatically:**

```
public synchronized int get () { return counter; }
```

(`static` methods will use the `Class` object for the lock instead.)

> The larger your critical regions are the slower the application will be. **Keep synchronized blocks as short as possible.**

A thread never blocks by trying to acquire a lock it already holds. This feature is called *reentrant synchronization*. So there is no problem invoking one synchronized method from another. (Note how much simpler it would be to just use atomic data if possible.)

The `java.concurrent.*` Packages

An old-school (pre-Java 5) Java lock only allows a single thread to acquire it at a time, so only one thread can use a shared resource at a time. Such a lock is sometimes called a *mutex*, short for "mutually exclusive".) But what if you have performance issues instead of safety issues, and you need to limit the number of threads concurrently using some resource to a (small) number greater than one? (Ex: server with pool of 50 DB connections.) You don't need mutual exclusion in such cases; if you can safely have 50 DB connections simultaneously then a mutex lock is no good as it will limit your code to a single connection at a time.

You can use a type of lock called a *counting semaphore*. A **`Semaphore`** can be used to easily limit the number of threads accessing some shared resource. The semaphore is initialized with a maximum count. Each time someone "acquires the lock" the count is decremented, and each time the lock is released it is incremented. A thread attempting to acquire the lock will only block if the count becomes zero. Here's a class that uses a semaphore to control access to a pool of items, taken from Java docs):

```
class Pool {
  private static final int MAX_AVAILABLE = 100;
  private final Semaphore available =
      new Semaphore(MAX_AVAILABLE, true);
  public Object getItem() throws InterruptedException
  {
    available.acquire();
    return getNextAvailableItem();
  }
  public void putItem(Object x) {
    if (markAsUnused(x))
      available.release();
  }
```

`java.util.concurrent.locks` package (Java5) defines more powerful locks than the (relatively) easy to use ones (mutex) locks) built into class `Object`. Using them makes certain common complex tasks a bit easier. The new `Lock` objects work very much like the implicit *monitor* locks used by `synchronized` code: only one thread can own a `Lock` object at a time. `Lock` objects also support a *wait/notify* mechanism to permit one thread to signal another when it is done with the shared resource (discussed below, page 427).

Unlike monitor locks, `Lock` objects can *back out* of an attempt to acquire a lock (that is, they won't block the thread if the resource is busy). The **`tryLock`** method backs out if the lock is not available immediately or before an optional timeout expires.

The package also contains some special purpose locks that elegantly solve some difficult problems, such as allowing multiple reader threads but only one writer thread at a time, and not having readers or writers starve each other. (This is a well-known but complex problem that would otherwise require two or more locks for the shared resource!)

Here's one example of using the new locks:

```
public class BankAccount
{  private Lock balanceLock;
   private double balance = 0;
   public BankAccount ()
   {  balanceLock = new ReentrantLock();
      ...
   }
   public void deposit ( double amount )
   {  try {  // Critical region for balance:
         balanceLock.lock();
```

```
                balance += amount;
            } finally {  balanceLock.unlock();  }
        }  }
```

The `try-finally` block is needed since an `Exception` would abort the method and never call `unlock`. Also, while this is more complex than just using `synchronized`, it adds many useful features such as multiple *conditions* you can wait for (using `await` and `signalAll`), and the ability to test if someone holds a lock.

Although the `ReentrantLock` is the most commonly used, others are also provided for other fairly common situations that are tricky to implement correctly just using `synchronized` such as `ReadWriteLock`.

> As mentioned previously, the new `Lock` objects can have one or more condition variables, and threads can wait on a specific condition before unblocking.

Avoiding Deadlock

Deadlock (a.k.a. *deadly embrace*) is a liveness issue that can occur whenever you have two or more threads and two or more locks. (See **Dining Philosophers** – Sun's Deadlock Demo.) Ex: `Thread` one has lock A and tries to acquire lock B. Meanwhile `Thread` two has lock B and tries to acquire lock A. Both `Threads` end up waiting forever.

The common technique used to avoid potential deadlock in your Java program is known as ***resource ordering***. This just means that all `Threads` must acquire locks in the same order. (`Threads` one and two must both acquire lock A before attempting to acquire lock B.) There are other possible solutions as well.

> Other liveness issues include *livelock* and unacceptable performance. There are mechanisms and code idioms to mitigate such issues.

Lock-Free Synchronization

Locks are a (relatively) simple programming model to use. But they are not without problems as noted above. Code with many shared mutable resources (and thus many locks) can be confusing. They often have liveness issues and can cause very bad performance. In some cases when using modern hardware, it is possible to avoid locks entirely. Lock-free algorithms rely on atomic operations and atomic references to objects.

A classic design pattern is to use a shared hash table for quick lookups of objects. Such a *locator* will have many readers and a few writers. Using a classic lock would cause a significant performance penalty. Can this be coded without locks? Yes; the trick is to use immutable (and therefore intrinsically thread-safe) objects,

so the hash table returns references to the former or latest object, never a corrupted one. But, when updating the hash table, if another thread tries to do a lookup, it may get false data.

You can make such a locator table thread-safe with an atomic reference to a `Hashtable`. Once a `Hashtable` instance is exposed to the other threads by this atomic reference, it must not be modified any more. Thus all read operations are safe without locks. If the `Hashtable` is to be modified, you must first make a copy, modify that, and then the atomic reference is switched to the new revision. If there are multiple writers, they can be synchronized by a loop using CAS operations, or simply by a synchronized block.

The only thing you have to care about is that the atomic reference does not provide repeatable read semantics; one read might be from a different version of the hash table. But a thread can easily take a snapshot of the locator, by simply making a local reference.

***Summary**: The techniques available to deal with thread safety and visibility issues include thread containment, immutable data, volatile fields, atomic (thread safe) objects, and using synchronization techniques including locks. Each technique comes with its own issues and trade-offs.*

Communication Between Threads

Inter-thread communications are a way for one thread to tell another something. There are a number of ways for one thread to tell another to pause, to wake up and go to work, and to terminate. Some of these are discussed below.

> Some languages don't allow threads to share any memory. Instead, such threads communicate by a technique called *message passing* which is inherently thread safe. However there is often a performance cost. Java uses shared memory plus just a few other simply ways for threads to communicate; these are usually just used to coordinate thread activities.

Using interrupts

An *interrupt* is an indication to a thread that it should stop what it is doing and do something else (e.g., a phone rings while setting the table for diner). Usually (but not always) you interrupt a thread to tell it to terminate ASAP. To interrupt a thread `t`, invoke (from another thread) **`t.interrupt()`**. If the thread `t` is running the only effect is to set the *interrupted status* flag of `t` to `true`. If a thread `t` is waiting or sleeping when another thread invokes `t.interrupt()`, `t` will wake up with an `InterruptedException` instead (and the *interrupted status* flag will be set to `false`). (If the thread was blocked for I/O at the time, the flag is set to `true` and a `ClosedByInterruptException` is thrown.)

Inside thread `t`'s `run` method, you check the *interrupted status* flag using the `static` **`Thread.interrupted`**`()` method:

```
public void run ()
{  while (work_to_do && ! Thread.interrupted() )
   {  do_something;  }
   clean_up;
}
```

> The `static` method `Thread.interrupted()` also resets the *interrupted status* flag. Use the non-static `Thread.isInterrupted()` method if you want to check the flag but not reset it if it was set.

Note that **if a thread detects it has been interrupted, it doesn't have to terminate**. A thread can do what it likes, including ignoring the interrupt or checking some other variable (field) to see what it should do next.

Using wait and notify

Using locks correctly prevents threads from interfering with each other, but there are times when two or more threads need to communicate. **Consider when one *consumer* thread waits for work to arrive in a queue and processes it, and a *producer* thread adds work items to the queue.** This queue is a shared resource so must be protected by a lock. But if the queue is empty and the consumer uses `Thread.sleep`, the producer can't ever add work to the queue! (The consumer still has the lock.) A way is needed to have the consumer thread release the lock (at a safe point) and simultaneously go to sleep, to wait until there is something for it to do.

In Java this is done by having one or more consumer threads **`wait`** for something to be done, and a producer thread using **`notify`** or **`notifyAll`** to alert the waiters that it's time for them to proceed. (Wait can also be invoked with a timeout (similar to sleep), to self-notify after the timer goes off.)

What's happening is that the lock built into every Java object also has one built-in ***condition variable***. This allows a thread to release its lock and *block* (sleep until awoken by another thread *notifying* to any threads waiting on that condition). Some other thread can access the resource, then wake up all threads that were blocked, using that condition.

The standard idiom to use is:

```
synchronized void doWork()
{  while ( ! ready_to_do_something )
      wait();  // waiting on the condition
   doTheWork;
}
```

1 Everything runs from inside a `synchronized` block. This is essential; all conditions are associated with some lock.

2 When wait executes, the `Thread` both pauses **and** gives up its lock. This occurs *atomically*. After waking up again (i.e., after `wait` returns), the `Thread` must re-acquire its lock before it can proceed. (Qu: What's wrong with this code?)

```
synchronized void transferFunds ( float amount) {
   this.checkingBalance -= amount;
   wait();  // can't safely wait here!
   this.savingsAccount += amount;
}
```

3 **The test on the condition should always be in a loop, never an `if`.** It is entirely possible to return from `wait` and not have the condition satisfied. (For example, if an exception is thrown.)

Producer threads execute code like this when there is more work to do:

```
synchronized void readyToWork()
{  setUpSharedResourcesForWork;
   notifyAll();
}
```

`notifyAll` wakes up all threads waiting on that condition. After one waiting threads acquires the lock again, it goes to work; the ones not involved will go back to waiting.

When using `wait` and `notify`, it is common to use a `try` block in the loop to catch and ignore any `InterruptedException`s that may occur.

More realistic *consumer* code needs to check if it should stop. Use something similar to this idiomatic code:

```
public void run ()
{  while ( ! time_to_stop ) {
      if ( work_to_do )
         do_something();
      else
         this.wait();  // Wait until notified
   }
   clean_up;
}
```

The *time_to_stop* field might be a `boolean` instance variable. When the main thread decides to stop the server, it can set *time_to_stop* to `true`. After each job is completed the server thread checks this field, so it won't stop in the middle of

some job. (Qu: should you use `boolean`, `AtomicBoolean`, or a `volatile boolean` for this? Ans: With a single writer and multiple readers, a `volatile boolean` is good; if there were multiple writers of the boolean value, use `AtomicBoolean`.)

There is a version of `wait` that takes a timeout parameter similar to `sleep`.

It is possible to use **`notify`** instead of **`notifyAll`**. `Notify` wakes up a single waiting thread only. However you never know which one gets woken up! Using `notify` instead of `notifyAll` is an optimization that works only when:

- All Threads sharing one lock are waiting for the same condition
- At most one thread can benefit from the condition being met (i.e., only one thread can work)
- The above must be true of all possible subclasses (`final` classes are your friend)

Working with `wait` and `notifyAll` can be tricky but there are standard idioms (or design patterns) for using them in common situations. These have names such as *producer-consumer* and *readers-writers*.

Using a `ReentrantLock` (new in Java 5) you can create any number of conditions from a lock. Instead of using `wait`, `notifyAll`, and `notify`, you instead use the `Lock` methods `await`, `signalAll`, and `signal`. Using the new locks, you can give conditions meaningful names, making your code easier to read and understand.

Consider the `BankAccount` class above. You might want to add a `withdraw` method like this (not really, but it makes for a simple example):

```
public class BankAccount
{  private Lock balanceLock;
   private Condition sufficientFunds;
   private double balance = 0;
   public BankAccount ()
   {  balanceLock = new ReentrantLock();
      sufficientFunds = balanceLock.newCondition();
      ...
   }
   public void withdraw ( double amount )
   {  try
      {  balanceLock.lock();
         while ( balance < amount )
            sufficientfunds.await();
         balance -= amount;
```

```
            } finally { balanceLock.unlock(); }
        }
        public void deposit ( double amount )
        {  try
           {  // Critical section for balance:
              balanceLock.lock();
              balance += amount;
              sufficientfunds.signalAll();
           } finally {  balanceLock.unlock();  }
        }
    }
```

Every time the main thread finds more work to do it can (for example) add it to a `List` or `Queue` in the server thread, and then run `this.notify` (or `notifyAll`). (The `List` would be an obvious choice for a lock object to use.) This will wake up the server thread if it was waiting and have no effect otherwise.

Summary: *Besides sharing access to memory (fields of objects) and files, one thread can wait for (join) another to terminate, interrupt a thread, or use wait and notify mechanisms on conditions. This is common for producer-consumer designs.*

Other Java Multi-Threading Features:

Threads and Exceptions

Exceptions occur in a specific thread. If an uncaught exception causes `run` to exit, the thread dies and the exception is lost. (So a try block around a `someThread.start()` call does nothing!) What happens is the method **`uncaughtException`** in the dying `Thread` is invoked. By default this displays a stack trace to `System.err`, but you can override this behavior by the `Thread.setUncaughtExceptionHandler()` method.

> This method is only since Java 5. For older Java you needed to extend `ThreadGroup` and override the `uncaughtException` method, and use the `Thread` constructors that take a `ThreadGroup` argument). Yuck! Thankfully, the `ThreadGroup` API was removed in Java 14.

Thread Scheduling and Priorities

Thread scheduling relies on the underlying system, so there are few guarantees. Generally, threads with a higher priority will run before (or perhaps more often than) threads with lower priorities. On some systems, a ready-to-run thread with higher priority can *preempt* a running thread with lower priority (which may never run again, the problem known as *starvation*), or it may run but not as often as the higher priority threads (a feature known as *priority aging* mitigates this.) This

depends on the underlying operating system; starvation can be a common "gotcha".

Typically, all `Threads` have **`NORM_PRIORITY`**, except for the AWT event handling thread, which has a higher priority. Note that when creating a new thread, it will inherit the priority of the current thread. This would be a problem if you create a new thread in some listener (event handler) method, which get run by the AWT event handling thread. In this case, use `setPriority(Thread.NORM_PRIORITY);`

The priority can be set to any value in the range `MIN_PRIORITY` to `MAX_PRIORITY`, with a default of `NORM_PRIORITY`. You can set background thread priorities to `NORM_PRIORITY-1` and user interface thread priority to `NORM_PRIORITY+1`, but usually it isn't necessary to adjust the priority.

> The problem with priorities is that the JVM uses the underlying OS to schedule the threads, and different systems work differently. Windows 7 has seven priority levels, but Java thread priorities are ignored on Linux systems. Some systems may permit starvation, others permit priority aging.
> **Changing thread priorities is usually unnecessary and useless, and may be dangerous.**

A thread can provide a hint to the JVM (the part known as the *scheduler*) that it's okay to switch to another thread at this point (**`Thread.yield()`**). On some systems no thread will preempt another, so you must call `yield` inside of the loop in the `run` method or the thread will run (and blocking all others) until it exits.

Thread Names

The following are all `public void` methods of class `Thread`:

`getName(), setName( String` ***`name`*** **`)`** – `Thread` names are useful when debugging, or to determine which thread is currently running. The name can be passes as a `Thread` constructor argument too. The default name is "`Thread-`*`n`*", where *n* is an integer.

`static currentThread()` – returns a reference to the current `Thread`.

ThreadLocal Variables

Sharing the properties of a class between all the threads is often useful, but there are times when you'd like a field of an object set that has a different value for each thread. For example: the socket (or IP address) to use for a telnet or web server thread (with one thread per request), a user ID for a DBMS thread or a chat server, etc. Or consider a service that many threads are clients of, and must keep track of some data for each `Thread`. Since the heap is shared between all threads, it is

normally impossible to have per-thread objects (or objects with per-thread field values).

The general use case is when your code uses a framework with callbacks. The framework may receive a web request and call one of your code's methods, which in turn may need to call back a framework method.

One technique for this situation is to have the framework pass a context object to your code as a parameter. In turn, your code will pass that object when doing call-backs. While this works and is efficient, you need to pass that parameter (and possibly several such parameters) to every method that invokes a call-back. And also to methods that invoke methods that invoke call-backs, So basically, to all methods.

A server may have one `Thread` per conversation, with data such as transaction status and/or security (user logged in status). This is often referred to as thread ***context data***. But such data is generally part of an object and thus shared between threads. Java allows you to set thread-specific values from some fields using `ThreadLocal`. Another common use of `ThreadLocal` is for logging, where each thread maintains context data that the logger needs to display, and also each thread may have separate output buffers (else output may get mixed up).

An example use of `ThreadLocal` is in JSF and `FacesContext.getInstance()`. Compare this to many other frameworks which forces framework users to pass `HttpRequest` et. al. around, polluting method signatures just because somewhere deep inside there might be some `HttpRequest` utility method that needs to be called. (This is just an example of context data.)

This is a common idiom with frameworks and similar code. Values are created in a top-level method (e.g., main), and only used in the bottom-level methods. So you either make such variables public fields, or you must pass the values via method parameters through all intermediate methods (which don't use the value). `ThreadLocal` was an attempt to address this.

Why not use ThreadLocals? `ThreadLocal` (available since Java 1.2 aka Java 2) is generally only appropriate for highly optimized multithreaded applications such as application servers. It allows non-shared heap data access (within a single-thread of execution) without synchronization (*thread containment*). The potential performance gains are small with newer JVMs.

Like any object, you can declare the `ThreadLocal` reference as `public`, `protected`, or `private`, to limit the scope. But the main use usually means access from multiple classes. Thus `ThreadLocal`s are in some ways `Thread` *global* variables, with all the problems that globals cause. It may be better to pass such context data as method parameters despite the hard-to-read code. Or use

some framework that handles security and transaction details for you. (These likely use `ThreadLocal`s internally but correctly.)

ThreadLocal variables are inherited by child threads (and thus are not actually local to a single thread!) and this is rather dangerous and slow.

Using a local variable in `run` is private to a thread (as it's on the stack), but local variables can't be seen from other methods. One way is to extend class `Thread` and add constructors and instance variables. But extending class `Thread` is not always the best choice. (In fact, it almost never is the best choice!)

Another way is to use a global array of objects (or database or file), say some `static` array, with some careful naming conventions so each thread knows which object in the array belongs to it. This is fragile, hard to extend or enforce, and easy to mess up! It is only slightly better to use a global `Map` of `Thread`s to `Object`s. Both cases require using synchronization, vastly slowing down access to the data.

> **The best solution is to avoid having per-thread data in the first place.** After that, consider passing the data around with parameters. Or using a framework that handles the per-thread data for you. The last resort is using `ThreadLocal` directly.
>
> As of Java 25, an alternative to ThreadLocal is available, *scoped values*. That is discussed after ThreadLocal, below.

Using ThreadLocal

The `ThreadLocal` class is a wrapper class for objects that need to be thread specific. Each object has a `set` and a `get` method you can use to set and get the current Thread's value. These are usually called from static public methods. For example:

```
private static ThreadLocal userId = new ThreadLocal();
userId.set( new String("Hymie") );
...
String name = (String) userID.get();
```

There is also an `InheritableThreadLocal` class so child Threads can access the values.

If you do use ThreadLocals directly be careful to clean them up when done, as in most cases they will <u>not</u> be garbage collected until the last thread referencing that ThreadLocal is.

Another place where this bites people: Hot-deploy application servers. For example with Jakarta Commons Logging, passing *`ClassName`*`.class` to the

constructor of the logger (a fairly common usage pattern) causes the class to be saved as a `ThreadLocal` object. When you hot deploy a *war* file that replaces the old war file the classes never get reclaimed and unloaded, resulting in a memory leak. Of course hot deploy shouldn't be used in production. You should always use a `LogFactory.release(Thread.currentThread().getContextClassLoader());` in a `ServletContextListener`. It catches the cases where the application forgot to cleanup.

An alternative is to use `MessAdmin`, which will give you this behavior (and lots of others!) for free.

ScopedValues [JEP 506]

> [*Quoting from InfoWorld article, 3/6/2024*]
>
> "`ScopedValues` is a new way [a Java preview feature since Java 21, added officially in Java 25] to achieve `ThreadLocal`-like behavior. Both elements address the need to create data that is safely shared within a single thread but `ScopedValue` aims for greater simplicity. It is designed to work in tandem with `VirtualThreads` and the new `StructuredTaskScope`, which together simplify threading and make it more powerful. As these new features come into regular use, the scoped values feature is intended to address the increased need for managing data-sharing within threads.

`ScopedValue` variables are typically `static` and `final`. You can initialize it from within a method (one called by some framework). Once initialized, you can use it like so:

```
class MyClass {
  private static final ScopedValue<MyContext> CONTEXT
      = ScopedValue.newInstance();
  void serve(Request req, Response resp) {
      var sessonData = createContext(request);
      ScopedValue.where(CONTEXT, sessionData).run(
          () -> Application.handle(req, resp)
      );
  }
  public Data readData(String key) {
     var context = CONTEXT.get();
     var db = getDBConnection(context);
     return db.readData(key);
} }
```

The `readData` method in this example is invoked by `Application.handle` and can access CONTEXT even though it was never passed as a parameter.

Notice the `where` method. It creates a scope from which you can run code. Once `where().run()` returns, the CONTEXT object is cleaned up and ca no longer be used.

`ScopedValue` objects thus create a limited form of *dynamic scope* in Java. Code using them is likely easier to read (and debug) than code using `ThreadLocal` objects.

> To use a ScopedValue object method, the JRE must search backwards on the thread's call stack to find it. To make subsequent references fast, the value is cached (in a ThreadLocal!). The size of this cache is at most 16 entries. So if you use more than 16 ScopedValue objects in a cyclical manner (in one thread), you will have mostly cache misses and poor performance.

ThreadGroups and Executor

`ThreadGroups` were used so the threads of different Applets don't interfere with each other; a separate `ThreadGroup` was generally created for each Applet. You could create new `ThreadGroups` and control which threads belong to which group, and ThreadGroups could be nested. **However, `ThreadGroups` were deprecated (so were Applets!), and then removed completely in Java 14.**

The **`java.lang.concurrent` package (since Java 5) provides a much better way to manage multiple threads in most situations.** Instead of creating (and managing) your own thread objects, you create an `Executor`. This has convenient methods to create new threads. For example, instead of:

```
new Thread( runable1 ).start();
new Thread( runable2 ).start(); ...
```

and them managing `ThreadGroups`, you can use:

```
Executor e = ...;
e.execute( runable1 );
e.execute( runable2 ); ...
```

An `ExecutorService` is similar but has a `submit` method, which is passed either a `Callable` or a `Runnable`. A **`Callable`** has a `call` method, not a `run` method. The advantage is the `call` method can return a value when the thread terminates. When a `Callable` is submitted to an executor service, a `Future` object is created and returned. This object has methods you can use to determine the state of the created `thread`, a `cancel` method to (attempt to)

terminate the thread (safely), and another method to collect the return value, if any.) With the old way, you needed to have a shared and synchronized heap object to have a thread communicate a return a value.

(Maybe you've noticed by now, but pre-Java 5 only had very low-level and difficult to use thread support. Java has had incremental improvements since then, until Java 21 and virtual threads. Even better APIs are in the works (as of 2024)!)

Thread Pooling (Since Java 5)

A single-threaded server is slow and difficult to write. A multi-process server (which creates or *spawns* a process for each session) is better but has a lot of overhead, since a process must be created for each conversation. A multi-threaded server is better still, allowing the threads to share data (not so easy for a multi-process server). (High performance servers can be built with a single thread, but generally that requires platform-specific code and is very difficult to write or read.)

But there can be unacceptable overhead on some platforms to create (I like *spawn*) threads. One common real-world solution is known as ***pooling***. In this scheme you create threads in advance and start them; they initialize themselves and then wait until needed.

When the server needs a thread, it just notifies one that's in the pool and it's ready to go. Thus you only need the overhead of starting a new thread when the pool is empty.

When a `Thread` is done with its work, instead of terminating it: cleans itself up (back to the initial state), puts itself back into the pool, and suspends itself via `wait`.

> Be careful about using thread-specific data when using `Thread` pools, it is easy to forget to reset these resources.

The **`java.util.concurrent`** package has methods to make managing thread pools easier than before. (See the `Executors` class.)

Suppose you have a set of data elements (or *jobs*), and you need to perform some kind of processing over each one of them. You want to maximize the speed at which this processing is done, but, on the other hand you don't want to hog every system resource available if the system is being used. A good strategy would be to have a thread pool with a predefined number of maximum active threads, which will process the data one item at a time as soon as the threads become available. This strategy can be quickly implemented using a fixed thread pool executor service. Here's an example network server:

```
class NetworkService {
  private final ServerSocket serverSocket;
  private final ExecutorService pool;
```

```
  public NetworkService(int port, int poolSize)
                                throws IOException {
    serverSocket = new ServerSocket(port);
    pool = Executors.newFixedThreadPool(poolSize);
  }
  public void serve () {
   try {
    for (;;)
      pool.execute(new Handler(serverSocket.accept()));
      logger.info(String.format(...));
   } catch (IOException ex) { pool.shutdown(); }
  }
  public shutdownPool () {
    logger.info( "Shutting down server..." );
    executor.shutdown();
    while ( ! executor.isTerminated() ) {
      try {
        executor.awaitTermination( 1000,
            TimeUnit.MILLISECONDS );
      } catch ( InterruptedException e ) {
        ... // cancel shutdown
      }
    }
    logger.info( "Server shutdown complete." );
  }
}
class Handler implements Runnable {
   private final Socket socket;
   Handler(Socket socket) { this.socket = socket; }
   public void run() {
     // read and service request
}  }
```

Fork-Join

Fork-join is another concurrency framework, added in Java 7. It was inspired by MapReduce (see Map-Reduce above) and implemented using the executor framework. (Streams, introduced in Java 8 and discussed previously, are implemented using fork-join.) Fork-join is for workloads that benefit from a *divide-and-conquer* approach. It works well for recursive tasks that use threads per task. (It's not so good for other types of work, so the executor framework is still very much used.)

Here is a pseudocode example ~~stolen~~ copied from Oracle's article on fork-join:

The first step for using the fork/join framework is to write code that performs a segment of the work. Your code should look similar to the following pseudocode:

```
if (my portion of the work is small enough)
  do the work directly
else
  split my work into two pieces
  invoke the two pieces and wait for the results
```

Wrap this code in a `ForkJoinTask` subclass, typically using one of its more specialized types, either `RecursiveTask` (which can return a result) or `RecursiveAction`.

After your `ForkJoinTask` subclass is ready, create the object that represents all the work to be done and pass it to the `invoke()` method of a `ForkJoinPool` instance.

Today, fork-join is rarely used. Java 8 parallel streams is much easier to use, and works in almost all cases where fork-join would be useful.

Virtual Threads

Java 21 adds virtual threads! These are very fast to create and to destroy, and are not limited by the OS's threads; your program can have millions of these! With virtual threads, there is rarely a reason to use pooling (and several good reasons not to do so).

However, in some rare cases virtual threads can result in worse performance compared with executor pools. When used with a thread feature known as *pinning*, liveness issues (starvation, deadlock) may become possible; there are some reports on the Internet claiming such issues were encountered (2024).

Even newer stuff is coming! A new API called structured concurrency promises fewer errors, simpler coding, and other good stuff.

Testing Concurrent Programs

The value of encapsulation is that it makes it possible to analyze the behavior of a portion of a program without having to review the code for the entire program.

Similarly, by encapsulating concurrent interactions in a few places, such as workflow managers, resource pools, work queues, and other concurrent objects, it becomes simpler to analyze and test concurrent programs. Once the concurrent interactions are encapsulated, you can focus the majority of your testing efforts primarily on the concurrency mechanisms themselves.

Concurrency mechanisms, such as shared work queues, often act as conduits for moving objects from one thread to another. These mechanisms contain sufficient synchronization to protect the integrity of their internal data structures, but the

objects being passed in and out belong to the application, not the work queue, and the application is responsible for the thread-safety of these objects. You should make these domain objects thread-safe (making them immutable is often the easiest and most reliable way to do so).

Testing code that uses low-level concurrency features for liveness and thread safety issues is very hard, since such issues are often not easily reproduced.

Summary:

Threads are terminated by an uncaught exception, which prints a stack trace by default but you can change that by `setUncaughtExceptionHandler`.

Rather than create threads to use as timers, Java provides a simple swing timer, as well as a powerful general-purpose timer in java.util. Beware of clock resolution, which can be unpredictable on some platforms.

All threads have a priority but there is no guarantee that changing priorities will have any effect. If there is an effect, you can't tell what it will be.

Threads can have a name set on them; the default is thread-n (n is a number).

Threads can use ThreadLocal to contain objects to a single thread.

A common technique to improve performance is known as thread pooling.

You can manage pools of threads easily using the Java 5 Executor classes instead of the old (and now removed) ThreadGroups. In some cases the Java 7 fork-join framework (inspired from map-reduce) can be used. Java 8 Streams (aka aggregate operations) are your best choice.

Use tried-and-true code such as the Java 5 concurrency additions including atomics, executor framework, locks and conditions. Try using Java 21 virtual threads. Avoid "rolling your own" stuff using older Java mechanisms whenever possible.

Look for more concurrency improvements in future Java versions, for example scoped values (still in preview in Java 23).

Multi-thread Summary

- A *thread* is a sequence of steps executed one at a time. A *multithreaded* program has several (up to thousands) of such threads all running at the same time. Each thread maintains its own state: a stack for local variables, thrown exceptions, priority, status (running, sleeping, blocked, etc.), and other information.

1. Normal Java threads are "out-sourced" to the underlying operating system and simply use whatever threads it provides. Java 21 introduced *virtual threads*, which only indirectly use native threads; they are very fast and lightweight, and they enable advanced concurrency mechanisms and coding idioms.

- A multithreaded program may run much more efficiently (especially on multi-core servers) than a single threaded one. Threads are useful for background tasks such as slow loading media (assuming you don't need the media right away), increasing responsiveness of GUIs (by having short event handlers that *spawn* threads to do the hard work, then return quickly), interactive systems, for servers, and many other uses.
- Threads share access to the heap and thus to all objects. Each thread gets its own stack, so a method's local variables are never shared.
- Multithreaded (concurrent) code can have issues of thread safety (corruption). The solutions to that can cause other issues: visibility, liveness, and poor performance. Using low-level concurrency mechanisms (pre Java 5) is error-prone and requires a deep understanding of the topic. (I recommend the book Java Concurrency in Practice. It covers the concepts and mechanisms through Java 5, but it's still considered the best book for now.)

  Using some newer Java concurrency features can help mitigate these issues. Always use them, and/or use standard idioms that are known to mitigate liveness and performance issues.
- Document the thread-safety of your classes' methods, so users can use your classes safely (and will know when to synchronize their use). It is recommended to use various annotations for this.
- A multithreaded program doesn't terminate until the last non-*daemon* (or *user*) Thread does. Use daemon threads for background tasks (animations, tickers, music, etc.). Such threads typically are set with a lower priority as well. Use `setDaemon(true)` to set a user thread to be a daemon thread. It is not legal to try to turn a daemon thread into a user thread.
- A Java thread can be in one of six states: *new, runnable, blocked, waiting, timed waiting*, or *terminated*.

- Having and using multiple threads is easy unless the threads must share access to mutable (writable) resources such as properties (*fields*) of classes. Unfortunately this is a most common situation. If access isn't controlled (*synchronized*) than a *race condition* or *race hazard* exists (a thread safety issue). These are tough to troubleshoot, as you may not discover the problem during testing.
- For each shared mutable resource, the code that modifies and/or reads the resource is called a *critical section* (or *critical region*), which must behave *atomically*. This means all changes to the resource must appear to happen all at once to the other threads using it. Each critical section should be inside a *synchronized block*. In some cases it is enough to declare the variable as `volatile`. In most cases that isn't sufficient and atomic data types should be used. Entry into a synchronized block requires that the thread *acquires* a *lock*, which is automatically *released* when the block exits.
- Not every object needs its own lock (e.g., Bank.java). The rule is that every set of resources that are updated or read together (atomically) requires a unique lock to protect its critical sections. Often, shared resources can be used in multiple critical regions, so a single lock for a set of resources won't work. Instead, each resource has its own lock and a thread must acquire them all before proceeding. (E.g., To transfer money between two accounts, both accounts must be locked while the transfer is in progress. but it wouldn't make sense to lock all accounts with a single lock, as many operations only access a single account.) Having multiple locks required for some operations to proceed safely can cause liveness and performance issues.
- Since one thread has to wait (or *block*) to acquire a lock another thread already has, there might be a considerable delay before the second thread runs. Keep critical sections as short and fast as possible.
- When a multithreaded program uses more than one lock, *deadlock* (a most serious liveness issue) is possible. Use the *resource ordering* technique to prevent potential deadlocks. Other liveness issues include *livelock* and unacceptable performance. There are mechanisms and code idioms to mitigate such issues.
- Threads have priorities in the range `MIN_PRIORITY <= NORM_PRIORITY <= MAX_PRIORITY`, but there are no guarantees about how the Java thread scheduler will work. Scheduling threads depends on the underlying operating system, so the piority may have no effect, or different effects on different systems. A system's scheduling policies can cause *thread starvation* or *priority aging*. It is usually best to not change thread priorities, except to set the priority of threads started from the AWT event handling thread back to `NORM_PRIORITY`.

- A thread can (and should) use `Thread.yield` to tell the system it is OK to let another thread run now, that shares resources.
- A thread can use `Thread.sleep` to *suspend* itself for a while, but there is no guarantee that the thread won't wake up early, or that it will run as soon as it does wake up. You can specify how long to sleep in milliseconds (an overloaded version allows nanoseconds as well). A sleeping thread still has whatever locks it previously acquired. The time specified is approximate only; it depends on the platform's clock's resolution which can change at any time.
- Threads can communicate by using a condition variable's `wait`, `notify`, and `notifyAll`. (These methods are invoked on the Java object serving as the lock, so the code can be confusing to read.) `wait` causes the thread to become suspended and releases all its locks (these steps happen atomically). Once the thread awakens, it must first re-acquire its locks before proceeding. A different thread uses `notifyAll` to waken all suspended threads; they usually check if it is okay to proceed and all but one usually go back to waiting. An optimization is to awaken a single suspended thread using `notify`.
- The standard idiom for using `wait` and `notifyAll` is:

```
void aMethod ()
{  synchronized ( this )
   {  while ( ! condition )
         wait();
      doSomeWork();
   }
}
```

  It is important to use a `synchronized` block, and to use a loop (and not an *if* statement) to see if it is ok to do some work. Typically some other thread will create some work (such as a printjob), and use `notifyAll` to alert the server Thread there is some work to do.
- One thread can wait for another to terminate before proceeding by using `Thread.join`. An overloaded version of this method takes timeout values similar to `sleep`.
- `wait`, `notify`, `notifyAll`, and `join` are primitive operations but there are standard design patterns that use them such as *producer-consumer*, and *readers-writers*.
- Since Java 5, atomic variables, new easier-to-use locks, and other features are available and should be preferred. The newer lock types can include multiple, named condition variables (and use different methods from `wait`, `notifyAll`, and `notify`: `await`, `signalAll`, and `signal`). The use of

newer locks can greatly simplify your code. For example, there is a built-in reader/writer lock.

- Other newer Java mechanisms such as parallel streams can avoid the need for any multithreading code, in many cases, but still provides the benefits of concurrent and parallel threading.

  As of Java 21, virtual threads are available. An even newer API, *structured concurrency*, is being prepared (as of 2024) and promises to be even easier to use.

- It is generally not safe to stop (*cancel*) or suspend a thread from another thread, as a thread might be in the middle of a critical section. Instead, set flags that a thread will check periodically (when it is safe to stop/suspend), and/or *interrupt* the thread. There are coding idioms you should use in such cases, however newer Java concurrency features may have better ways to achieve your goals.

- `someThread.interrupt` causes *someThread* to set a flag which indicates the Thread has been interrupted. This causes waiting and sleeping threads to wake up with an `InterruptedException`, but doesn't otherwise stop a thread from whatever it was doing. A thread can use `isInterrupted` to see if it has been interrupted, or `interrupted` to check and also reset the flag. Using `interrupt` and `interrupted` is often easier than using `wait` and `notifyAll`.

- Using the `ThreadLocal` class it is possible to have non-shared or *thread-specific* properties of a class. Normally that isn't possible since all threads share access to all objects. `ThreadLocal` is not always a good idea; try to avoid its use.

- *Thread pooling* is a technique wherein threads are created (*spawned*) in advance, suspended, and put in a list. This allows for a more responsive server (since thread creation is the most time-consuming part).

- Collections of threads (a *thread pool*) can be managed by using `Executors`. An executor handles all the low-level details of thread management for you. Such threads can be inspected and even stopped, and return values to the calling thread, by using Futures.

  Other such frameworks have been added to Java over the years, such as the fork-join framework in Java 7 and parallel streams (part of what's called *aggregate operations*) in Java 8. Generally, use parallel streams in preference to the fork-join framework directly.

- Overriding the `Thread.uncaughtException` method is the only way an exception in one thread can communicate this fact to another. By default, uncaught exceptions in a thread terminate the thread and then vanish.
- `java.util.Timer` and `java.util.TimerTask` were added in Java 1.3, as was the `jakarta.swing.Timer` class. Use a `Timer` object to schedule tasks—`TimerTask` subclass objects—for execution. Timer uses a thread internally. To create a Timer object, call either the `Timer()` or `Timer(boolean isDaemon)` constructor. Then call one of the `Timer`'s schedule methods, passing a `TimerTask` and an indication of when and how often to run the `TimerTask`.

  A single `Timer` can handle the scheduling of many `TimerTasks`, but all `TimerTasks` share a single thread so don't let one task hog the `Timer`. You must extend class `TimerTask` and override its `run` method. The swing Timer is easier to use to update swing GUIs. Be careful of the name clash if importing both `java.util.*` and `jakarta.swing.*`!

Lecture 17 — Service Management & Monitoring, and Incident Response

Management Tools: JMX and MBeans

Management of a large application can be a problem. With an EE application running on a cluster, it can be very hard to see what parts (classes) of the application are heavily used, which versions are deployed, and to turn on/off parts of the application functionality. If some part of an application running on a cluster fails, or is just over-burdened and slow, how will you know? Neither logging nor testing can help you here.

Code that supports management and monitoring tools (especially those that measure performance) is often referred to as ***instrumented***. Managed applications can be updated, reconfigured, and have parts of the application enabled (deployed) and disabled (undeployed). Managed applications provide statistics about application performance (in whole and for various parts, often timing data for every method entry/exit).

This data is used by managers to decide on configurations and hardware upgrades, and where to spend development efforts for code improvement. At the very least, for a web-based application, you should monitor the delay (latency) and response times of all user requests, and also collect supporting metrics: number of requests per second/minute/hour/day, number of DB transactions and response time of those.

Managed applications can tell a system administrator that something bad has happened (rather than wait for customer complaints).

Every (or at least some) feature of an enterprise application should have a "switch" to enable/disable just that feature. For example, if there's a database problem, you should be able to click on a switch in a GUI to make your application operate in read-only mode, until the problem is fixed. If response time is too slow, you can click a switch to use a simpler user interface, or perhaps turn off ad injections, until response times improve. When deploying a new feature, if it causes problems it should be easy to disable just that new feature.

ITSM — Information Technology Service Management

Applications, especially enterprise ones, can be thought of as providing services to customers; either internal customers (for example a messaging service or a reporting service) or external customers. *Service management* is concerned with more than just delivering services. Each service, process or infrastructure component has a lifecycle, and service management considers the entire lifecycle from strategy through design and transition to operation and continual improvement, all the way to decommissioning.

A (computer) *Systems Analyst* is a person who specializes in designing, building, monitoring the IT infrastructure of an organization, to make sure it meets the organization's needs. Such a person must know service management such as ITIL, but not necessarily detailed knowledge of software design.

ITIL is a public framework that describes best practice in IT service management. It provides a framework for the *governance* of IT, and the management and control of IT services. Such frameworks (there are several) provide broad guidance on the business side of IT. As a developer, you must be aware of the overall goals of your employer; those will be business goals, not technical ones. You need to understand how your work affects the business. Career growth depends not only on keeping up with technical skills, but also working on these management aspects.

Increasingly, the old model of developers delivering source code to operations (system administrators) for deployment and monitoring, is giving way to a new model called ***DevOps***. In a DevOps world, developers work with system admins and share responsibilities. Thus, you will need to understand how to deploy your services and applications, how to monitor them, upgrade them, and so on. (In some larger companies such as Google, all developers take turns doing system administration, and vice-versa. Google won't even hire developers who cannot do service management nor system administrators (which they call *system reliability engineers*) who cannot develop code.

Understanding the operations part of the IT lifecycle helps you to understand why you need to develop code that can be controlled and monitored, including performance metrics.

JMX (Java Management Extensions) — Monitoring Application Metrics

JMX is a technology that provides a simple, standard way of managing resources such as applications, devices, and services. It can be used to **monitor and manage** resources as they are created, installed and implemented, and to monitor and manage the Java Virtual Machine.

Using JMX technology, a given resource is *instrumented* by one or more Java objects known as *Managed Beans* or ***MBeans***. These MBeans are registered in an *MBean server* that acts as a management agent and can run on most devices enabled for the Java programming language.

An MBean is a Java object that represents resources to be managed. It has a management interface consisting of named attributes that can be read and written, named operations that can be invoked, and notifications that can be emitted by the MBean. Note the JVM (and possibly the OS itself) can be managed the same way, only such **system resource management beans are called *mxbeans***.

A JBoss (and other) Jakarta EE servers have a management console that uses JMX to discover the mbeans and mxbeans. It provides an interface where you can get information (statistics) about them, deploy/undeploy them, and view notifications. This sort of technology is very valuable to manage Jakarta EE applications, especially in a cluster. See the Javadoc guide for *management*, the tools docs for **`jconsole`** and the newer **`visualvm`**, the package API docs (`jakarta.management.*`). (Compile using "`-g`" option to include debugging info.)

The metrics library most commonly used for Java is DropWizard.metrics. Some examples (From metrics.dropwizard.io):

A *meter* measures the rate of events over time (e.g., "requests per second"). In addition to the mean rate, meters often also track 1-, 5-, and 15-minute moving averages. Here's some sample code:

```
private final MetricRegistry metrics
   = new MetricRegistry();
private final Meter requests =
metrics.meter("requests");

public void handleRequest(Request request,
      Response response) {
   requests.mark();
   // etc
}
```

This meter will measure the rate of requests in requests per second.

A *gauge* shows an instantaneous measurement of a value. For example, we may want to measure the number of pending jobs in a queue:

```
private final MetricRegistry metrics
      = new MetricRegistry();
   metrics.register(..., new Gauge<Integer>() {
      @Override
      public Integer getValue() {
         return queue.size();
      }
   });
```

In the metrics display (or timestamped in a metrics event log), this will show the current number of things in a queue.

Other metrics are instrumented similarly. You can *count* things (user logins, requests for a service, web page hits, etc.) over some interval of time. You can use *timers* to measure how long something took (time to process some request, time to fetch data from storage, etc.) (TODO: *Add DropWizard.metrics demo.*)

Adding metric collection has an overhead, so don't collect metrics that don't help you. **Collect *key performance indicators* (KPIs)**: all errors, basic metrics (CPU and number of threads, memory use and paging/swapping, network throughput and latency, disk I/O rates and space available), and application availability (uptime).

Then collect important business metrics needed for management reports: queries/requests per second, response time, number of unique visitors, etc. These depend on the type of application. (Some metrics can be calculated from the time stamps of log messages. Direct collection is much simpler and more common.)

If you need more insight as to why your applications are not performing as expected, also collect metrics on all external dependencies (e.g., count and time DB queries made by your program to external DB), cache hit/miss ratios, and major garbage collection counts and durations. Note some of this information can be obtained via log statements instead.

Your instrumented code is collecting metrics; now what? Those values are typically stored for later historical analysis, and also displayed in ***dashboards***. Dashboards show various moving graphics and status information.

Examining Metrics and Profiling Data in Dashboards

The JDK includes some simple (and one advanced) dashboards: `jcmd`, `jconsole`, **`visualvm`** (no longer bundled with JDK), and `jfr`/`jmc` (now open source and bundled with the JDK). To see these, you need to launch some program that collects data and reports it via JMX.

Profiling

Determining if your code is not performant (missing your performance goals) and where in your code the problem lies requires *profiling*. This shows various things about your code at a method or even statement level. Profilers work by either *instrumenting* the code (inserting measurement code) or by *sampling*. Instrumenting can be done by adding metric collection code in strategic locations (such as the entry and exit of each method) or using log statements there. Instrumenting alters the code and slows performance. Sampling profilers don't need any code modifications and have a low overhead when used.

With JVM languages, sampling profilers collect data from the JRE at a rate you control, usually about 10 times per second. The data collected includes memory use, disk and network I/O, time spent in each method, and thread lifetime events (creation and deletion). The profiler does this by snapshots of the stack of each thread, the heap, and various performance counters

maintained by the JRE. Some profilers have additional capabilities such as collecting data about database queries or web requests.

Sampling data can be collected for a specific duration and saved to a file (using Java Flight Recorder). Continuous sampling is also possible.

Finding the method that's hogging 90% of available memory, or taking 90% of the time used by your application, tells you where the performance problem is. (Fixing that is up to you!)

A simple demo: start any long-running Java program:

```
java -jar "%JAVA_HOME%/demo/jfc/Java2D/Java2Demo.jar"
```

Then use task manager (or `ps` on Linux/Mac) to determine the process ID (*pid*) of that running JVM. Then from another command line prompt, run:

`jconsole` *`pid`* *The PID is optional; if omitted, jconsole allows you to pick.*

Also try:

visualvm (Now an open source project.) Show some features, e.g., a heap dump, then show String objects. Or show profiling by CPU time or memory use. Can add plug-ins, too; then show Visual GC.) See also Oracle's Java Mission Control and Oracle's Advanced Management Console. Demo `jmc`.

Profiling tools to know (from bell-sw.com):

- JDK Flight Recorder or Java Flight Recorder (JFR) is a low-overhead profiler built into OpenJDK, which collects data about various JVM events. JFR can record hundreds of event types, plus you can create custom events to suit your needs. JFR recordings are binary log files, which can be analyzed in JDK Mission Control, a set of monitoring and profiling tools with a GUI.

  JDK Mission Control (JMC) is also free and open-source, but it is not included in the OpenJDK distributions, and has to be downloaded separately from vendors that provide it.
- VisualVM
- Async Profiler
- New Relic is not only a profiler, but an observability platform that offers a wide variety of features. Profiling is only part of the pack that also includes anomalies alerts, infrastructure monitoring, security testing, and more. The available services depend on the subscription plan, so this is not free.
- Digma is an IntelliJ plugin that is free for developers. It apparently uses AI to analyze the data it collects, to indicate code problems. But you must use IntelliJ.

- JProfiler is a commercial Java profiler with many features not found in the free profilers.
- YourKit is similar to JProfiler. I think it is slightly more popular (as of 2024).

(Note! Logging, profiling, and management tools **may not work** if your code isn't in a package! MBeans have trouble with classes in the default, nameless package, and most profilers use that.)

These and other tools show you what is happening with your application, on a basic level. To see more interesting metrics, you may need to instrument your code.

> Note that dashboards serve no purpose if nobody is looking at them. Since no one can stare at a dashboard all day, the data is usually collected and saved to be viewed on-demand. Then some ***alerting software*** is used to notify operations that something interesting is happening.

Managing an application generally implies there is some way to configure the service. It is unacceptable to have to re-code and redeploy software for this. Generally, applications obtain configuration settings from a file with overrides available on the command line. However, such an approach doesn't help to dynamically reconfigure services when needed without stopping and restarting them. Another approach can be used: have applications able to be signaled from management consoles that they need to update their configuration. For clusters of servers, you can even deploy configuration servers; these are useful for large clusters.

A typical use of dynamic configuration is when the monitoring system alerts you to a dangerously high spike in user traffic to your service; unless you reduce the response time of requests you are in danger of dropping valuable customer requests or even crashing your service. In such cases, you can turn off selected parts of the service: using a simpler user interface (say by turning off ads or data collection), use cached data instead of live data, or even turn the service into a read-only state. Likewise, if bugs are discovered in some new feature that was recently deployed, you can update the configuration to turn off that one feature.

The code for this is generally simple (and known as *feature flags*): Have `if` statements at the top of methods that test the current value of some configuration flag, and either do the normal action or some simpler one (maybe return right after logging). The tricky part is getting the current value of those flags, but there are many well-known solutions.

> AT&T, a.k.a. *the phone company*, monitored their network 24/7. If long distance calls were overwhelming the lines, the managers would turn on a throttle: 1 out of every X called would not even be attempted and a "fast

busy" signal (which sounded slightly different than the normal slow busy signal) would be immediately sent to the caller. If you knew about that (most folk didn't), you could immediately retry your call and likely get through.

Some busy websites do this sort of throttling today.

Monitoring Deployed Services

Effective service operation depends upon knowing the status of the infrastructure and the services' components. ***Event management*** does exactly this. There are three types of event: informational, warning and exception. An **event** may indicate that something is functioning incorrectly or is in danger of doing so shortly unless action is taken, potentially leading to an incident being raised. Events may also indicate that a normal activity, or routine intervention, has occurred or is necessary.

Events are generated either by the software service or by monitoring software that polls a service every so often. All events are generally saved in a database and/or a log file for a period of time (usually hours, days, or months).

You should use continuous monitoring of your services and applications. Some items that are very commonly monitored include if the service is up (running) or down, the percentage of instances of the service in a cluster (for example, a cluster of web servers) that are working, the response time of requests, and number of requests (or queries) per second. If using a web service front end (common for Jakarta EE applications), you may need information about the customer flow (which views (web pages) do the see, in what order, how long do they spend viewing each, the starting view and the last view before they left using your service) through your application.

Aside from these important business metrics, you need to measure CPU use, network use, memory use, and storage use. Such information can show you memory leak issues, insufficient resources allocated, or that it is time to perform some routine cleanup to improve performance.

All that raw data is stored, condensed, and analyzed by monitoring software. Such monitoring systems include log files (or databases holding that event info) and *dashboards* to visualize the important information in various graphs and logs. The software also tries to detect potential incidents and alert you to the issue.

Incident Response

An ***incident*** is an unplanned interruption to an IT service or a reduction in the quality of an IT service. If a monitoring system shows a potential incident, the first step is to confirm the problem. This is usually done by examining the log and/or performance data on the affected service carefully. Then you must decide on what action, if any, to take immediately and/or later. You need to prepare in advance for

such situations. These plans are known as incident response or (ITIL) incident management.

The urgency of confirmed incident runs from logging the incident and ignoring it for now, to opening up a trouble ticket for the issue, to emergency *all hands on deck* issues. (That last one is common but a bad way to handle serious security incidents.) Note that some incident responses may involve risk and are thus a business decision to make, not a technical decision. On the other hand, some incident responses are mandated by law, industry regulation, or best practices.

If an incident cannot be resolved quickly, it is escalated. *Functional escalation* passes the incident to a technical support team with appropriate skills; *hierarchical escalation* engages appropriate levels of management. Security-related incidents are usually escalated both ways.

The basic steps for incident response depend on who you ask. It also depends if you are considering a security attack, a disaster, or simply a bug. Wikipedia defines the ITIL steps as:

- Incident detection and recording
- Classification and initial support
- Investigation and analysis (also known as *root cause analysis*)
- Resolution and record
- Incident closure
- Incident ownership, monitoring, tracking and communication
- Establish incident framework management
- Evaluation of incident framework management

There are many templates for how to perform incident response, such as:

- Identify/confirm an incident has occurred.
- Ensure the safety of the users.
- Ensure the safety of the users' data.
- Open a ticket about the issue, unless there is one opened earlier.
- Perform initial incident classification and decide on urgency and what immediate actions may be required.
- Restore service as quickly as possible. (Depending on the nature of the incident, that might be a simple quick action. For example, if a memory leak crashed software, a restart will temporarily work. If there was a security breach, a fix may involve preventative steps before restoring from backups, and thus be very time consuming.)
- Notify legally required entities (and others as appropriate).
- Document details, checking for completeness of information gathered on the incident (who, what, when, and where).
- Explore causes of the incident.

- Make necessary additions or corrections to your software and/or to your procedures. Test then deploy updated software and procedures.
- Follow-up and completion: monitor restored services closely, complete reports, and close the ticket.

Once the root cause of some incident is known, unless it is fixed immediately it should be entered in the "**known errors database**" (KEDB), also called the ***known errors log***. You list the issue and any work-arounds for it here, along with the date+time the issued was present (if known) or the date+time it was detected. References to any related issues in the issue tracker are included. Once the problem is fixed, the entry can be deleted, but ideally, just the date it was fixed (and the fixed version numbers of effected software) are included.

The known error log is often overlooked in the US but is mandated by the ITIL and UK law. It should be created when the software or service goes live (when it is first released).

Lecture X — Jakarta EE (J2EE) Overview (*The remaining notes are very out-of-date)*

Most real-world applications need access to central resources, such as up-to-date inventory, available seats on a plane, highest bid to date on an auction item, etc. Often these resources are stored on a database, but other resources are accessed centrally too: current stock prices, credit-card processing, inventory adjustment, etc. This implies the client is remote from these resources, and must use a network connection to access them.

Instead of having smart clients installed remotely (so-called ***thick clients***), ***application servers*** install the client code centrally, along with the other resources. Users can then use a much simpler client to connect to the application server. Often just a web browser is used, but other so-called ***thin clients*** can be used as well.

Such networked applications are known as ***enterprise applications***, or ***web applications*** when the client is just a web browser. These often have different requirements (scalability, use in a cluster, reliability, etc.) than simple, stand-alone applications. While you could code up enterprise applications in standard Java, it would be like coding your own version of the Oracle relational database for each application. It is much easier to let some vendor code up all that infrastructure, called a ***framework*** or sometimes ***middleware***. Then you just use the framework for DB access, network (including HTML) access, transactions, messaging, etc.

To avoid having each vendor have incompatible frameworks, **the Jakarta EE specification defines the interface for these features**. To use Jakarta EE, you need some vendor's server that implements some or all of these interfaces. For example, you use the standard Jakarta EE API to send email, but you still need some vendor's server that implements that API. Note you can change vendors without changing your code, since the API is standard.

> The Jakarta EE server is called a ***container***, probably because it contains Jakarta EE applications. A ***servlet container*** (or *web container*) such as Tomcat only implements some of the Jakarta EE specification. An ***EJB container*** such as Glassfish implements the complete Jakarta EE specification.

The server provides many services to your Jakarta EE applications, and can even automatically inject code into the server-managed components of your application, which can simplify your code by eliminating a lot of "boilerplate" code. (However, the hidden nature of the code injected can lead to confusion when reading and debugging your application code.)

You simply code up your applications normally, making sure the appropriate Jar file(s) from the vendor are on your `CLASSPATH`. Some servers are free, such as

Tomcat (which only implements a subset of the Jakarta EE standard) and Glassfish. The tricky part is adding your code to the vendor's server, known as *deployment*. Details of deployment vary between vendors, although some parts are standardized in Jakarta EE. To help with deployment, many IDEs (such as NetBeans) include automatic deployment, if you install them with a server. A more popular approach, not requiring some GUI IDE, is to use Apache Maven.

> The original Jakarta EE spec, known as J2EE, left much to be desired. Many additional frameworks have been developed to fill in the gaps or provide simpler APIs. But Jakarta EE keeps evolving, taking the best ideas from those other frameworks, and including them in Jakarta EE. Version 6 is considered pretty good all by itself.

Learning Jakarta EE means learning the new concepts and APIs to use. There is a lot to learn, including handling HTTP requests, producing HTML output from Java, Using the new persistence model to query and update the database (SQL is not needed), Creating and using transactions, sending and receiving messages and email, finding resources on the network, and so on. It is usually best to start small, learning one new technology at a time. (This is why Tomcat is popular; it handles a useful but simple subset of Jakarta EE.)

Because of the many additional concepts and APIs, Sun (and now Oracle) provides additional training and certification for Jakarta EE developers. Starting in 10/2011, Oracle introduced a new, mandatory training requirement for anyone interested in pursuing the highly coveted *architect* designation. Formerly known as the *Sun Certified Enterprise Architect* (SCEA), the newly branded *Oracle Certified Master, Jakarta EE 5 Enterprise Architect* examination requires candidates to complete a multiple-choice exam, submit an enterprise design project, and complete a proctored essay exam. Candidates must also take one of a half dozen topical courses offered by Oracle before the architect credential is granted.

> **The really, really, brief version:**
>
> Jakarta EE is mostly about creating web applications, the scalable and distributed servicing of requests from clients. That usually means separating the user interface code (the web tier) from the rest of the code (the business tier). The core of such web apps is the Servlet API in the web tier, and session beans and message-driven beans in the business tier.
>
> So Jakarta EE is basically all about applications where a client is expected to use a web browser to access a web app. This means creating an app that handles HTTP/HTTPS requests from web browsers, or SOAP (XML) requests coming in over HTTP/HTTPS for a Java web service (to support SOA). The other aspects of enterprise applications are messaging (JMS, or *Java Message Service*, or AMQP (Advanced Message Queuing Protocol)

which is an open standard from OASIS), and talking to a database (EIS tier, using the *connector* part of Jakarta EE).

In a Jakarta EE application, you don't write the low-level Java code to do all that. An application server provides those services to your app, and you just need to "plug-in" your code, by providing the classes and methods the application server will invoke when clients make requests. There are many application servers that meet Jakarta EE specifications (so your code will plug-in without problems), including Glassfish, Wildfly (formerly called JBoss), Geronimo, and some commercial ones (IBM's WebSphere, Oracle's WebLogic, Red Hat's JBoss). (But, you need to check the version of Jakarta EE each is compatible with!)

In addition, there are scaled-down application servers that implement a subset of the Jakarta EE specification, the most popular being Apache Tomcat. Tomcat is often used as the heart of other app servers, including some of the ones mentioned above.

Oracle has stopped support for the open source Glasfish server. Today I would suggest JBoss/Wildfly or Tomcat.

Jakarta EE started as a very low-level API with many important parts missing, and other parts very badly designed. There are large differences between the original J2EE, Jakarta EE 3, 4, 5, 6, and the current version, 7. **If at all possible, avoid all older versions and any old books.**

In addition to the standard Jakarta EE interfaces, there are a number of available *frameworks* that aim to simplify one or another aspect of creating web apps in Java. Such frameworks are a *wrapper* around one or more of the standard Jakarta EE interfaces, with a (hopefully) simpler interface. Struts, Wicket, Tapestry, and the core technology "Java Server Faces" (JSF), are web-tier frameworks that aim to simplify processing of requests. They all support MVC. Technologies like JSP and Facelets are *view* technologies and are used by these web frameworks (or you can use them directly) to provide a Java way to create HTML web pages automatically.

Other frameworks are designed to help with persistent data (the database, or EIS tier): JPA (*Java Persistence API*) is an API that is equally applicable to desktop Java or enterprise Java. It's an API for *object-relational mapping* (ORM, or OR Mapping), and combines lessons learned from popular (and not-so-popular) ORMs like JDO, Hibernate, Toplink, and others. Using this means avoiding the low-level JDBC API. Jakarta EE even includes a query language so you don't need to use SQL (and the results are collections of objects).

Spring is a collection of frameworks (about 20) you may hear about, designed by those who hated J2EE. Jakarta EE 7 is so much better than the original Java 2 EE that you probably won't need any part of the Spring frameworks. None the less, Spring continues to be popular and useful to land a job.

Learning Jakarta EE and the related web, XML, database, and other technologies will take years, so don't expect to understand it fully at first. Start with the latest Jakarta EE tutorial (currently for version 6).

It is possible to write a Java application that accesses such resources on the local computer. But in the real world, such central resources are on a different host. Also, some parts of the application may run on a different machine than other parts, and they still need to communicate. We can write code that uses networking APIs to contact these central resources. What's wrong with that? Plenty:

- Every change to the central resources will require every copy of the application to be updated. Customers don't like frequent updates, and if they don't update and the application fails, they will go to your competitor.
- By allowing remote applications to access your precious resources, all sorts of security problems can arise, including unauthorized use and denial of service.
- A remote application will need to read the central resources every time, from the network. It is usually not feasible to cache this on each and every client. So frequently access information, such as a catalog, will be slow to access. If you do cache information you risk it going *stale* (out of date), unless you check for changes frequently. No matter how you look at it, your central resources will have lots of network traffic to contend with. Network bandwidth is expensive.
- With remote applications doing all the work, logging and auditing become impossible: If something doesn't work right you won't know until the customer complains. You will not have a complete audit trail for legal and financial purposes. (Imagine a customer buys something, the credit card service is contacted directly and the customer billed, but the network connection back to your database fails and the order isn't completed!)

One simple way around this is to keep the application with the central resources, and only have a remote user interface. This is called ***client-server*** computing and it works very well. This scheme is been in use for over 20 years. Since some of the application is in one place and other parts are elsewhere, this is also called distributed computing, enterprise computing, or other terms depending on who you ask.

Definition of Terms for Enterprise Computing (*from Sun's Java Enterprise Cert*)

- **Scalability** – ability to support the required quality of service as the load increases
- **Reliability** – the assurance of the integrity and consistency of the application and all of its transactions. Reliability spans from the OS to the application, to the service provided.
- **Availability** – the assurance that a service/resource is always accessible
- **Maintainability** – the ability to correct flaws in the existing functionality without impacting other components/systems
- **Extensibility** – the ability to add/modify additional functionality without impacting existing functionality
- **Manageability** – the ability to manage the system in order to ensure the continued health of a system with respect to scalability, reliability, availability, performance, and security.

SOA and RESTful Web Services

Service-Oriented Architectures (SOA) provide a mechanism for very different systems to work with one another, even across programming languages and platforms. The basic concept is that if you have a proprietary system or a system with which you want other systems to interact, it is better to place a service in front of that system and then allow your other systems to interact with the service rather than directly with the destination system.

While SOA defines the architectural concepts, it does not specify the implementation details. That's where ***web services*** come in: web services are an SOA implementation. In the early days of web services, the majority of communication was performed using the Simple Object Access Protocol (SOAP), which defined a standard XML-based document format with applications could communicate. (SOAP was designed to replace the never popular CORBA standards.) SOAP evolved to be a very rich document format that allows you to transport data from one system to another and to invoke commands on a remote server; in the process of evolving it was no longer "simple".

The concept behind **REST** (*Representational State Transfer*) services is that a service exposes the state of its resources. For example, the service may be a music store that exposes its music library and the current stock of its items as its resources. Then diving deeper, a musical item may expose the name of the artist, album, and songs on the album, as its resources. If the client allows a user to buy an album then the client may update the quantity of the album by subtracting one. The key is to think about what you're exposing in terms of the state of your resources, rather than as a set of operations that manipulate that state.

A **RESTful web service** (also called a RESTful web API) is a simple web service implemented using HTTP and the principles of REST. Such a web service can be thought about as a collection of resources. The definition of such a web service can be thought of as comprising three aspects:

- The base URI for the web service, such as http://example.com/resources/
- The MIME type of the data supported by the web service. This is often JSON, XML or YAML but can be any other valid MIME type.
- The set of operations supported by the web service using HTTP methods (e.g., GET (fetch some data item, or a whole collection), PUT or POST (create new or update an item or whole collection), and DELETE).

Here is an example RESTful web services call to the Google Maps GeoCoder service, for the White House:

```
http://maps.google.com/maps/geo?q=1600+Pennsylvania+Ave,Washington,DC&
output=xml&sensor=false&key=mykey
```

And the response looks like the following:

```
<?xml version="1.0" encoding="UTF-8" ?>

<kml xmlns="http://earth.google.com/kml/2.0"><Response>

 <name>1600 Pennsylvania Ave,Washington,DC</name>

 <Status>

 <code>200</code>

 <request>geocode</request>

 </Status>

 <Placemark id="p1">

 <address>The White House, Washington, DC 20500, USA</address>

 <AddressDetails Accuracy="9"
xmlns="urn:oasis:names:tc:ciq:xsdschema:xAL:2.0">

<Country><CountryNameCode>US</CountryNameCode><CountryName>USA</
CountryName>

<AdministrativeArea><AdministrativeAreaName>DC</
AdministrativeAreaName><SubAdministrativeArea>

 <SubAdministrativeAreaName>District of
Columbia</SubAdministrativeAreaName><Locality>

<LocalityName>Washington</
LocalityName><PostalCode><PostalCodeNumber>20500</PostalCodeNumber>

 </PostalCode><AddressLine>The White
House</AddressLine></Locality></SubAdministrativeArea>

 </AdministrativeArea></Country></AddressDetails>

 <ExtendedData>

 <LatLonBox north="38.9062952" south="38.8889273" east="-77.0203584"
west="-77.0523732" />
```

```
 </ExtendedData>
 <Point><coordinates>-77.0363658,38.8976118,0</coordinates></Point>
 </Placemark>
</Response></kml>
```

Client Server Computing

Client server systems come in different configurations:

- **Thick (fat) client** — A standard (or web-start) Java application or applet that communicates across a network to the server. The client is the user interface or ***presentation layer*** *or* ***tier***. Such systems can use different approaches to networking: using TCP/IP and define a custom protocol for the application, use HTTP GET and POST protocols to pass messages and data, use SOAP (See SOA=*service oriented architecture*, WSDL (*Web Service Description Language*), UDDI, apache AXIS for Java SOAP to pass objects back and forth, and others too.

  With a fat client, most if not all the work is done at the client. This design requires large executables to be downloaded. Updates require another d/l. The server (if any) only provides a central storage place with very little business logic.

- **Chubby client** — all the grunt work (SQL etc.) is done on the server along with all the business logic. However, the client has a lot of (JavaScript) code which runs to provide the user with a "rich experience". This code is reloaded every time it changes with no interaction by the user. AJAX (and Web 2.0) falls into this category.

- **Thin client** — A web browser talks to a web server, which presents a form typically using JSP or servlet technology. (Of course, PHP and ASP pages could be used too, but in this course, we will stick to learning only one language, Java, to keep the amount of languages a developer needs to master to a minimum.) Note the client in this case is the JSP/servlet web server code. This can get confusing (the client of the server is on a web server). This is how most web pages currently work.

The thick client has the advantage of a rich and fast user interface, that can include extra features such as caching. Also, a thick client can operate even if an Internet connection is unavailable (common for mobile users), and in any case appears much more responsive (no network latency for updates). However, the thick client has many of the same problems (especially deployment and security) that plague stand-alone (non-client-server) applications. Java WebStart was an attempt to mitigate this problem but was never widely used, and has been removed since Java 11.

The thin client is limited to what a web browser will show. This can be more difficult as a good-looking web page requires use of HTML/XML, JavaScript, and CSS. Worse, there are browser compatibility issues so some page may not look good on some older, different, newer, or different platform web browser unless you go for dull, boring user interfaces. Still, this approach does keep everything in one place, so no deployment and few security issues arise. Because the processing is done on a server, the user experience may be slower than with a thick client. If the Internet is unavailable, the thin client fails to work at all. The thin client also needs to worry about issues such as the back button, and multiple browser windows open at once.

Today the thin client is the preference of many developers especially with *AJAX* which provides some thick client benefits. However this changes over time as more powerful laptops with more RAM become popular, more different web browsers become incompatible, and networks become more or less reliable and secure.

Either way the client must communicate with the server. Often this is with RMI or other means such as CORBA or even very low-level *sockets*. (CORBA was supposed to do this but ended up too complex and although Java supports this it is not often used. This situation may be changing though. Sockets is too low a level to develop significant distributed applications with.)

Multi-Layer or Multi-tier Design

The client talks to your application on the server, which access the central resources. There are different types of resources, and even from the same type (such as a database), there are different vendors. Often you need to access mainframes (yes these are used in the world!) or legacy systems running proprietary software that can't be easily (or at all) updated. (Even if you don't need that now, your application may need to support this someday!)

Such resources are known as ***Enterprise Information Systems*** or **EIS**. It has been found to be a generally useful thing to split your application into several parts:

- **Presentation layer** (or **tier**) - Separating the client code from the rest means you can have different sorts of clients for the same application: a web interface or a standalone application. Also you can have designers make functional and attractive UIs, and developers (with little artistic talent) work on the logic of the application. With Jakarta EE 5 a framework called Java Server Faces (JSF) is used to develop the UI for a Jakarta EE application. Often a web interface is developed on JSF that uses another standard technology, servlets and Java Server Pages (JSPs).
- **Business logic (BL) layer** (or **tier**) - This is the meat of your application. The client (presentation layer or tier) code talks to this code. In Java terms, the

client will create objects of these classes and invoke their methods; the results are then formatted and displayed to the user.

- **Data access and integration layer** (or **tier**) - This is the code that talks to the EIS components. This code hides the messy details from your nice simple business layer code. The code in the business layer will invoke methods of objects in this layer. This layer can also be known as the ***EIS layer*** *or* ***tier***.

Here's an example: The user clicks a link on a web page to view their current balance (before they get to this page assume they logged in somehow). The link (which may be a form's submit button or just a link) sends the request to the client, typically a servlet. The servlet in turn invokes the business logic code to get the client's payment information, then requests the amount of outstanding orders, and adds all that up and returns the data to the presentation layer, which formats it as HTML. The business logic methods don't invoke JDBC or networking API directly, rather they invoke methods in the data access and integration layer. This code contains the JDBC and SQL or other code to get the information from the EIS (say the Visa interface to the bank, PayPal, or we track it ourselves in a database).

There are many different ways to organize this code. The common way is to use JPA *entities* (Enterprise JavaBeans (EJBs)) in the EIS and BL layers, but of course it may be better to use a different technology or organization in any given case. Knowing when to use which technology needs a lot of background knowledge and experience, which is why enterprise computing is so hard to learn!

Scalability, Reliability, and Availability

You won't get a lot of business done with a single client talking to a single server. In some large-scale systems, it is hard to predict the number and behavior of end users.

Scalability refers to a system's ability to support fast increasing numbers of users. [*From Wikipedia:*] Scalability is a desirable property of a system, a network, or a process, which indicates its ability to either handle growing amounts of work in a graceful manner, or to be readily enlarged. For example, it can refer to the capability of a system to increase total throughput under an increased load when resources (typically hardware) are added.

The intuitive way to scale up the number of concurrent sessions handled by a server is to add resources (memory, CPU or hard disk) to it. To ***scale vertically*** (or scale up) means to add resources to a single node in a system. This typically involves the addition of CPUs or memory to a single computer. Such vertical scaling of existing systems also enables them to leverage virtualization technology more effectively, as it provides more resources for the hosted set of operating systems and application modules to share.

Vertical scaling has limits. You typically run one JVM per server, with many threads (one per request). However this design only scales up to one to four thousand concurrent requests, no matter how many CPUs are available. Also, 32-bit systems cannot use more the 2GB of RAM per process (per JVM). Tasks that use I/O heavily can only use so much CPU power; the bottleneck becomes the IO. Running multiple JVMs per server is possible to make use of additional RAM but leads to other problems, such as resource sharing, singletons, synchronization, etc.

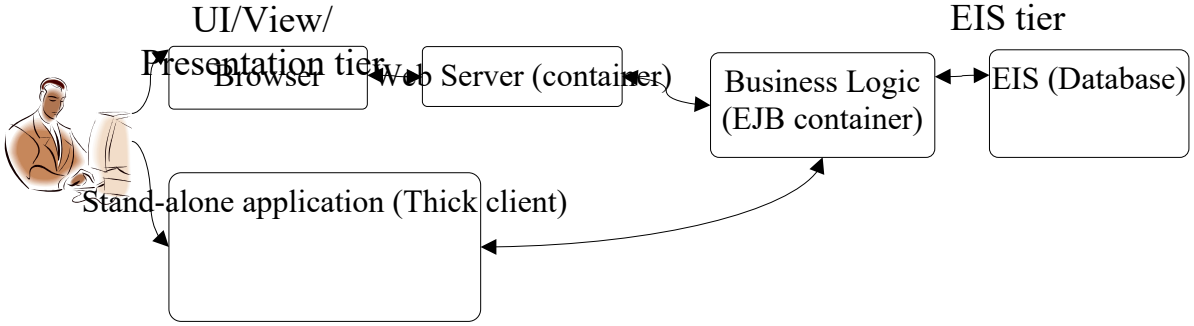
To ***scale horizontally*** (or scale out) means to add more nodes to a system, such as adding a new computer to a distributed software application. An example might be scaling out from one web server system to three. As computer prices drop and performance continues to increase, low cost commodity systems can be used for high performance computing applications such as seismic analysis and biotechnology workloads that could in the past only be handled by supercomputers. Hundreds of small computers may be configured in a *cluster* to obtain aggregate computing power which often exceeds that of traditional RISC processor based scientific computers.

Clustering or ***grid computing*** are horizontal scaling alternatives that allow a group of servers to share tasks and operate as a single server logically, enabling the system to serve multiple clients at the same time.

Note you may have a cluster of web servers and a single application server, a single web server and a cluster of application servers, or (independent) clusters of each.

This approach requires ***middleware***, software that can direct incoming requests from users to available thin clients, and route thin client requests to available servers. This is called ***load balancing***. A good load balancer/middleware is very complex and most people buy these instead of writing their own. A server running middleware, which in turn invokes your application code, is called an ***application server***.

A typical picture of an enterprise application cluster looks like this:



Clustering is a popular technology that provides highly available and scalable services, with fault tolerance. The lack of details in the Jakarta EE specification means different vendors implement clustering differently. Some popular products include the BIG-IP™ load balancer from `F5.com`. Tangosol's Coherence™ provides a distributed data management platform which can be embedded into most popular Jakarta EE containers to provide clustering environment. Coherence also provides distributed cache system which can share java objects among different JVM instances effectively. See `www.tangosol.com/`. Other solutions include Terracotta DSO and JavaSpaces.

Additionally a single-server solution (add memory and CPU) for scalability is not robust due to a single point of failure. Mission-critical applications such as banking, billing, medical systems, etc., cannot tolerate any service outages. These services must always be accessible with reasonable and predictable response times. Clustering is a solution to achieve this kind of *high availability* by providing redundant servers in the cluster in case one server fails to provide service.

Clustering in the Web tier is the most important and fundamental type of clustering. Web clustering technique includes web server load balancing and HTTP Session failover.

"JavaGroups" is currently the communication layer of JBoss and Tomcat clustering, and provides core features such as "Group membership protocols" and "message multicast", which are very useful in making clustering work. For more information about JavaGroups see jgroups.org.

Another scalability problem is usually the Database management system. This is the most common bottleneck for most of enterprise applications, since the database is normally shared by the JVM threads. So effectiveness of database access, and the isolation levels between database transactions, will affect the scalability significantly. Once possibility is to put most of the business logic resides in the database, using stored procedures, while keeping the Web tier very lightweight, just to perform simple data filtering actions and process the stored procedures in database. However, this architecture causes a lot of issues with respect to scalability as the number of requests grow.

Question: What issues arise with a clustered design that you don't have with a simple application? [*Adopted in part from "Mastering Enterprise JavaBeans" by Ed roman, et. al., (c) by Wiley.]*

- **Remote method Invocation (RMI),** so client code can call server code.
- **Load balancing**. This is one of the key technologies behind clustering, which obtains high availability and performance by dispatching incoming requests to different servers. The middleware must be able to know if some application server is busy and direct the client request onto a different application server.

A load balancer could be a hardware product with SSL accelerators such as the **F5 Load Balancer**, or just another web server with load balancing *plug-ins*. Even a simple Linux box with `iptables` (packet filter firewall) can load balance very well.

Dispatching can be implemented with *round-robin*, random, source-IP based, or work load based algorithms.

In addition to dispatching requests a load balancer should perform some other important tasks such as "session stickiness" to have a user session live entirely on one server and "health check" (or *heartbeat*) to prevent dispatching requests to a failing server. Sometimes the load balancer will participant in the *Failover* process (see below).

- What if a web or application server crashes? The middleware must be able to restart (or at least back out of) any transactions and user sessions with a different server. This is known as ***transparent fall-over***. There is usually a time delay associated with this in seconds (could be minutes if this feature isn't carefully designed). Note that data corruption must be avoided even if a server fails in the middle of a session; this is called *fault tolerance*.
- **Transaction processing**. If you use files or lightweight databases, you need to worry that two or more sessions might simultaneously access the same piece of data. Even with powerful back-end databases you need to be careful. This is because DBs use *connections* to separate transactions, and as these are slow to setup, you normally use a technique known as *resource pooling* to reuse connections. Also, the user session might start on one application server (using one DB connection) and finish on another (using a different DB connection).
- **Application server state**. Ideally all persistent information is stored in the EIS so nothing is lost if the session moves to a different server (for instance when one server crashes). However this design is slow and causes a lot of network traffic to the EIS resources. So, if you store some information on the application server, does it need to be copied (replicated) on all servers? When a server restarts, how is that information loaded/initialized?

  Just about every web-based application has some session state, such as remembering whether you are logged in or the contents of your shopping cart. Because the HTTP protocol is itself stateless session state needs to be stored somewhere and associated with your browsing session in a way that can be easily retrieved the next time you request a page from the same web application. This can be done in several ways; the best is to store a session "cookie" on the browser. This cookie is sent to the server with every HTTP request. A session cookie expires when the browser session terminates (that is

when the browser closes). Also the servlet container (web server) can associate a timeout for sessions.

When a browser visits a stateful web application this application creates a session object in memory to store information for later use. It sends the browser a globally unique `HTTPSession` ID cookie which can identify this session object in the cluster. There are methods to add and access data in a session object.

Other options include using complex and long URLs that contain session information, or hidden form fields.

Note the problem of load balancing and fail-over with session data; if the session can be dispatched to a different server next, where is the session data kept? The middleware must make one or more copies of the session objects on different servers, and keep them synchronized as changes occur. (Using the EIS DB is simple by very slow.) Tomcat5 copies all session objects to all servers on all updates. JBoss and other servlet containers follow a more efficient approach of paired servers or paired sessions, so each session object is backed up onto one other server. IBM's Websphere uses a special high-speed central DB for sessions.

No matter what the choice you may need to configure ("tune") the frequency of session replication, and the granularity (whole session or just the modified bits).

- **Deployment**. How can you cleanly update your application code an all the servers without stopping all of them at once?
- **Clean shutdown of application server**. When a server must be stopped for updates or other maintenance, you don't want to abort the currently running sessions.
- **Logging and auditing**. You need a central log for determining if problems exist (and if so, what went wrong), and for auditing (information about what was done, by whom, and when). Auditing is required (compliance) by various legal, accounting, and regulatory bodies (FBI, SEC, FTC, ...). Be careful about logging user identification data: don't log if possible, else use *data blinding* (substitute user IDs with unique random strings: you can still relate all messages to that user but don't know which user it is). Use encryption to protect logs from modification (digital signature per entry, plus manager sig: *dual controls*) and access.
- **Systems management (IT)**. Who gets notified (and how) when a problem occurs, or in the event of a disaster (power loss, fire, ...)?
- **Distributed applications**. One object's methods may invoke another object's methods. With distributed applications, the second object may be loaded in the

same JVM, on the same server but loaded by a different JVM, or loaded on a different server (and thus a different JVM). So, how do you make sure to not load the same class twice? What happens if two copies of some class are loaded (and you change static properties), or you create too many copies of some object (e.g., a DB connection to your Oracle server)? Given the many different vendors of application servers, where are your classes put so that the JVM classloader can find them? (This is why you often use a custom `ClassLoader`.)

- **Multi-threading issues**. Since each web server will likely have to handle several user sessions, and will each application server, you need to support multi-processing or multi-threading.
- Support ***message oriented middleware***. Some EIS are web services that you talk to, not by RMI or by some proprietary network protocol, but by passing (simple) messages back and forth.
- **Resource pooling**. The obvious candidate is database connection pooling, but other resource that take time to setup and initialize, and to tear down when done, can often benefit from having a bunch of such resources pre-created in a pool (a collection of some kind), ready to go. When you need (for instance) to send a SQL query to the database, you grab a pre-built connection from the pool, pass it the SQL code, and run it. After the connection returns the results, you clean up the connection and return it to the pool.
- **Security**. You must normalize and validate all user input, and possibly sanitize it as well (to prevent SQL/XML injection, XSS): be careful of HTML constructs such as "<", "&", ">", other illegal characters, length restrictions, numerical limits, etc. User data used for filenames has extra restrictions: "..", "/" and "\", etc. Validate everything, not just user input fields but cookies and all data. After a user has been validated, issue a new session ID. Invalidate session IDs at logout (or after timeout). Use a quality source of random numbers (i.e., have a large entropy pool). Avoid native methods (no security). Store sensitive data (credentials) security encoded, even in RAM. Hide all low-level error messages from user (show only generic message, ok to log full message.) Mark all *scaffolding* (*instrumentation*)/*testing* code with annotation, to easily find and remove before production release. Use `BigInteger`/`BigDecimal` to convert string input data to numbers, then check limits before converting to `int`/`float` so no over/underflow errors.
- **Caching**. All sorts of data are frequently needed by applications. It makes sense to have copies on each server for quick access. But these caches need to be updated when the data in the EIS changes.

- **Business tier object management (lifecycle).** These objects need to be created when the client makes a request and/or a session is started. They need to be destroyed when no longer needed.
- **Persistence of objects**. An object representing some user's shopping cart needs to persist for a long time (minutes). What happens if the application is restarted on a different server, or if the server crashes? Such objects need to be persistent. You could of course write your business logic so that every change by the user is immediately sent to the database, but such a design causes way too many database connections! If too many users try to use your system at once you will overwhelm your DB and/or network, and it may cause data loss, dropped sessions, or crashed DB or servers.

  Another problem is exactly how to store objects into relational database tables (known as ***O/R Mapping***, ORM, or *object/relational mapping*). With Jakarta EE 5 and newer, **JPA** (*Java Persistence API*) does this. You only need to define objects, then run an SQL-like query language to populate the fields of these objects or to store any changes made back to the DB. (Of course, someplace you need to define the DB, username, password, and other connection information.)

  A common question is why bother using all this ORM stuff if you already know SQL? One reason is the SQL isn't portable; it isn't really standardized except for the most basic uses. Using some ORM also gives you:

  - caching (first level & second level)
  - lazy loading of related objects
  - transaction support (EJB, transaction context propagation)
  - better/cleaner query language
  - fetching of related objects in a join, good for performance

  Many expensive DBMSs offer useful features not available if using ORM, such as fine-grained access control, hierarchical queries, triggers, stored procedures, and Analytic Functions. However, most ORMs allow one to mix in SQL when needed.

  On any project you must weigh factors such as portability, need for advanced features, local expertise, and ease of development.

  A number of technologies for this are available: JDBC (no mapping, straight SQL), JDO (mostly a dead project now), Entity EJB (either *bean-managed persistence, BMP,* or *container managed persistence, CMP*), **JPA** (replaces Entity EJBs), Hibernate, Oracle Toplink, and others. Each has unique strengths and weaknesses, so there is no one best solution.

  Example of JPA code:

```
import jakarta.persistence.Entity;
import jakarta.persistence.Id;

@Entity public class Customer {
@Id private Long id;
private String firstname; private String lastname;
private String telephone; private String email;
private Integer age;
// constuctors, getters, setters
}

//...

@PersistenceContext(unitName="MyDB")
EntityManager em;

Employee emp = new Employee(...);
emp.setName("Miller");
em.persist(emp);
// ...
Employee emp = em.find(Employee.class, "Miller");
// The DB updates automatically:
emp.setAge( Integer.valueOf(21) );
// ...
if (emp != null) em.remove(emp);
```

The `EntityManager` takes a parameter that refers to a specific *persistence unit* ("`MyDB`" in the code above). Persistence units are declared in the standard `persistence.xml` file, and contain information such as the database to use, the JDBC driver, and any implementation-specific properties. Note the table and column names are automatically derived from the `Customer` class (but may easily be over-ridden if desired).

The source code for Microsoft's ADO.NET Entity Framework (EF) has been made available as open source software under the Apache 2.0 license (7/2012). This code is the ORM used for Microsoft's .NET framework.

- **Naming services**. In a clustered environment, how do different objects find each other? How does the client "know" where to find the server? How does the various business logic objects "know" which EIS to use and how to contact it? How do you refer to some EJB or other object, when that object may not be loaded in the local JVM but rather some other server's JVM? An analogy is "how does a web browser know where some website such as www.wpollock.com is?"

The obvious answer is to hard-code that information everywhere. Then when your DB changes its IP address you have to re-write and then deploy all new versions of your application. Not a good idea! The answer is a name-to-resource database (just like a browser uses DNS to translate a name "wpollock.com" to an IP address.) Your objects can "lookup" some resource by name, and then use the returned information to access that resource. This is the **JNDI API**. Note some servers don't use a global (cluster-wide) JNDI DB but rather one per server (Sun and IBM). However these use special *agent* software to achieve the same effect.

Enterprise applications are often quite large: either a large number of users (so performance matters), a large geographical area, a large amount of data to organize, or some combination of these factors. Such applications end up as very large applications (in terms of code size). Because of the size it is quite common to purchase some parts "off the shelf" from vendors such as Sun, IBM, BEA, Oracle, etc. This is true of the middleware, load balancing, web servers, backend databases, and especially common application services (transactions, persistence, logging, creation/destruction of business objections, ...). Other parts (the presentation web pages/forms/servlets/JSPs, business logic, and sometimes EIS tier code) will be developed by independent (teams of) programmers.

Many of the issues above (and may others) are common enough that a market was created for "*application servers*", systems that included middleware and support for persistence of objects, transactions, load balancing, systems management, etc. To date there have been about 50 such products you can purchase. In the past, every one of these application servers had proprietary interfaces. So your business logic classes (and other code) had to be written for one specific brand of application server, or you had to do all that code yourself. If you picked the wrong company or product, you probably would need to start the application over nearly from scratch.

The answer to the need for off-the-shelf services that don't have proprietary specifications and interfaces is Jakarta EE. **Jakarta EE is a *specification*, not a product**. So if the database product you bought yesterday is Jakarta EE compliant you can be confident it will work with the Jakarta EE application server you buy tomorrow. And if you develop Jakarta EE business logic components (sets of classes), you know they can be deployed without change in any Jakarta EE compliant application server (assuming the same Jakarta EE version, of course).

Some of the most common Jakarta EE specifications include EJBs (Enterprise Java Beans), servlets, JSPs (Java server pages), JDBC, JavaMail, JMS (Java Management Services), etc. Often the data is passed around using XML. Security technologies such as user authentication and data encryption (and IPsec for network security) are common.

Containers are standardized runtime environments that provide specific services to components. Components can expect these services to be available on any Jakarta EE platform from any vendor. Thus, we talk about servlet (or web) containers, or full Jakarta EE containers (usually referred to as EJB containers).

EJBs come in three different types: session beans, entity beans (now replaced with JPA), and message beans. Note an EJB is different from a JavaBean. Not all applications use all types of EJBs but *session beans* are common.

Because the business logic deployed in an application server is so often a set of EJBs, an application server compliant with Jakarta EE is also called an EJB container. (Similarly, web server part of the client code is often deployed in a Web application container or servlet container, which is just a web server that knows about servlets, JSPs, and WARs.) The client computer where application code runs (especially for thick clients) is sometimes referred to as an application container.

A typical enterprise application is used this way: a user connects to the client part of an enterprise application, starting a *session*. The client is often implemented as part of a web server with a servlet or JSP page. Once the session is established the client and application exchange messages or even objects (which are serialized and send as regular messages) using XML and HTTP or other network protocols. The application often gets data from other servers, perhaps a central database server. Updates (orders, reservations, ...) are handled with *transaction management*. When finished the session is terminated (or times out).

To fully understand enterprise applications (well enough to design them properly) you need to master many technologies including web server technology, database technology, networking technology, security technology, multithreading/clustering (for high volume and good performance), and others too. Security and encryption technology should also be understood. You don't have to be able to write code for all these technologies but you need to know enough to decide which technologies you need for a given application, how to buy a product with the required features, and how to write a bit of "glue" code to tie the different pieces together.

Sun also provides a series of classes that can be used to implement parts of a Jakarta EE enterprise application (such as JDBC to talk to legacy databases using SQL, messaging interfaces, etc.), as well as other useful bits such as deployment tools.

A competing technology is the .NET series of standards from Microsoft. But these are not being widely adopted (at least not yet outside of MS).

Jakarta EE Development Issues

One problem today is that the Internet is a hostile place. The data must be encrypted when send and the sender (and recipient) must be verified using special

security methods. Since many parts of the Internet are protected by firewalls today, many enterprise applications with thick clients use port 80 (the standard web server access) port to go through the firewalls. This creates additional security problems.

Another problem is that not all parts of an enterprise application are standardized through Jakarta EE. A common example is the configuration of the application server or web server. Newer editions of the Jakarta EE standard provide the standards, tools, and Java classes help solve these problems.

In spite of all the care that makes Jakarta EE applications work on nearly any server, some of the APIs used are "low-level": full of messy details that are difficult to make correct, efficient, or both, or are just plain repetitive and boring for most programmers. To help with this a number of technologies were created, not really part of the Jakarta EE standard but built on top of these APIs and specifications, that are at a "higher" level and hopefully easier to learn, use, and reuse. These are called ***Frameworks***. Examples include Struts, JavaFaces, and Apache Tapestry, to simplify the web/presentation layer development for user interfaces (so you can avoid servlets and non-standard custom JSP tag libraries); Hibernate, JDO, and others to manage persistent data (so you can avoid JDBC; however there is now the JPA or *Java persistence architecture* so these frameworks are less needed), Spring and ***seam*** to simplify overall Jakarta EE development (especially of CRUD—Create, read, update, delete—apps), and others such as **AspectJ** (See Wikipedia for this). Other frameworks include JBI (`servicemix.org`) and RAILs (for Ruby). (Rails doesn't scale to clusters though, all state in in a single DB.)

Design pattern such as "Singleton" use `static` variables to share a state among many objects. For example, tracking the number of users. **Using `static` variables works well on a single server but fails in a cluster.** Each instance in the cluster would maintain its own copy of the `static` variable in its own JVM instance, thereby breaking the mechanism of the singleton pattern. A preferable way in a cluster is to store all state data to a database.

Caching state is a big problem. Putting all state in the backend DB works but can be very slow. Storing some state in the presentation or application tier can greatly enhance performance, but *stateful session beans* cause a lot of trouble with clustering, failover, performance (the non-functional requirements, or NFRs, are also called *operational requirements*). One technique is only use stateless session beans, and hold any *session state* in the servlet containers (the presentation tier). Another is to replicate the state on one or more (or all) application and/or web app servers. This required more sophisticated middleware/app servers.

Similar problems arise in a cluster if you store any data in files. Besides not using files, consider a SAN (*storage area network*) to provide a central file service all servers share.

Remember that although there are a lot of APIs and technologies as part of Jakarta EE, all are optional! You only use what your application needs. Usually there is more than one way to design the application too. Knowing the best (or at least a good) design takes experience.

The specific API's mandated for Jakarta EE 5 are:

- Enterprise JavaBeans (EJB) 3.0
- Servlet 2.4
- JavaServer Pages (JSP) 2.1
- Java Message Service (JMS) 1.1
- Java Transaction API (JTA) 1.0
- JavaMail 1.3
- JavaBeans Activation Framework 1.1
- Jakarta EE Connector Architecture 1.5
- Web Services for Jakarta EE 1.1
- Java API for XML-based RPC (JAX-RPC) 1.1
- Java API for XML Web Services (JAX-WS) 2.0
- Java Architecture for XML Binding (JAXB) 2.0
- SOAP with Attachments API for Java (SAAJ) 1.3
- Java API for XML Registries (JAXR) 1.0
- Java 2 Platform, Enterprise Edition Management API 1.0
- Java 2 Platform, Enterprise Edition Deployment API 1.1
- Java Authorization Service Provider Contract for Containers 1.0
- Debugging Support for Other Languages (JSR-45)
- Standard Tag Library for JavaServer Pages (JSTL) 1.1
- Web Services Metadata for the Java Platform 1.0
- JavaServer Faces 1.2 Requirements
- Common Annotations for the Java Platform 1.0
- Streaming API for XML (StAX) 1.0
- Java Persistence API 1.0

Issues for selecting Jakarta EE application servers

Jakarta EE branding, pluggable JRE, conversion tools (Old J2EE to new Jakarta EE), 3rd party JDBC driver support, Lazy-loading (Load beans on-demand), deferred DB writes (don't update the DP until commit), Data Cache (some servers us *pass-through* so every access results in a slow DB operation), Presentation-EJB integration (one server with one JVM doing both), scalability references, high availability, security (JAAS support), IDE, UML Editor integration, intelligent

load balancing, transparent fail-over (for stateless sessions), clustering with transparent fail-over, support for optional Jakarta EE features: Java Management Extension Jakarta EE Connector Architecture (integration of EIS components), Hot Deployment (no reboot needed to re-deploy), EJB instance pooling and automatic generation, clean shutdown, some useful EJB components included, ...

Almost every Jakarta EE project uses ***object caching*** to improve performance, and all popular application servers provide some type of caching. Caching is not currently part of the Jakarta EE specs, so every vendor does this differently. Note these caches are typically designed for a standalone environment and can only work within one JVM instance.

Caching is needed because some objects are so complex that creating a new one will cost much (i.e., DB lookups across the Internet). So the server maintains a pool to reuse the object instances without further creation.

You gain performance only if the maintenance of the cache is cheaper than objects creation. In a clustered environment each JVM instance will maintain its own copy of the cache which should be synchronized with others to provide inconsistent state in all server instances. Sometimes this kind of sync will bring worse performance than not caching!

JBoss began as an open source EJB container project six years ago has become a fully certified Jakarta EE application server with the largest market share, competitive with proprietary Java application servers in features and quality. JBoss isn't just a Jakarta EE server. It has a dynamic architecture which allows one to alter the services to make Jakarta EE work the way you want, or even throw Jakarta EE away completely. The newest version has been renamed to Wildfly. Sun Jakarta EE SDK contains ***Glassfish***, another Jakarta EE application container. Others include *Resin*, *Geronimo, CarbonAS*, Oracle's *WebLogic*, and *WebSphere*. For learning purposes, I recommend Sun's free *glassfish*. (JBoss is very popular.)

Perhaps the most important aspect of clustering is memory replication. In memory replication, information about the state of user sessions with deployed applications in a server instance is replicated to a *peer* instance in the cluster. With the Glassfish Jakarta EE container, each instance in a cluster sends session-state information to the next instance in the cluster, its replication partner. The order of instances in a cluster is based on the names of the instances. As the session-state information for a deployed application is updated in any instance, it is replicated around the cluster.

Glassfish also has a load balancer plug-in. The load balancer is responsible for distributing workload among multiple server instances. It also takes part in rerouting a session when an instance fails. If a GlassFish v2 instance in a cluster fails, the load balancer reroutes the sessions interacting with that instance to another instance in the cluster.

Java Management Extensions

[From: http://www.theserverside.com/articles/article.tss?l=JMXWebApps]

The ***Java Management Extensions*** (**JMX**) standard provides a standard for the management of applications, as well as application servers and other infrastructure software. The point of this is to improve productivity of your support (and development) staff, and to improve efficiency (reducing downtime of servers, and other efficiency measures). JMX makes it possible to manage and monitor applications using a choice of management systems and consoles, including ***SNMP*** (*simple network management protocol*, a common standard) consoles. It also simplifies the task of making applications manageable, i.e. management instrumentation, and enables improved management of deployed applications in the enterprise. JMX provides a lot of management value with little effort.

While management tasks should be part of the design requirements, it isn't always possible to anticipate these needs. It's hard to change the application software at that point. This is where business-specific application management can be very useful.

The first step in managing applications is monitoring availability and performance. In order to minimize downtime and avoid disruption to business operations, you must monitor and measure the availability of applications (and their servers). Also the performance of the application needs to be tracked. This ensures the application performs to the service levels needed, and enables a quick response if the application is under-performing. Performance is usually measured by counting key business method invocations or user transactions, in a given interval of time (say per 5 minutes).

The required performance of some service is reported as one or more ***service-level objectives*** (SLOs). SLOs are specific measurable characteristics including availability, throughput, frequency, response time, quality of service, and sustainable load. Developers need to monitor production services to ensure they meet their SLOs. For services that become popular, developers will try to refactor the code to improve the SLOs.

> Related to SLO is ***service-level agreement*** or SLA. An SLA is a formal document or statement that makes a commitment to customers, usually by listing several SLOs that must be met. Often, the SLA is legally binding with penalties if the service provider cannot maintain the service as required.

Beyond monitoring availability and performance, managing business-specific aspects of the application can be helpful in maximizing productivity. For example, a banking application may require the ability to monitor cash withdrawals to ensure sufficient cash at a required ATM or branch office. Another example is the need to monitor inventory levels for an on-line store. While some of these

management needs may be anticipated and designed into the application, many of these needs arise from production use of the application.

Common web applications may include financial applications like banking, insurance, or brokerage applications, as well as a number of other intranet applications like HR, call-center, and other applications. All of these can benefit from management, to some degree or another. Work-flows and click-trails can be studied to make the application's interface easier to navigate, marketing and other data can be collected to base business decisions on, inventory levels can be monitored, etc.

Here's a simple example: an application that allows users to transfer money between accounts using a set of JSP pages. To complete the transaction, the user navigates through four pages: a start page, a select transaction details page, a confirm transaction page, and a transaction complete page.

In this example, as in many web applications, the user navigates a series of pages to accomplish a task or transaction. Depending on the application, it may be useful to know whether users are having difficulty completing these tasks. For example, if a number of users start a transaction but fail to complete it, it may indicate some work needs to be done on making navigation easier. You can use the session tracking capabilities available with JSP to track the data we need on how users are doing.

In addition to tracking user success in completing their transactions, for this specific application it may be useful to track aspects of the money transfer. For example, we may wish to send a management notification when the transfer amount exceeds a threshold.

The JMX architecture is based on a manager and agent model. The *agent* typically resides with the application server and provides *management instrumentation*, i.e. application data and control. The *management console* collects and presents management information from one or more agents on the network.

JMX management instrumentation is a set of Java components called ***MBeans***. The `MBean` components are Java classes that fit design patterns based on the type of `MBean`. The defined `MBean` types are *Standard*, *Dynamic*, *Open*, and *Model* `MBeans`. JMX agent `MBeans` are all registered with a common `MBeanServer` interface, which remote management consoles access to get management information from the agent. (A bit like RMI.)

Standard `MBeans` provide a defined interface that follows a naming convention. The interface class is named by adding "`MBean`" to the name of the standard MBean class. For example if you call a *standard* MBean `MyJspStats`, then the MBean interface will need to be named `MyJspStatsMBean`.

Determining the size of an application (determining resources needed such as number of servers in a cluster)

You can simulate a system using queuing theory. A number of tools are available (none for free AFAIK) that allow you to adjust various parameters, such as the number of customers, time to process each session, etc., and graph the results in terms of response time and max load versus cluster size. Or you could hire a math magician to implement the queuing models and provide you with the numbers. (But playing with a GUI Java applet with Sliders and TextFields for input is more fun!)

Those numbers can be learned either by experience or by observing a similar (e.g., the current website) system and collecting statistics. (An interesting project is to write a program to parse a month's worth of server logs, to determine number of customers number of sessions, number of page hits per session, and average length of time for each session and each page load.)

What constitutes acceptable response time varies per organization, but there are publicly available marketing results that show a page must be rendered in under X seconds or customers will go elsewhere. X is usually quoted as either 4.something or 8.something seconds, as far as I remember.

Now, if you can get acceptable response time on a single server, then you can avoid cluster issues and the project is either small or medium. If a cluster is needed, it is large (some experienced developers might say medium or large, but if you are new to this, it is large!)

Most useful websites require persistent storage and the common solution is to use a database system. If you can get acceptable performance with a single host providing both the web service and DB service, the project is small.

Usually a separate server will be needed for each, in which case you need to deal with networking and possibly encryption (if you use credit card numbers for instance). To me, these extra dimensions make the project medium or large.

Until you gain enough experience to estimate a project's scope on your own, one option is to hire a "mentor" consultant who has the experience and who can get you started on your project with a good estimate of the resources needed. Often this is cheaper than hiring a consultant to do the project for you.

Depending on local expertise available, a small project need not be done in Java. However medium and larger projects are often easier if done using Jakarta EE technology, which provides a (sometimes) simpler API for the common tasks needed. Usually a servlet container will do for small and medium projects, and an EJB container will make life easier if you have a large one. Also the extra frameworks available and common (seam, struts, spring, hibernate, ...) handle these

tasks at an even higher level, and in theory that makes the project more manageable.

My experience is that learning too many frameworks for your first big (medium or large) project is unlikely to save time on that project. But maybe subsequent ones. Indeed, practicing re-doing an existing website using a different framework is a good way to learn the frameworks. This may be true even (or especially) for a small site that doesn't really need the framework.

Synchronization Issues Affecting Scalability

[http://www.theserverside.com/tt/articles/article.tss?track=NL-461&ad=664753USCA&l=ScalingYourJavaEEApplications&asrc=EM_NLN_4588276&uid=3138114]

Let's take a real-world case as an example. This is an ERP system for manufacture, when tested its performance in one of the latest CMT servers (2 CPUs, 16 cores, 128 strands), we found the CPU usage was more than 90%. This was a big surprise, because few applications can scale so well in this type of machine. Our excitement just lasted for 5 minutes before we discovered that the average response time was very high and the throughput was unbelievable low. What were these CPUs doing? Weren't they busy? What were they busy with? Through the tracing tools in the OS, we found almost all the CPUs were doing the same thing - "`HashMap.get()`", and it seemed that all CPUs were in infinite loops. Then we tested this application on diverse servers with different numbers of CPUs. The result was that the more CPUs the server has, the more chances this infinite loop would happen.

The root cause of the infinite loop is on an unprotected shared variable — a "`HashMap`" data structure. After added "`synchronized`" marker to all the access methods, everything was normal. (Sun engineers didn't think it a bug, but rather suggested the use of "`ConcurrentHashMap`". So take it into consideration when building a scalable system.)

Lecture X — Servlets and JSPs, WARs and EARs

JSP — Java Server Pages

When the presentation (user interface, or UI) code is for a thin client, you write a servlet to handle the HTTP methods, read the request data, and create a reply. That often means a lot of statements to produce HTML code.

Instead there is a simpler way. You could create an HTML page that contained various snippets of Java code, inside special tags. The web server processes such HTML pages by finding the Java bits and creating, then compiling a servlet for you. This special type of web server is called a *servlet container*. Apache Tomcat is a popular one. Many enterprise applications do not need the additional support of a full EJB container; Tomcat supports servlets and JSP pages, over HTTP or HTTPS, supports sessions automatically (with cookies), and the java used in the servlets/JSPs can use databases. So if you don't need transaction support, messaging, etc., then a servlet container is all you need.

An HTML page may contain various special (large) tags, which contain Java code. Each chunk of code is called a ***scriptlet***. When the JSP is first loaded, the page is parsed, the scriptlets are collected into a .java file, the `.java` file is compiled into a servlet `.class`, and thereafter using the JSP's URL actually returns the output of the compiled servlet. It is possible to configure tomcat to pre-load JSPs, to make this process seem faster to the user. Usually files that contain JSP use the extension "`.jsp`" rather than "`.htm`" or "`.html`".

JSPs often use JavaBeans ("beans") to actually do something, such as fetch data from a database. This is handy if some tasks is too complex to be mixed in with the HTML directly. You just create a JavaBean that has methods that do what you want, and invoke them from the JSP. A special XML tag is mixed in with the HTML to indicate the use of a JavaBean. A good JavaBean can be reused in other applications.

> The Java code and special tags in a JSP page is often considered part of the middle (business logic) tier and not the presentation layer (tier). However some could argue the other way around.

JSP supports a variety of other special XML tags that perform specific tasks. More recent versions of the JSP specification provide a way for users to define and use their own XML tags, which can do custom tasks. A collection of such tags is known as a ***taglib***.

Eventually the set of standard tags for JSP was greatly expanded and organized into 4-5 standard taglibs. This collection of taglibs is called the JSP Standard Tag Library, or JSTL. This will be discussed below.

In addition to the JSTL, the 2.0 version of JSP is now available with many new features that make using JSP a lot simpler.

Java Server Faces

JSPs are simpler than creating a servlet by hand, but still require you to manually type in the HTML you want. That means you need to know both HTML and related technologies such as JavaScript, as well as Java. A "higher level" view of the web application was desired, so users could build the UI by drag-and-drop components using a tool such as NetBeans. Java Server Faces (JSF) is a framework that supports this type of development. Using JSF, you can build the user interface without typing any HTML code! The result is an XML document that looks very much like a JSP page, except all the code and the HTML itself is handled by special tags. This XML document gets converted into a servlet (often it gets converted into a JSP first, which in turn becomes a servlet).

One nice feature of JSF is that the output need not be standard HTML. However that is all that is currently supported by this framework. But someday JSF could support mobile devices or pure Java (thick) clients, and the JSP page you wrote won't need to be changed! (Just a configuration setting someplace to say what kind of output to produce.) Also, JSF can be extended by adding new components (each with its own tag).

Another nice feature of JSF is that the resulting HTML can be customized by non-programmers; they only need to modify CSS style sheets.

> There are a number of tools that provide WYSIWYG GUI building tools for EE apps: JBoss Tools for Eclipse, GWT Designer (you can create cool UIs, but data-binding seems to be too complex), JSF (NetBeans no longer supports the visual web editor), ZK (supports easy data-binding and has got an Eclipse-based visual editor), Echo2, and Wicket (wicket.apache.org).

Creating JSPs

The basic HTML page "`basic.htm`" looks like this:

```
<html>
  <head>
   <title> Sample HTML Basic Page </title>
  </head>
  <body>
    <h1> Sample HTML Basic Page </h1>
    <p> This is a dull web page!
        I have nothing more to say.
    </p>
```

```
  </body>
</html>
```

(Discuss HTML and HTTP GET and POST). Such a page contains only static data. It is possible to jazz it up with animated images, Java applets, CSS, and JavaScript. However, these technologies are limited in what they can do, and all the processing is done on the client workstation (by the web browser).

To include dynamic content on the server side, you need the file to include special tags. The web server doesn't send the page directly to the web browser in this case. Instead it reads the file itself, ignoring HTML but looking for the special tags. The server than does something for each such tag; usually this involves running some servlet code and replacing the special tag with the output of some method call. After the document has been processed it then only contains HTML (and possible CSS, JavaScript, ...). It is then sent to the client.

The server uses the file extension to determine how to process the file. It might use PHP, ASP, JSP, or other special processors depending on the type of file. These all work the same way, looking for special non-HTML tags and replacing them with some other HTML. (Some documents today are not HTML at all, but a related technology of xHTML or even XML. Still the special processor doesn't care.)

The simplest JSP tag contains a legal Java expression. The tag will be replaced with the result of the expression evaluation. Here are a couple of examples:

A simple JSP page "`basic1.jsp`" looks like this:

```
<html>
  <head>
   <title> Sample JSP Basic Page </title>
  </head>
  <body>
    <h1> Sample HTML Basic Page </h1>
    <p> This is a dynamic web page!
        The current time
        is <%= new java.util.Date() %>. Hit
        refresh/reload to see the time update.
    </p>
  </body>
</html>
```

The "<%= *expression* %>" tag can be used for any valid Java expression, such as "2+2", "StockQuote.getPrice("IBM")", etc. The result is always converted to a string and inserted in place of the tag.

The special tags don't have to fit all on one line. They can appear anywhere, even inside of HTML tag attributes or inside quotes.

To deploy basic1.jsp, you must be using a JSP-aware web server, sometimes called a "JSP container" or "servlet container" or "Web Application Container". Apache Tomcat is often used for this. Your web content (including non-JSP files too) must be placed in a directory named for the web application, and placed within the server's "deployment" location, say "*server-install-directory*/webapps/MyApp/basic1.jsp". Then you start the server and use the proper URL to get there. For a locally installed Tomcat server with default settings, this would be "http://localhost:8080/MyApp/basic1.jsp".

Note a web application requires special files and sub-directories too. The whole set of files is often put into a single Jar file called a *web archive* or **WAR**. A WAR is standard, so any servlet container can use any WAR file. Most work like Tomcat, you simply have to place the WAR file in the correct directory.

Arbitrary Java can be placed in a similar tag. Here the result of running the Java code is not used to replace the tag. Instead you must send output to be inserted by writing to a pre-created PrintWriter called "**out**". Here's the same example using this type of tag:

The JSP page "basic2.jsp" looks like this:

```
<html>
  <head>
   <title> Sample JSP Basic Page </title>
  </head>
  <body>
    <% java.util.Date d = new java.util.Date();
       int sum = 2 + 2;
    %>
    <h1> Sample HTML Basic Page </h1>
    <p> This is a dynamic web page!
        The current time
        is <% out.print( d ); %>.
        The sum of 2 and 2 is <%= sum %>
    </p>
```

```
  </body>
</html>
```

Notice the sole difference: a missing "=". Also note how you can define variables in one scriptlet and use them in a later one. This works because **all the scriptlets are collected into a single servlet, in the order they appear**. This technique is exploited to easily generate HTML tables from Java arrays:

The JSP page "`table.jsp`" looks like this:

```
<html>
  <head>
   <title> Sample JSP table </title>
  </head>
  <body>
    <h1> Sample JSP table </h1>
    <table>
    <% for ( int i=1; i<=10; ++i) {  %>
    <tr><td> <%= i %> </td></tr>
    <% } %>
  </body>
</html>
```

when using control-flow statements, always use curly braces.

There are other special JSP tags too. A JSP comment looks like this:

```
<%-- The client never sees this --%>
```

(Note you can create a comment with "`<% /* comment */ %>`" too.) A regular HTML comment tag will be passed to the client with the page but not a JSP comment.

Although the ability to include arbitrary Java code mixed in with your HTML is good, too much code spoils the readability. Remember that JSPs are primarily web pages, and will likely need to be understood by Web designer and not Java programmers. Instead it is better to have complex Java code in a JavaBean, and then just use the proper JSP tag to use the bean. That makes the JSP page much simpler, especially for a non-Java programmer.

To invoke methods of your JavaBeans you use a set of special JSP tags:

```
<jsp:useBean id="user" class="PersonBean" />
```

which invokes the default constructor as if you had used:

```
PersonBean user = new PersonBean();
```

You can add the attribute "`scope`" to control the life of the bean object, like this:

```
<jsp:useBean id="cart" class="shoppingCartBean"
 scope="session" />
```

to have the cart object created the first time a user visits the page. That same object will be available each subsequent visit by the same user in the same session. The default is `page` scope, which creates a new instance each time the page is used. You can also specify `application` scope.

To set and get properties of your bean you use these special tags:

```
<jsp:setProperty name="user"
     property="name" value="Joe" />
<jsp:getProperty name="user" property="age" />
```

which invokes the setter and getter methods. The result of using `jsp:getProperty` is converted to a String and replaces the tag, much like the "`<%= expression %>`" tag. To make this work, you must compile and install the JavaBeans to the server's required directory, in tomcat this would be "`WEB-INF/classes/`", which is a subdirectory of wherever the JSP is installed (`webapps`). (I.e., it goes in the WAR.)

You can use a special tag to abandon processing the current page, and cause the JSP server to instead load a different page:

```
<jsp:forward page="URL" />
```

This type of tag is often used within if-statements:

```
<% if (...) { %>
    <jsp:forward ... />
 <% } %>
```

Pre-defined JSP Objects

In addition to **`out`** there are a number of pre-created objects you can use:

`request`	is a `HttpServletRequest`, used to access the data in the request. This can include useful information such as locale, time zone, browser type, user name, client workstation IP address URL information (such as the query string), etc. If the user clicked a "submit" button on some form to get to this page, you can also access the form field values with `request.getParameter("name")`.
`response`	is a `HttpServletResponse`, used to obtain an output stream out, and to set the response headers (especially the type).
`out`	is a `PrintWriter` obtained from the response.

`session` is a `HttpSession` associated with the request, used to store persistent data for this session only.

`application` is a `ServletContext`, which refers to the server itself and allows persistent data to be stored and retrieved by any client from any session.

`config` `ServletConfig`, used by the server to initialize servlets.

`pageContext` is a `PageContext`, that provides a few useful methods such as `getException()`.

`page` refers to the `Servlet` object, and is the same as using `this`.

Gotchas:

HTML is case-insensitive but XML, JSP, and Java are not. Since all JSP tags are really XML tags, the normal rules apply: All tags must end with "`/>`" or have a separate closing tag. All values must be quoted with either single or double quotes. If you use double quotes than any Java included inside must escape any double quote with a backslash.

Don't use a semicolon after expression, but don't forget them after statements!

Don't confuse the "`id`" attribute (used with `jsp:useBean`) and the "`name`" attribute (used with `jsp:getProperty` and `jsp:setProperty`).

Don't forget that your bean classes must really meet the JavaBean standard. If not, they will likely compile and run without errors or warnings. But they won't work!

Servlets

Part of Web Application, see WAR example on website and tomcat install directions (tomcat is reference implementation of the servlet API). Today you rarely need to write a servlet directly. Instead you use JSP. The Java code from the JSP page is collected and compiled automatically as a servlet. (So the first use of a JSP page may be slow.) One good use of a servlet is to control which page to display next. This would be confusing in a JSP. Here's a small sample:

```
public class PickPaymentMethod extends HttpServlet
{ protected void doPost ( HttpServletRequest req,
                          HttpServletResponse res )
   { ...
     String nextPage = "/process-" + paymentMethod + ".jsp";
     ServletContext sc = getServletContext();
     RequestDispatcher rd = sc.getRequestDispatcher( nextPage );
     rd.forward( req, res );
```

```
}
```

JSTL — JSP Standard Tag Library (and custom tags)

JSTL encapsulates, as simple XML tags, core functionality common to many JSP applications. For example, instead of suggesting that you iterate over lists using a scriptlet (servlet), or use different iteration syntax from numerous vendors, JSTL defines a standard `<forEach>` tag that works the same everywhere. Instead of writing JSP this way:

```
<% if
((User)session.getAttribute("user")).getName() .equals(
"Joe") { %>
```

you can use the simpler:

```
<c:if test="<%= user.name %> == 'Joe'">
```

This standardization lets you learn a single tag and use it on multiple JSP containers. Also, when tags are standard containers can recognize them and optimize their implementations.

The standard tag library contains 5 collections of tags. These are documented at java.sun.com/products/jsp/jstl/1.1/docs/tlddocs/index.html.

Each collection is called a ***taglib***. The tags all look like:

```
<taglibName:tagName attr1="val1" attr2="val2" ...>
    body  (optional)
</ taglibName:tagName>
```

If there is no body, just as with standard XML you can use the shortcut "<... />". If the value is the result of some JavaBean method call, or (as in the example below) the value of some previously defined Java variable, you use the standard JSP expression syntax to refer to it:

<someTags:aTag attribute="<%= pageContext.getAttribute("aName") %>">

In JSP 2.0 they added a special "expression language", that allows a simpler way:

```
<someTags:aTag attribute="${aName}">
```

Besides fetching data you can use simple math and boolean expressions:

```
<c:if test="${user.name == 'Joe'}">
```

zzz to be completed

In addition to the standard taglibs you can define custom taglibs. This is done by creating java classes with methods for each tag you define in your taglib. Then you create an XML file that tells the web container how to map tag names to classes and methods.

You declare that a JSP page will use tags defined in a tag library by including a taglib directive in the page before any custom tag is used:

```
<%@ taglib uri="/tlt" prefix="tlt" %>
```

For more information consult the Sun on-line tutorial at java.sun.com/products/jsp/tutorial/TagLibrariesTOC.html.

Deployment —Anatomy of a Jakarta EE Enterprise Application Archive (EAR)

Jakarta EE code today is mostly Jakarta EE *components* assembled into an application. It is hoped that some components can be reused as thus reduce the overall cost and time of development. A bundle of similar type of component is called a *module*. The ***EJB module*** will contain all the EJB components, the ***web module*** contains all the web server components (JSPs, servlets, and other web files). Additionally some enterprise applications may use thick clients and thus have an ***application module***. Finally some designs use standard interfaces to access EIS components. These interfaces (e.g., database specific JDBC drivers) form a ***resource access module***.

A given application typically has some but not all four modules. For example a very simple Jakarta EE application may only have a few servlets and HTML files, that directly use JDBC to talk to some database. In this case your whole application might only consist of the web module. More commonly you will have both that module and the EJB module.

WARs — Web Archive Files

The web module may consist of one or more *Web ARchive* files, or WARs. Each WAR is a Jar file that contains servlets, JSPs, and other web application components such as HTML files, graphics and other resource files. Sometimes your Jakarta EE application is small and simple, and the entire application can consist of a single WAR. Larger applications may consist of one or more WARs as part of the EAR.

WARs can be deployed in any Web Application server/container (sometimes called a *web container*, a *servlet container*, or *JSP container*), including the popular Tomcat, usually by simply dropping the WAR file into the appropriate place. (Depending on the server you may have to update it configuration, usually by editing an XML file.

Working with EARs

Each module contains the classes (and interfaces) for the components, plus a ***deployment descriptor*** that tells the container what components are there, what container services are needed, and how everything hooks together. This description is an XML file. It and the classes are packaged in a Jar file, which becomes the module. Note that the EJB and application modules use the regular

`.jar` extension, but the web module Jar file should be renamed "`.war`", and the resource access module extension should be ".`rar`".

The idea is you can list all Jars needed on the `CLASSPATH` in this file, plus any other options needed. Note for enterprise applications you often need to do other tasks such as initialize a database or define what should be included in the `web.xml` or `deploy.xml` files. You may need to specify config data such as JNDI data, `persistance.xml` (needed to use a DB), logging server config info, etc.

To play with the database, use the JMX console, click on "`database=localDB,service=Hypersonic`" link, then click on the "Invoke" button below the "`void startDatabaseManager() \n MBean Operation`" line. This starts up a database management console app.

The (up to four) modules are then packaged in a Jar with the extension "`.ear`", which forms the Jakarta EE application that can be dropped into the application server (sometimes called an EJB container or Jakarta EE container) for deployment.

EARs also include a *deployment descriptor* that tells a Jakarta EE compliant application server which services are needed and which classes implement required interfaces. The application server runs code to deploy the EAR components in the right places.

There are few books or resources to help you design a Jakarta EE enterprise application. Most books and on-line resources focus on one or more of the Jakarta EE technologies. There are a couple of case studies though, such as the Pet Store example from Sun, as well as some books from Sun (one of which can be read on-line).

A typical development process for Jakarta EE applications contains two steps:

1 Create JavaServer Pages (JSP), servlets, and other web components that handle the presentation logic of the application; and build Enterprise JavaBeans (EJB) components to manage the application's business logic.

2 Assemble those components into modules (with the appropriate deployment descriptors), and then combine those modules into an EAR that can be deployed to a Jakarta EE platform-compliant application server, such as Sun's Application Server PE 8.

The deployable unit can be either of the following:

- Jakarta EE module — A collection of Jakarta EE components that reside in the same container. The modules can be a Web archive (WAR), an EJB Java archive (Jar), a resource archive (RAR), or an application client Jar. Such

stand-alone modules are used in tiny applications. (Note RAR is not stand-alone.)

- Jakarta EE application — A collection of Jakarta EE modules, along with application deployment descriptors, in the form of an EAR file.

JBoss Setup

JBoss is a free Jakarta EE application server that supports all Jakarta EE 5 plus EJB 3.0. Visit `www.JBoss.com` or wildfly.org (for the new version). Red Hat has purchased JBoss. This product uses Tomcat for the Web app container part, so you automatically get that too. To develop software and to run your WARs and EARs you use the Jars of your application server. You don't need the Sun Jakarta EE SDK, but it may pay to bookmark the Javadoc APIs for Jakarta EE.

Download JBoss from `labs.jboss.com/portal/jbossas/download`

Just click on the "run installer" button. First pick "`Eng`" for English language installer. Click Next to see some version info and a link to the `JBoss.com` homepage. Click Next to see the Release notes. It might be a good idea to look those over.

Click Next to see the open source licenses used and to click the "I Agree..." button, then click Next.

Select the installation directory. The default of `C:\Program Files\jboss-4.0.5.GA` is fine. Then click Next.

On the next screen you pick which component sets to install. To use the current (new) EJB 3.0, you need to select the "`ejb3`" choice. You will see a warning that this requires Java SE 5, and that this is not J2SE 1.4 compliant. That's because the new EJB spec will be part of Jakarta EE 1.5. Note the "`all`" choice includes cluster-middleware support, but not EJB3.

The next screen is a list of "packs" (really Jars implementing various parts of Jakarta EE. By default, all are checked and that is fine.

On the next screen, you pick the name of the configuration to use. The default is "`default`", and if you change that you will have to supply the name on each command line, so don't change it.

The next screen allows you to select which database product to install and use. If you don't use the default choice ("Hypersonic SQL") you will need to install the various drivers and update the config files. I suggest you use the default.

The next screen "Enable deployment isolation/call by value" I don't understand, so I left the default of unchecked.

The next screen lets you chose to use secure console access (`https`) and to set the administrator name and password. I left the default (use security) and "`admin`" username and "`admin`" password.

The final screen shows your choices. Next the install begins. It didn't take long!

Now set some environment variables: `JAVA_HOME` to point to your JDK install directory (this may be set already) and `JBOSS_HOME`, to where you installed JBoss. On my system: `JAVA_HOME=C:\Java\jdk` and `JBOSS_HOME=C:\Program Files\jboss-4.0.5.GA`

Run the Application Server

To start the server run `%JBOSS_HOME%\bin\run.bat`.

Deploy EARs and WARs with JBoss

To deploy WARs and EARs, copy them into the directory `%JBOSS_HOME%\server\default\deploy`. (There is a bunch of files in there already.

To run your web app, point your browser to `http://localhost:8080/`*`nameOfWarFile`*. For example: `http://localhost:8080/myServletWAR`. This should show the `index.htm` page. To run the servlet, use the URL `http://localhost:8080/myServletWAR/hello`

(Note! The link in the `index.htm` file points to `.../myServletWAR/servlet/hello`, which doesn't work! The JSP page works though.)

Compiling Jakarta EE Code

To compile your code, you need to include the various Jar files with the Jakarta EE APIs for your Application Server, in this case all the Jar files in `%JBOSS_HOME\client\*.jar`. (A "meta" jar file that includes all Jakarta EE APIs can be downloaded from Maven Central.)

You could copy all these Jars to your JDK JRE's "`ext`" (extensions) directory, or list them on the javac cmd-line (with the "`-cp`" arg), but the common and simplest way is to use an Apache.org tool called "**`ant`**".

Lecture X — Enterprise JavaBeans (EJBs)

[A*dapted from "Mastering Enterprise JavaBeans" by Ed Roman et. al., Wiley pub.*]

A software *component* is some code (typically one or more classes, often in their own package) that implements a set of well-defined interfaces. It is sometimes possible to purchase software components that fulfill some need in your application, rather than write the software yourself. If you do end up writing the software yourself, you may be able to reuse it in another project someday. And even if not, writing your code to a set of interfaces is in general a very good idea.

An ***Enterprise JavaBean*** is a server-side component that can be deployed in a distributed multi-user environment.

Consider the pricing of goods and services you sell. There are base prices, quantity discounts, bundling discounts (buy both and save), preferred customer discounts, location discounts, taxes, and shipping and handling charges. If every part of your application (or of several applications) needed to compute prices, it makes a lot of sense to have a pricing component that can be used over and over.

When a client wants to use an EJB, it cannot invoke the EJB directly. This is because the EJB may not be running on that server but instead some JVM on another server. The client can only invoke a local object called a ***stub*** which acts as a proxy to the remote object and has the same (remote) interface. The stub is responsible for accepting method calls locally and delegating those method calls to the remote EJBs across the network. Stubs are running within the client JVM, and know how to look over the network for the real object through RMI/IIOP. The stubs can be created manually (and were in old Jakarta EE versions), but a simple JNDI lookup can find EJBs and create the stubs.

A *smart stub* (such as found in JBoss) contains the list of target instances it can access, can detect any failure of the target instances, and also contains complex load-balancing and fail-over logic to dispatch requests to the targets. Also if the cluster topology changes (for example: new instances are added or removed), the smart stub can update its target list to reflect the new topology without manual reconfiguration.

If the web/servlet container and the EJB container are on the same server, they can share a single JVM. This means the stub can avoid any network access and is thus noticeably more efficient. If your cluster uses this *collocation* design (rather than the standard *distributed* design) you can use an EJB's local interface rather than its remote interface.

Deployment Descriptors

To use software components you include them in an archive file (usually), placed in the application server's specific location that it uses to look for components, and then configure them, usually through a ***deployment descriptor***, which is an XML file that tells the server what the component is, what middleware services it needs, and how and when to deploy it. The deployment descriptor is a key piece of Jakarta EE. Prior to this each middleware vendor provided a proprietary way to pass this information for each component.

In some cases an additional step is needed. You may be to run a middleware-vendor-supplied tool to generate *client request interceptors*. These are similar to the stubs of RMI. The request interceptor takes the client request, performs any middleware actions to provide required security, transaction, and persistence support, and forwards the request to the component.

Today there is no real market for enterprise software components (but lookup crystal reports in the Internet).

Enterprise Java Beans

The ***Enterprise JavaBean*** (or ***EJB*** or ***Enterprise Bean***) standard provides a component architecture for deployable server-side components. An application server can import, load, and communicate with EJBs using this standard. The EJB standard is a 500+ page PDF you can obtain from `java.sun.com`. It also includes a set of interfaces that components and application servers must adhere to.

EJBs are designed to communication with other software: in the presentation layer, other EJBs, Web services, and EIS layer components.

For some projects this is overkill. If your application is just a GUI to a database, or is very simple, or you need the initial system built rapidly (say as a prototype), then the three-tiered approach is not appropriate. Instead just use a two-layer approach, having JSPs use simple Java classes to talk to the backend DB. However if you need to support transactions, sessions, banner ads, special services, communications with outside EIS or web services, or anticipate growth so scalability becomes very important, then EJB can be the way to go.

Types of EJBs

There are different types of beans, each used for a specific purpose. Not all types must be used in any given application. The most common are ***session beans***. These beans model business processes and workflows (such as a user session while shopping on-line). Session beans typically do something; when your description uses a verb, think session bean. Such beans might be either *stateful* or *stateless*.

Entity beans model data, such as back accounts, items, bids, purchase orders, etc. These components perform the *object/relational mapping* and form the bridge

between the business-logic layer and the EIS components. They are often referred to the *persistence* components.

Entity beans have been less commonly used because of poor performance and other problems. Many newer Jakarta EE servers provide better support for the newer EJB standard, which is supposed to address these problems. However many replacement technologies (not part of Jakarta EE, but supported by many application servers) have sprung up to solve this problem: Java Data Objects (JDOs), Hibernate, etc.

Entity beans may use either ***bean-managed*** or ***container-managed persistence***.

Message-driven beans are similar to session beans in that they perform some task, but the interface is via messages passed back and forth according to some protocol.

EJB Interfaces

The EJB specification defines standard interfaces that you bean classes can implement, depending on the type of bean it is and the services it requires/provides. All EJBs regardless of type, must implement **`jakarta.ejb.EnterpriseBean`** interface. This marker interface (doesn't require any methods to be implemented) makes the class `serializable`.

Different types of beans have more specialized interfaces, that extend this one. Session beans must implement `jakarta.ejb.SessionBean`, entity beans implement `jakarta.ejb.EntityBean`, and message-driven beans implement `jakarta.ejb.MessageDrivenBean`.

The clients access your bean's methods through the request interceptor. But how does the request interceptor know what methods of your bean classes to intercept? And how do clients know what bean methods are available? The answer is to define the interface to the methods your bean wants to expose. This is called the ***remote interface***. This interface must extend the **`jakarta.ejb.EJBObject`** interface. The methods defined in this interface are automatically implemented by the application server (a.k.a. the EJB container). You only need to implement the additional methods you add to your remote interface, implemented by your bean.

`jakarta.ejb.EJBObject` (and hence your remote interface) implement the `java.rmi.Remote` interface, making all your beans accessible via RMI (actually RMI-IIOP). All the normal RMI restrictions apply: pass-by-copy, must throw `RemoteException`, ...

Unlike RMI there is no central registry of JavaBeans. To get a reference to an EJB object, a client must obtain one from an *EJB Object Factory*, a part of the EJB container responsible for creating, locating, and destroying EJB objects as needed. The EJB specs calls this factory a ***home object***.

Home objects are specific to each vendor's EJB container, because they do container-specific tasks (such as load balancing). Home objects are automatically generated by the EJB container. However the home object needs information about your bean, such as the constructors to use and their arguments. You provide this information by providing an interface, called the bean's ***home interface***, that extends **`jakarta.ejb.EJBHome`**.

Although accessing beans via the remote and home interfaces is logically simple (relatively anyway), because such accesses must use RMI they are potentially very slow. In version 2.0 of the EJB standard, an alternative was provided for access on a single server. Clients can use a ***local interface*** (**`jakarta.ejb.EJBLocalObject`**) and the EJB container can use a ***local home interface*** (**`jakarta.ejb.EJBLocalHome`**). These interfaces cut out the RMI part and so are must faster. However, note that defining local interfaces is optional but you must still provide the remote and home interfaces, even if not used.

EJB Jar File

To deploy your bean, you bundle the bean classes, home and remote (and possibly local) interfaces, the deployment descriptor, and possibly vendor-specific extensions, all in a Jar file. Some IDEs will do this for you.

Lecture X — XML (and JSON)

The ***eXtensible Markup Language*** (**XML**) is a powerful way to describe hierarchically structured text information. Using XML means you can have your file formats well defined in a standard way, using ***DTD***s or ***Schema***s, which allows them to be automatically checked for validity. In addition, XML data can be easily transformed into other formats using ***XSLT***. You can create style sheets for your XML file formats that allow them to be nicely displayed by any web browser. Finally there are standard APIs (libraries) that implement a lot of the low-level details of reading, modifying, searching, and writing XML files (***SAX*** and ***DOM***).

> Java comes with a number of ways to work with XML. The Java API for XML Processing (JAXP) enables applications to parse, transform, validate, and query XML documents using an API that is independent of a particular XML processor implementation. It includes a number of APIs including DOM, SAX, StAX, XPath, and XSLT (although the implementation must support these).
>
> To use JAXP you must install some implementation of it (a Jar file found on CLASSPATH). The reference implementation only comes with Jakarta EE (not SE). But it is just the Apache XALAN-J XML implementation, and you can download that Jar file yourself to use with Java SE.

XML technology replaces custom file format parsers and is today used for SOAP, RSS, and many other standard formats. What's so great about this is that to use (say) RSS, you only need an RSS document, which includes a link to its definition and so can automatically (up to a point) be parsed. (We used to call such documents *self-describing*.) Using standardized document definitions makes it much easier to share documents!

An XML document should start with an XML declaration:

```
<?xml version ="1.0"?>
```

You can (and should) specify the document's encoding too:

```
<?xml version ="1.0" encoding="UTF-8"?>
```

And include an optional (from XML's point of view) DOCTYPE header:

```
<!DOCTYPE person ...>
```

This tells a read the type of the document (and optionally, where the definition for the document's structure can be found).

You can include comments anywhere:

```
<!-- anything not containing a double dash -->
```

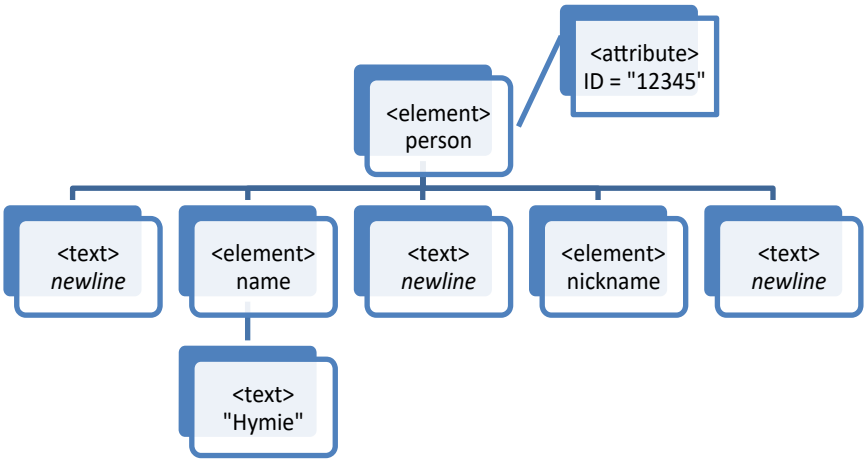
After these headers comes the document itself. An XML document forms a tree, similar to the structure of a filesystem on a disk. The topmost tag is the *root* node of the tree, and it contains other nodes.

An XML document uses *tags* that **always have an open and a close pair**. Tags can have required or optional *attributes* of the form *name="value"* (Note in XML the *value* must always be quoted.) The first tag defines the type of the document (and should match the DOCTYPE declaration if you provided one). In-between an open and close tag are other tag pairs and/or text.

If there is no enclosed data, then you can combine the start and end tags into a single *self-closing* tag. Here's an example:

```
<person id="12345">
  <name>Hymie</name>       tag data is: Hymie
  <nickname></nickname>   OR: <nickname/>
</person>
```

This XML document forms a graph that looks like this:



In this example, the *root* element (or *node*) is "person", which has 5 *children* nodes and one attribute node. The five children of person are *siblings*. One of these ("name") also has a child. Note siblings have an order. So you can refer to the first-child of a node, the next-sibling of a node, or a parent of a node. The tree can also be searched but functions that return a node or a list of nodes (e.g. all "name" nodes, or the "person" node with an "ID" attribute of "12345"). You can also add new nodes, modify nodes, or remove nodes from the tree.

> A (starting) tag is generally referred to as an *element*. Both elements and text are *nodes*. The document above has seven nodes but only three of them are elements. The distinction may be important; the second element is "name" but the second node is a text node.

Depending on the document definition, whitespace between tags may be permitted or not. If permitted they from text nodes; if optional they are called *ignorable* and may not be shown in the tree.

Some characters such as the left or right bracket (or double-quote) would be difficult to include as data. Instead you can use *Unicode character references* (in decimal or hex): `™` (hex) or `™` (decimal) represent the TM mark. A few characters have names (the ***character entity references***): `<`, `>`, `&`, and a few others.

> HTML documents are processed in exactly the same way, forming a tree of nodes known as the HTML DOM. JavaScript can be used to search this tree and modify it.

XML Namespaces

A single XML document may have some parts defined by one schema, and other parts by another. What if each part had a tag that used the same name as another? This would be a tag name conflict.

To avoid this, tag names can be given a prefix. You pick the prefix and associated it with some schema (DTD). Often a single letter prefix is used. For example:

```
<p:person id="12345">
  <p:name>Hymie</p:name>
  <p:nickname></p:nickname>
</p:person>
```

The namespace declaration has the following syntax: `xmlns:prefix="URI"`. It can be used with most tags, but is often used in the *root* (outermost) tag. For example:

```
<people xmlns:h="http://www.w3.org/TR/html4/"
        xmlns:p="http://wpollock.com/people.dtd">
  <h:h1>List of People</h:h1>
  <p:person id="12345">
    <p:name>Hymie</p:name>
    <p:nickname></p:nickname>
  </p:person>
</people>
```

The URI isn't actually used for anything except to provide a unique name. It is confusing since you expect to be able to type that URI in a web browser and see something. Sometimes organizations will put something there, a DTD or some web page, but this isn't required.

If one namespace is used more often than the rest, you can make it the *default namespace*. This is done by adding this attribute (again, often to the root tag): `xmlns="namespaceURI"`. (Note this is the same as before, except no prefix.)

> XML namespaces are commonly used for JSP (and JSF) documents, XSLT stylesheets, and xHTML.

XML Validation and Processing

XML documents that obey the above rules are said to be ***well-formed***. This is a simple check for a corrupted document. On the other hand, you can supply a definition for some type of document, with a URL in the DOCTYPE header, or by including the definition within special tags. Either way a *parser* can read the document and its definition, and verify the document. Such a document is called ***valid***. The definition can be either a ***DTD*** (*document type definition*) or a ***Schema***. Neither is suitable for all documents, so some will have a DTD, some a Schema, and some will have both. This is why the language is *extensible*.

The definition can include new character entity definitions, legal tags and attributes, the type of data that can be used for an attribute or tag data, required and optional tags and attributes, the order of tags, etc.

> Normalization of XML data is required for proper security. This should be done before validating the data. Failure to normalize and validate can lead to *XML injection* attacks or *XML External Entity* (XXE) attacks.

To process an XML you need to *parse* it. You can use any of several methods, (all part of "JAXP", Java APIs for XML Processing) including:

- **SAX** (simple API for XML), which reads the document once from the beginning to the end, invoking *call-backs* to your code when tags are noted. Use for very large documents that won't fit into RAM, or if you are only interested in a few bits of the document (imagine a bank's daily transaction log, and you are only looking for a particular customer's records). No tree is built; each bit of the document is read, call-back methods are called, then the next bit is read. (This API is considered difficult to use compared to the others.)
- **DOM** (document object model), which reads the whole document into RAM as a tree, that can be walked, searched, or modified and then saved back as a file. Naturally this requires a lot of memory and CPU power, but the familiar tree processing paradigm is easy to understand and use. Some documents won't fit into memory at once, so you can't use DOM in that case.
- **StAX** (Streaming API for XML) is designed to be easier than SAX but as powerful as DOM, without the memory and CPU resource requirements of DOM. (There's a good tutorial at java.net.) With StAX, you set up a reader (the same for SAX) and call the `next()` method. This reads the next thing

from the XML, and returns the type of thing (`COMMENT`, `ATTRIBUTE`, `ELEMENT`, `CHARACTERS`, ...) it found. (There are 15 of these.) You define event handlers for each, and invoke them in a switch as you read through the document. You can write XML the same way, one thing at a time.

This method of working with XML is called the *cursor* API. The "cursor" always points to the current part of the XML document you are working on, and you can move it forward but never backward. StAX also provides an *iterator* API,

> There are other APIs as well. Read the JAXP tutorials (for SAX, DOM, StAX, etc.) from Oracle.com for details.

There are a number of packages used for XML processing. See **`jakarta.xml`**, **`jakarta.xml.parsers`**, and other `jakarta.xml.*` packages. In addition the SAX processing classes you need are in **`org.xml.*`** packages while the DOM processing classes are in **`org.w3c.dom`** package (and a few others).

To use SAX, an instance of the `SAXParserFactory` class is used to generate an instance of an XML SAX parser. The parser wraps a `SAXReader` object. When the parser's `parse()` method is invoked, the reader invokes one of several callback methods implemented in the application. Those methods are defined by interfaces such as `ContentHandler`. `ContentHandler` has methods such as `startDocument`, `endDocument`, `startElement`, and `endElement`, which are invoked when an XML tag is recognized. This interface also defines methods such as `characters()`, which is invoked when the parser encounters the text in an XML element.

Here's an example adopted from the CERT Oracle Secure Coding Standard for Java. Suppose the XML data (after normalization) is:

```
<item>
  <description>item description</description>
  <price>500.00</price>
  <quantity>1</quantity>
</item>
```

In order to validate this, you need as *Document Type Definition* (DTD) or Schema. Here's a schema (`item.xsd`) that could be used to validate this XML:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="item">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="description" type="xs:string"/>
```

```
      <xs:element name="price" type="xs:decimal"/>
      <xs:element name="quanty" type="xs:integer"/>
     </xs:sequence>
   </xs:complexType>
 </xs:element>

</xs:schema>
```

The following SAX code uses the schema to validate the data (and prevent XML injection attacks), and a custom EntityResolver (to prevent XXE attacks), and is thus a best practice (but fairly complex; note a similar effort needs to be made for DOM and STaX):

```
import org.xml.sax.*;

class safeEntityResolver implements EntityResolver {
 public InputSource resolveEntity ( String publicId,
  String systemId ) throws SAXException, IOException {
  // Only allow know good external entity files:
  // Check systemId against a list:
  if ( systemId.equals( "safe_file" ) )
    return new InputSource( systemId );
  else
    return new InputSource();
 }
}

class SAXdemo {

// This method creates a valid, safe XML file from
// user input (of the quantity in this case) and
// some hard-coded XML:
void createXMLStream ( BufferedOutputStream outStream,
  String quantity ) throws IOException {
 String xmlData = "<item>\n"
  + "<description>item description</description>\n"
  + "<price>500.00</price>\n"
  + "<quantity>" + quantity + "</quantity>\n</item>";
 InputSource xmlStream = new InputSource(
   new StringReader( xmlData ) );

 // Create Schema object to use with SAX parser:
 SchemaFactory sf = SchemaFactory.newInstance(
  XMLConstants.W3C_XML_SCHEMA_NS_URI );
 DefaultHandler dh = new DefaultHandler() {
  public void warning ( SAXParseException spe )
    throws SAXParseException
```

```
    { throw spe; }
  public void error ( SAXParseException spe )
    throws SAXParseException
    { throw spe; }
  public void fatalError ( SAXParseException spe )
    throws SAXParseException
    { throw spe; }
 };
 StreamSource ss =
  new StreamSource( new File( "item.xsd" ) );
 // Construct a validating SAX parser:
 try {
  Schema schema = sf.newSchema( ss );
  SAXParserFactory spf =SAXParserFactory.newInstance();
  spf.setSchema( schema );
  SAXParser saxParser = spf.newSAXParser();
  // Use the custom EntityResolver:
  XMLReader xmlReader = saxParser.getXMLReader();
  reader.setEntityResolver( new SafeEntityResolver() );
  // Now read and validate the XML:
  saxParser.parse( xmlStream, dh );
 } catch ( ParserConfigurationException pce ) {
  throw new IOException( "Invalid XML", pce );
 } catch ( SAXException se ) {
  throw new IOException( "Invalid quantity", se );
 }
 // XML is valid, okay to write to file:
 outStream.write( xmlData.getBytes() );
 outStream.flush();
}
}
```

To build a DOM document in RAM (`org.w3c.dom.`**`Document`**) you use a **`DocumentBuilder`**. Class `Document` inherits from class `Node` which provides most of the interesting methods you want to use (show).

To allow for the future use of different types of DOM documents, Sun makes you get a `DocumentBuilder` object from a **`DocumentBuilderFactory`** class:

```
DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document doc = builder.parse(file);
                  // or inputStream or URL
Element root = doc.getDocumentElement();
String name = root.getTagName();
```

You use **`NodeList elements = root.getChildNodes`** to get the remaining elements (tags and data). (This was invented before Java Collections.) See on-line Demos for examples.

[*Adapted from a JavaWorld article, Pitfalls #4, by Michael Daconta*] A common problem with DOM is when a search for an element fails. All well-formed XML files have a tree structure. For example `myaddresses.xml` can be represented by a tree with two ADDRESS nodes:

```
<?xml version="1.0"?>
<!DOCTYPE ADDRESS_BOOK SYSTEM "abml.dtd">
<ADDRESS_BOOK>
  <ADDRESS>
    <NAME>Joe Jones </NAME>
    <STREET>4332 Sunny Hill Road</STREET>
    <CITY>Fairfax</CITY>
    <STATE>VA</STATE>
    <ZIP>21220</ZIP>
  </ADDRESS>
  <ADDRESS>
    <NAME>Sterling Software</NAME>
    <STREET> 7900 Sudley Road</STREET>
    <STREET> Suite 500</STREET>
    <CITY>Manassas</CITY>
    <STATE>VA </STATE>
    <ZIP>20109</ZIP>
  </ADDRESS>
</ADDRESS_BOOK>
```

Often, XML beginners wrongly assume that a DOM tree will look exactly like their mental image of the corresponding XML document. Let's say you must find the first `NAME` element of the first `ADDRESS`. By looking at the XML, you might think that the DOM's third node is the one you want. Unfortunately, the DOM tree includes text nodes for ignorable whitespace. Ignorable whitespace is whitespace that falls between tags, such as a newline or leading spaces or tabs.

One way to deal with this problem is to walk the DOM tree by fetching elements (the opening tags) rather than nodes (which are either elements or data):

```
Element root = doc.getDocumentElement();
NodeList = root.getElementsByTagName( "*" );
```

Another way is first to *normalize* your DOM by removing the ignorable while space nodes from the DOM tree. (Note there is another meaning to *normalize*; see `Document.normalizeDocument()`.) You must be careful to not remove all whitespace nodes, just the ignorable ones! Then you can process as normal. (Show sample code on-line that shows both ways.)

You can validate an XML document from a schema or from a DTD (DTD was the only method supported in older Java versions). Here's how:

```
final String s = XMLConstants.W3C_XML_SCHEMA_NS_URI;
SchemaFactory sf = SchemaFactory.newInstance(s);
StreamSource ss = new StreamSource("SomeSchema.xsd");
Schema schema = sf.newSchema(ss);

Validator v = schema.newValidator();
v.validate( new StreamSource(xml) );
```

A faster method of validation when using SAX is to replace the last two lines above with this:

```
SAXParserFactory spf = SAXParserFactory.newInstance();
spf.setSchema( schema );
SAXParser parser = spf.newSAXParser();
parser.parse( XML_DOCUMENT );
```

To use StAX to read an XML document, you can use this (with the cursor API):

```
import jakarta.xml.stream.*;
import static jakarta.xml.stream.XMLStreamReader.*;
XMLInputFactory f = XMLInputFactory.newInstance();
XMLStreamReader r = f.createXMLStreamReader( ... );
while(r.hasNext()) {
    int type = r.next();
    switch ( type ) {
      case START_ELEMENT: ...; break;
      case ATTRIBUTE: ...; break;
      case CHARACTERS: ...; break;
      ...
    }
}
```

While more efficient, the cursor API is not as flexible as the similar iterator API. The `XMLEventReader` (iterator) API in StAX provides the means to map *events* in an XML stream to event objects that can be freely reused, and the API itself can be extended to handle custom events. The event objects are immutable, simplifying they use in collections or with multi-threading. It implements the standard `java.util.Iterator` interface as well. Here's the reader code:

```
import jakarta.xml.stream.*;
import static
jakarta.xml.stream.XMLStreamConstants.*;
XMLInputFactory f = XMLInputFactory.newInstance();
XMLEventReader r = factory.createXMLEventReader
    (filename, new FileInputStream(filename));
while (r.hasNext()) {
    XMLEvent e = r.nextEvent();
    switch (e.getEventType() ) {
      case START_ELEMENT: ...; break;
      ...
    }
}
```

Java 5 and Java 6 include many enhancements to Java's XML capabilities and performance. Java 6 includes the *Java XML Digital Signature* API. This API allows you to generate and validate XML digital signatures. XML signatures are a standard for digital signatures in the XML data format, and they allow you to authenticate and protect the integrity of data in XML and web service transactions. (See dig_signature_api Sun article.)

In addition to these three low-level, primitive APIs (all part of JAXP), there is also JAXB. **JAXB** (*Java Architecture for XML Binding*) is the stuff under `jakarta.xml.bind` that uses annotations to bind XML documents to a Java object model. With JAXB, you attach (*bind*) the XML schema to an XML document, and the system automatically will parse the document into a collection of objects your program can use. To use JAXB, you create XSD (XML Schema files) to describe your data. Then use the `xjc` tool to generate Java classes with JAXB annotations. You can then use those classes similarly (in some ways) to using JPA classes: use them to read XML data into a *content tree* (similar to DOM) objects. (This can optionally validate the data as well.) Now you use the generated classes to modify the content tree, add to it, or just to examine it. After modification, you use the same classes to save the content tree as XML files.

> JAXB, along with some other Jakarta EE stuff, was added to Java SE every so often. However, starting with Java 9, Oracle stripped all that out of Java

SE. It is still possible to use jaxb in Java SE 9+, if you include the right Jar files on your classpath. See the JAXB demo for details.

JSON — JavaScript Object Notation

JSON is a syntax for storing and exchanging text information, much like XML. But JSON files are smaller than XML ones, and they can be faster to create or parse (read) than XML.

While JSON uses JavaScript syntax for describing data objects, it is language and platform independent. JSON parsers and JSON libraries exists for many different programming languages, making it a good choice for file or message formats. The details can be found at JSON.org, and in RFC-4627.

JSON is one of two things: a collection of name-value pairs, or a list of values. The values can be anything, including other collections or lists. A value can be a string in double quotes, or a number, or `true` or `false` or `null`, or an object or an array. Collections are enclosed in curly braces, using a colon between the name and value, lists in square braces, and items in a collection or list are separated with commas. You can have arbitrary whitespace between any tokens, to make the results human-readable. That's it! Here's an example (from Adobe Labs):

```
{ "id": "0001", "type": "donut", "name": "Cake",
 "ppu": 0.55, "batters": {
   "batter":
   [{ "id": "1001", "type": "Regular" },
    { "id": "1002", "type": "Chocolate" },
    { "id": "1003", "type": "Blueberry" },
    { "id": "1004", "type": "Devil's Food" }
   ] },
 "topping":[
   { "id": "5001", "type": "None" },
   { "id": "5002", "type": "Glazed" },
   { "id": "5005", "type": "Sugar" },
   { "id": "5007", "type": "Powdered Sugar" },
   { "id": "5006",
     "type": "Chocolate with Sprinkles" },
   { "id": "5003", "type": "Chocolate" },
   { "id": "5004", "type": "Maple" }
  ]
}
```

(Note, this example includes inconsistent whitespace.)

Currently, only Jakarta EE has built-in support for JSON (See JAXB). However, it was voted to add JSON to Java SE in 1/2012 (JSR 353); look for it in Java 8.

(Update: JAXB is supported in Java 8 SE, but in Java SE there is no JSON support and you need a third-party Jar file that implements JAXB. (Without that, you can compile but not run Java SE code that uses JAXB.) EclipseLink not only implements JAXB but also JPA and other persistence APIs. It also includes several other utilities and features. (Only Jakarta EE JAXB supports JSON, in the package `jakarta.json`.) See the Java Setup resource for info on obtaining EclipseLink.)

For Java SE, you need to download one of the many free JSON libraries available to use JSON easily now. One of the most popular is called **JSON-lib**, available at json-lib.sourceforge.net. To use, download the Jar file for this and add it to your CLASSPATH. However, you also need several additional Jar files, available from Apache.org.

A simpler, stand-alone library is **`org.json`**, available as a zip of source files from github.com/douglascrockford/JSON-java. I have compiled these and added them to a Jar file; just download it into your "JavaExtensions" directory that you listed on CLASSPATH. The Java docs for `org.json` are available at json.org/java.

Using org.json

To construct new JSON objects, use `JSONArray` to construct JSON lists, and `JSONObject` to construct JSON collections. The JSON arrays and objects have useful `toString` methods, making it easy to write JSON to files. These classes can also read in ("parse") JSON and build `JSONObjects`, which can then be easily changed or converted to regular Java objects. Finally, this library supports converting JSON to and from XML. (***Show JSONdemo.java***)

Lecture X — Using Apache Ant

A problem with deploying complex code is automating the process. Consider just a simple Java application with Java docs, that (when built) must be put into a Jar and uploaded to a website. In the very old days, that sort of task was handled by crafting a shell script or DOS batch file that could recompile the code, run tests on it, package it up, generate the documentation, and copy everything to the correct servers. Later (in the "merely" old days) a Unix tool "`make`" was invented that allowed you to do all that, by specifying the "targets" to build and what each depends on, so "`make all`" could be all you type to compile hundreds of files.

Although popular and ported to Windows, `make` is not without its problems. For one thing, each source file in C (what `make` was created to handle) would compile into a single "object" file. `Make` can examine the date of the source and object files and only rebuild the minimum number of files. But for Java, a single source file may compile into multiple .class files. With `make` the various building instructions are put into the `makefile` as a script. (Only the most basic building tasks are built-in). However that ties you to a platform that supports that scripting language (shell or DOS).

A replacement for scripts and `makefiles` is Apache Ant. Ant is a Java based, command line build tool. You define what to build, when and how to build and deploy it, using an XML file (by default called **`build.xml`**).

Today (2020), ant is considered obsolete. Instead use Maven or Gradle.

Ant comes with many pre-defined tasks for you (and you can define new ones), such as tasks to build a WAR or EAR file automatically. The Ant tasks are actually Java classes.

Using XML for the build file means most errors are caught automatically, and the build file (unlike with make) is not tied to a particular platform. Ant is common and supported by Eclipse, NetBeans, and other IDEs.

Ant reads a file **`build.xml`** (or a different file you specify on the command line) to determine exactly what do to for each task. You can specify the task to do or else ant does the task you marked as the default one. (So most of the time all you do is type "ant".) You create a `build.xml` file (or find one to use as a model on the Internet), put it with your source, then edit your code and to compile it use "`ant` *`task`*". You define the task names, such as "`compile`", "`test`", "`package`", "`deploy`", "`generate-javadocs`", "`clean`", etc., and list a default task to use if you just type "`ant`". Also tasks can be dependent on one another, so you can have "`package`" depend on "`compile`". (See

AntBestPractices from O'Reilly.com, wikibooks.org/wiki/Apache_Ant, and other Ant demos and resources.)

Here's a slightly complex example. Suppose a project puts all the source code under .../src, and the resulting class, Jar, WAR, EAR, etc., files in .../build. The package name for classes is com.example.hello. The build.xml file might look like this:

```
<project name="HelloWorld" basedir="." default="main">
 <property name="src.dir" value="src"/>
 <property name="build.dir" value="build"/>
 <property name="classes.dir"
           value="${build.dir}/classes"/>
 <property name="jar.dir" value="${build.dir}/jar"/>
 <property name="main-class"
           value="com.example.hello.HelloWorld"/>
 <target name="clean">
  <delete dir="${build.dir}"/>
 </target>
 <target name="compile">
  <mkdir dir="${classes.dir}"/>
  <javac srcdir="${src.dir}" destdir="${classes.dir}"/>
 </target>
 <target name="jar" depends="compile">
  <mkdir dir="${jar.dir}"/>
  <jar destfile="${jar.dir}/${ant.project.name}.jar"
       basedir="${classes.dir}">
   <manifest>
    <attribute name="Main-Class" value="${main-
class}"/>
   </manifest>
  </jar>
 </target>
 <target name="run" depends="jar">
  <java jar="${jar.dir}/${ant.project.name}.jar"
        fork="true"/>
 </target>
 <target name="clean-build" depends="clean,jar"/>
 <target name="main" depends="clean,run"/>
 <target name="junit" depends="jar">
  <junit printsummary="yes">
   <classpath>
    <path refid="classpath"/>
    <path refid="application"/>
```

```
    </classpath>
    <batchtest fork="yes">
     <fileset dir="${src.dir}" includes="*Test.java"/>
    </batchtest>
   </junit>
  </target>
</project>
```

Now you can build with "`ant`", run with "`ant run`" and test with "`ant junit`"! You can also use ant to build deployment descriptors and create WARs and EARs, and even deploy them.

(*Demo installing ant, then create a simple build.xml for WordCount project model solution (to create a clickable Jar and have some JUnit tests), and run it.*)

Lecture X — Credit Card Processing

See on-line Resource.

Lecture X — Networking

Host to Internet (cloud) to Host (def ***host***): IP and IP addresses and DNS names (show picture, leave space inside host box for program icons + port numbers). Note IP addr.s are assigned to NICs, not hosts. A NIC may have many addresses. ***Peer to Peer*** versus ***Client Server***. Program to program: **port numbers**. Server numbers usually <1024, ***well known port numbers***. Some other servers use <49151, ***registered port numbers***. Numbers >49151 through 65535 are *dynamic port numbers*, used by clients.

telnet ***host port*** (chose option to not close window after connection closes).

TCP (connection, data exchange, hang-up) vs. **UDP** (data exchange). Note initial delay for TCP often offset by easier reliable programming. Qu: when is UDP useful? Ans: small single packets. DNS queries? (UDP) HTTP? TCP (note after 1st page, IMG, Applets, etc. are sent as well. Also, request may be long (and thus several packets).

`Java.net` includes many useful classes, such as `URL`, `INetAddress` (do DNS queries, get local IP addr, ...), supports HTTP directly.

(Show `ShowIPAddresses.java`.)

Sockets

`Socket`: an object representing a TCP communication channel over the network. For UDP, use **`DatagramSocket`** instead. To connect to a server from a client:

```
Socket s = new Socket( "host", port );
InputStream in = s.getInputStream();
OutputStream out = s.getOutputStream();
...
s.close();
```

On the server side, you need to open a socket *before* there is any client at the other end. You only need to say what *port* to listen on:

```
ServerSocket ss = new ServerSocket( port );
Socket incoming = ss.accept(); // Will wait for client
incoming.setSoTimeout( 1000 );
Scanner in = new Scanner(
    incoming.getInputStream () );
PrintWriter out = new PrintWriter(
    incoming.getOutputStream(), true /*autoflush*/ );
String s = in.nextLine();
out.println( "Type 'q' to exit" );
```

```
...
incoming.close();
```

A problem with the above code is that you need to find an unused port number. Java can do that automatically if you use zero for the port number (it will tell you the actual port number used in that case). Here's some code:

ServerSocket s = new ServerSocket(0);
System.out.println("listening on port: " + s.getLocalPort());

If you want to use a specific set of ports, then the easiest way is probably to iterate through them until one works. Something like this (from StackOverflow):

```
public ServerSocket create(int[] ports) throws
IOException {
    for (int port : ports) {
        try {
            return new ServerSocket(port);
        } catch (IOException ex) {
            continue; // try next port
        }
    }
    // if we get here, no port in the range was found:
    throw new IOException("no free port found");
}
```

Note it isn't safe to simply scan for an available port, by attempting to open, then closing a `Socket`, and returning the port number. ("Race condition"). However, some libraries don't accept a `Socket`, only a port number, so you may have no choice.

Often the server spawns a new Thread to handle the client's request. The socket is passed to a new "worker" thread and the main server goes back to waiting for the next client.

Demo `ChatServ` via telnet. *Note custom protocol. A good protocol should include a length (of packet or at least of the header), the type of the packet, grouped bit flags aligned on byte boundaries, and a protocol version number. Also, think about extensions in the future.*

Using Sockets is a low-level way to do networking, since you must implement your own client-server protocol. It can be much easier to adapt an existing protocol for your use. Often HTTP is used for that and Java offers a lot of support for HTTP. **Other options are to use RMI or Jakarta EE**, which handle all the messy networking details for you. Today it is fairly common to send data between

clients and servers (or peer to peer) using XML for the data and HTTP for the protocol.

Java objects can be *serialized*, which makes a deep copy of an object and stores it in a form suitable for loading into a different JVM (usually on a different host). The classloader at the other end converts the object back into a standard form and loads it. This process is tricky and time-consuming, which accounts for the popularity of RMI, XML, HTTP, and Java Messaging. (See `DeepCopy.java` demo.)

Service Queuing

Networking needs to be understood by Jakarta EE developers. Often, users fill out forms and click links on web servers that send requests to the application servers (sometimes, via a load balancer). Hopefully you have deployed sufficient servers to handle the load. However, requests typically have "bursty" behavior: in a short time, a batch of requests can arrive faster than your server can handle them.

The standard solution is to have arriving requests appended to the end of a "FIFO" queue. The application server(s) wait if the queue is empty, otherwise take one request and process it.

What if so many requests come in that the queue gets full? Some requests must be ignored. But the FIFO behavior doesn't scale up well, as older requests have often been retried already, or the client may have given up waiting. The solution is two-fold: have requests in the queue have a time-out parameter, after which they are ignored. Also, switch the queue to "LIFO" until it drains. Clever algorithms are adaptive in adjusting the behavior (pseudo-code):

```
myQueue.add( newRequest ) {
   now = currentTime();
   // Generate log data

   // Set request timeout value:
   if ( this.lastEmptyTime() < now - 100ms )
      timeout = 5ms;
   else
      timeout = 100ms;

   this.enqueue( newRequest, timeout );

   // Set queue mode:
   if ( this.size() > maxThreshhold )
      this.mode = LIFO;

   else if (mode == LIFO && this.size() < minThreshold)
      this.mode = FIFO;
```

```
    // Generate performance metrics (ie time in method)
}
```

As you can see, enterprise coding requires thinking about performance at scale, and using load-balancing clusters of servers to allow one or more to fail. New requests must be logged, and time spent in a method must be available in some way (performance metrics), as well as queue size and state.

Lecture X — RMI and Distributed Objects

In traditional client-server (and even peer-to-peer) networking, the application pieces (let's call them "A" and "B") communicate by passing messages back and forth. However, such low-level code is difficult to get right and maintain. It may contain tricky security problems too. Finally, what if you deploy both A and B on the same server? Then for efficiency you would need to re-write both pieces to use simple method calls.

> L. Peter Deutsch of Sun Labs is known for his list of Eight Fallacies of Distributed Computing:
>
> 1. The network is reliable
> 2. Latency is zero
> 3. Bandwidth is infinite
> 4. The network is secure
> 5. Topology doesn't change
> 6. There is one administrator
> 7. Transport cost is zero
> 8. The network is homogeneous
>
> Deutsch noted "All prove to be false in the long run and all cause big trouble and painful learning experiences."

One solution is ***Remote Method Invocation***, or RMI. In this scheme, A and B don't care if they are on the same platform or not. They communicate via normal method calls. If you deploy one piece on another server, you create a ***proxy*** class on A's host, known as a ***stub***. A client makes normal method calls to this proxy. This proxy class "magically" uses networking (actually a proxy server is started on localhost, as well as a name server for RMI use) to serialize objects and pass them across the network to a similar proxy on B's host. This receiving proxy is called the ***skeleton***. That class re-constitutes the objects and invokes B's methods normally. Any return value is sent back using the two proxies.

RMI handle's the "magic" for you, so RMI is easy to use. However it isn't your only choice for distributed object passing. You can also use ***CORBA*** (*Common Object Request Broker Architecture*), which is bulky, complex, and rarely used today, and ***SOAP*** (*Simple Object Access Protocol*), which uses XML. SOAP is evolving into something as complex as CORBA.

RMI is designed for calls between Java distributed objects only, while CORBA and SOAP are language-neutral, allowing you to mix calls between, say Java and .NET objects.

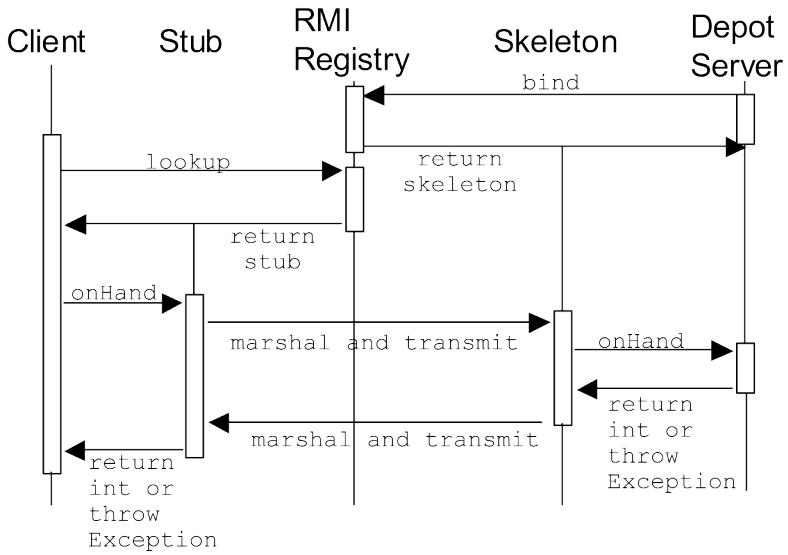
RMI-IIOP (***Remote Method Invocation over the Internet Inter-ORB Protocol***) is a CORBA compatible version of Java RMI, that is used for Jakarta EE (So you can talk to non-Java distributed objects too.)

Microsoft developed COM for object communications, and later DCOM, but DCOM failed, and today MS is pushing SOAP rather than COM.

No matter which solution you chose, you need to write the interface descriptions for the methods and parameters you chose to make available to other distributed objects. This is done using *IDL* (*Interface Description Language*) for CORBA and *WSDL* (*Web Services Description Language*) for SOAP. For RMI you create regular interfaces.

In RMI, if A invokes a method of remote object B, than A is the ***client*** and B is the ***server***. For example suppose a user clicks on a link in a browser to find the current inventory on-hand for some item. The item is represented by a class `Item`, the warehouse by a class `Depot`, and the client by class `Client`. `Depot` has a method `onHand` that takes an `Item` and returns an `int`.

The client invokes the find method on the stub (the client-side proxy class), passing it the Item object in question. The stub must serialize the parameters, a process known as ***marshalling***. The stub then contacts the skeleton (the server-side proxy class), which un-marshals the parameters, locates the correct `Depot` object, and calls its `onHand` method. The return value is marshaled and returned to the stub, which un-marshals the value and returns it to the client.



RMI does have a few complexities. The class object must be available on both ends, and there can be problems with exceptions or with locating the correct object to invoke. You will need to preform extra steps when deplopying and running your RMI code, sich as setting up the proxy servers and the RMI name service ("rmi registry").

All distributed object solutions use a network, but this is usually the slowest and least reliable part of the whole system. Carelessly designed systems end up making lots of RMI (and hence network) calls. Such a design is sometimes called *fined-grained*. A *coarse-grained* solution makes fewer network calls, passing more information (with larger objects) with each call.

Another problem is that a serialized object is a *copy* of the original. When the remote code invokes methods on the passed object, any changes only affect that copy, not the original. This *pass by copy* of objects is different than the normal way Java passes objects to methods (by passing a reference to the original object). (Note: RMI-IIOP supports both pass-by-copy and pass-by-reference, which of course takes longer.) This can sometimes be a problem with `equals` and `hashCode` methods.

Naming classes becomes difficult when using proxies. A standard naming convention can help here. Horstmann and Cornell recommend (p. 266 v2):

No special suffix (e.g., Item)	A remote interface
Impl suffix (e.g., ItemImpl)	The server class implementing that interface
Server suffix (e.g., ItemServer)	A server *factory* class
Client suffix (e.g., CustClient)	A client that calls remote methods
_Stub suffix (e.g., Item_Stub)	A stub proxy, not needed in Java5
_Skel suffix (e.g., Item_Skel)	A skeleton proxy, not needed in Java5

Qu: How does your client locate remote objects to talk to? Such objects are not listening for incoming TCP/IP requests! Ans: Pre Java5, the stub was manually configured to use a specific IP address and port number for the skeleton (which also needed that info.) Today you use a lookup service (a registry) to dynamically get a reference to the remote object. The reference is the stub!

Applications can use one of two mechanisms to obtain references to remote objects. An application can register its remote objects with RMI's simple naming facility, **`rmiregistry`**. This is called ***binding*** (a name with an object). Or the application can pass and return remote object references as part of its normal operation.

In Jakarta EE, you could use **JNDI** (***Java Naming and Directory Interface***) rather than use a separate RMI registry, however for non-Jakarta EE code avoiding JNDI is simpler. Names of remote objects are special URIs that can look like "**`rmi://host:port/name`**". (See www.softwarematters.org/jini-intro.html.)

The **`rmiregistry`** uses **`java.rmi.Naming`** class to support operations such as `bind`, `rebind` (same thing be replaces previous binding of that name, if any), `unbind`, and for the clients, `list` and `lookup`.

Prior to Java 5, after compiling the client and server code, you used a special tool to generate the stubs and skeletons: **`rmic`**. You first compiled the client code, ran this tool to generate the stub. In Java 1.1 this tool also generated the skeleton for you to

upload to the server host. In Java 1.2 the skeleton was handled automatically when a server object registered itself with a manually entered URI (host and port numbers). As of java 5 both stubs and skeleton generation is automatic, no more `rmic`!

In addition to the above problems, sending objects to a remote application, that will blindly accept them and invoke their methods, is not a secure design. So all distributed object solutions require a security manager be used to restrict what an untrusted object can do. This is the same as for an Applet (which already has installed a `SecurityManager`, so don't try to do it again in an applet).

RMI Steps

1 Create the interface that extends **`java.rmi.Remote`** for the server-side class, needed by both ends:

```
import java.rmi.*;
public interface Depot extends Remote
{ public int onHand ( Item i ) throws RemoteException;}
```

2 Create the server implementation class, having it implement the interface and extend **`UnicastRemoteObject`**. That's the easy way! The hard way is to not have it extend this, and instead manually export (register) the class so remote objects can find it:

```
UnicastRemoteObject.exportObject(this, 0); )
```

Note the no-arg constructor (which you need) also may throw a **`RemoteException`**, which you should handle.

3 Start the server-side code, and have it *register* or ***bind*** with the registry to be used. Registration in this case allows remote clients to use TCP/IP to contact the server-side *RMI registry* and find the server object. The following example of `ChatServerImpl` shows setup of the required security manager, creating an instance of the server, and registering it (uses port 1099 and host `localhost` by default):

```
import jakarta.naming.*;
import java.rmi.*;
import java.rmi.server.*;
class DepotImpl extends UnicastRemoteObject
                implements Depot
{ public static void main(String [] args ) {
   System.setSecurityManager(new RMISecurityManager());
   try {
    Depot depot = new DepotImpl();
    Context namingContext = new InitialContext();
    namingContext.bind( "rmi:depot", depot );
    System.out.println( "Server Ready." );
   } catch ( RemoteException e ) {
     System.out.println("General Server Error: " + e);
     System.exit( 1 );
  }}
  public int onHand ( Item item ) throws RemoteException
  { ...; return num; }
}
```

The server is ready, but the registry must be started before any client can talk to it. The simple Java tool is called **`rmiregistry`**. (For Jakarta EE, use JNDI.) You start it in Unix/Linux as "`rmiregistry&`", and on windows as "`start rmiregistry`". Add an optional port number argument if you don't want to use the default of 1099. (With some code hacks to detect your OS, you can do this from `main`.)

4 Now you need to create the client side. Since this code will call methods of the dynamically created stub, security is a concern here too. The client should start a security manager. Once this is done, you connect to the remote object registry using a special URI (URL). This will create and return the stub you need to access the remote object. Here is an example:

```
import jakarta.naming.*;
import java.rmi.*;
import java.rmi.server.*;
public static void main( String [] args )
{ System.setProperty( "java.security.policy",
     "client.policy" );
  System.setSecurityManager(new RMISecurityManager());
  try {
    Context namingContext = new InitialContext();
    Depot depot_Stub =
      namingContext.lookup( "rmi://host/depot" );
    ... Item item = ...;
    int num = depot_Stub.onHand( item );
  } catch ( RemoteException e ) {
    System.out.println("General Server Error: " + e);
    System.exit( 1 );
} }
```

The client is ready, but it must make network connections to reach the registry on the server host. Having installed a `SecurityManager` on the client, by default this is not allowed. (There's no problem on the server because the registry and the server code are on the same host.) This is done by creating the policy file and supplying the file's name in the `System.setPolicy()` method call, prior to loading in the `SecurityManager`. (Or, you can use the following on the `java` command line: "`-Djava.security.policy=client.policy`".)

***file* `client.policy`**

```
grant { permission java.net.SocketPermission
        "*:1024-65535", "connect";  };
```

Finally, make sure you don't have a firewall blocking your connections!

(For more info, check out the RMI FAQ and the RMI trail of the Java tutorial, from Oracle.)

> RMI is used but a pain. Newer systems can use a better design, message passing. ProcessA sends data to ProcessB, which may be across an internet. ProcessA can chose various networking options (such as synchronous or asynchronous delivery, timeouts, delivery notification, etc.) In many enterprise systems, this is handled via something called a ***message broker***, essentially a postal service but for data and not birthday cards. A very commonly used message broker (2017) is RabbitMQ.

Lecture X — Security

Policy Files and `policytool`

There is by default a single system-wide policy file, and a single user policy file. The system policy file is by default located at `java.home/lib/security/java.policy` (Linux/Unix) and `java.home\lib\security\java.policy` (Windows). Note: `java.home` refers to the value of the system property named "`java.home`", which specifies the directory that houses the runtime environment — either the JRE directory in the JDK or the top-level directory of the public JRE.

The system policy file is meant to grant system-wide code permissions. The `java.policy` file installed with the JDK grants all permissions to standard extensions, allows anyone to listen on un-privileged ports, and allows any code to read certain "standard" properties that are not security-sensitive, such as the `os.name` and `file.separator` properties.

The user policy file is by default located at `user.home/.java.policy` (Linux/Unix) and `user.home\.java.policy` (Windows). Note: `user.home` refers to the value of the system property named "`user.home`", which specifies the user's home directory. On Windows systems, given user name *uName*, the `user.home` property value defaults to:

C:\Windows\Profiles\uName on multi-user Windows 95 systems
C:\Windows on single-user Windows 95 systems
C:\Winnt\Profiles\uName on multi-user Windows NT systems
C:\Documents and Settings\uName on Windows XP systems
C:\Users\uName on Vista and Windows 7

When the policy is initialized, the system policy is loaded in first and then the user policy is added to it. If neither policy is present, a built-in policy is used. This built-in policy is the same as the original sandbox policy.

Policy files are plain text files with a syntax defined at java.sun.com/javase/6/docs/technotes/guides/security/PolicyFiles.html.

Policy file locations are specified in the security properties file, which is located at `java.home/lib/security/java.security` (Linux/Unix) and `java.home\lib\security\java.security` (Windows).

The policy file locations are specified as the values of properties whose names are of the form "`policy.url.`*n*" where *n* is a number. You specify each such property value in a line of the following form:

`policy.url.`*n*`=`*URL*

For example, the default system and user policy files are defined in the security properties file as:

```
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy
```

You can specify a number of URLs (including ones of the form "http://"), and all the designated policy files will get loaded. You can also comment out or change the second one to disable reading the default user policy file. The algorithm starts at policy.url.1, and keeps incrementing until it does not find a URL. So if you have policy.url.1 and policy.url.3 (i.e., no "2"), policy.url.3 will never be read.

Lecture X — JavaBeans

A JavaBean, or more commonly just "***bean***", **is a reusable software component, based on Sun's JavaBeans specification, and that can be manipulated visually in a builder tool**. Such a component may be a single class, or several related classes packaged in a Jar file. Most AWT and swing components are valid JavaBeans.

The idea is to develop classes that meet this specification and are thus beans. Then you can use visual editing tools of Eclipse, NetBeans, JBuilder, etc., to assemble the beans into a GUI application. For example, you might be able to drag and drop a bean from a list onto an (initially empty) Frame or Applet. Then perhaps right-click on a bean to adjust all its properties.

Such a system makes developing GUI applications much easier. So the idea is that you can develop beans and use them with a JavaBean compliant GUI IDE.

JavaBeans have nothing to do with "Enterprise JavaBeans", or "EJBs", despite the name. Sun liked the use of "bean" to mean a software component, but EJBs are not GUI components. However Java Server Pages (JSP) does make extensive use of JavaBeans.

JavaBean Standard (From Java 1)

Some classes define *getter* and *setter* (*accessor* and *mutator*) methods, for each of the object's properties. In general, this is a bad idea. Fields are only there to support the methods, and as an implementation issue are usually declared `private`. Sometimes, however, it is a good idea. For example, Graphics.setXXX methods.

> Too many such methods are usually a symptom of a poor design. Avoid the mistake of have "dumb" objects, each with getters/setters (essentially, just database records wrapped in objects), and one "brain" main method that does all the work. This is not object-oriented!

The ***JavaBean*** standard for objects requires a pair of methods for each *property*, with names **`get`**`Propname` and **`set`**`Propname`. Boolean properties may use **`is`**`Propname` for the getter instead. Read-only properties don't have a setter.

Enterprise frameworks and other advanced Java software use classes that follow this standard. Objects of such classes are known as Java Beans, or more commonly, just ***beans***. By following the standard, these objects can be treated as "block-box" components, added to applications without knowing the code. They can communicate with other beans using events (the *observable* pattern), be saved and restored as XML text, and even edited in a GUI (that is, a GUI bean editor can automatically detect all the properties and show them in an editor view. With a

bunch of beans, you can actually assemble a Java program without writing any code! (Note Swing components are beans, allowing building user interfaces with drag and drop in IDEs.) See the JavaBean Trail in Oracle's Java Tutorial for more.

Bean Events

Beans communicate by sending and receiving ***events***. In a visual IDE environment you normally can drag the mouse between beans, and chose what event is sent/received. All events have a pair of methods with names of `addXXXlistener/removeXXXlistener`.

ZZZ

Bean Properties

Beans have ***properties***. There are different kinds of properties depending on the intended use. All properties have a pair of `getXXX/setXXX` methods (if it is a read-write property; read-only properties only have the `getXXX` method). (Note boolean properties can use the isXXX/setXXX pair instead. This is preferred by the standard but not required.)

The Beans specification requires specific class and method naming conventions be used. By using reflection, the visual IDE can determine what properties and events are present. Sometimes the naming convention doesn't work well (say when inheriting from a class that used the special names). The specification allows for this, by allowing you to include an optional **`BeanInfo`** class. This class can provide additional information, such as an icon to use for your bean.

A visual IDE can show a *property sheet* of the bean properties, allowing you to see and change them using standard ***property editors*** for Strings, booleans, and numbers. But some properties aren't simple numbers or strings. You can include custom property editors with you bean that will be automatically found and used by the visual IDE. Also sometimes sets of properties are related and should be changed as a group (perhaps setting one constrains another). You can provide a ***customizer*** instead of (or in addition to) a bunch of property editors.

Bean properties can be **simple properties**, that take a single value such as a number, a String, a Color, etc.

An **index property** is one that gets or sets a whole array at a time. xxx

The last type is a **bound property**. This is used when you might want to have other listeners be notified when the property changes. This is done by sending a **`PropertyChange` event** to all registered listeners.

There is a special type of bound property called a ***constrained property*** that allows each listener a chance to *veto* the change. If any do then the property value is not changed. ZZZ

Bean Persistence

If you change the default properties of a bean, what happens when you quit the IDE? Since the beans are classes, the next time you start the IDE (or the application for that matter), the properties are all re-initialized using the class constructors. This problem is addressed by ***JavaBean persistence***. This is similar to serialization but not quite the same. When a bean is saved, the visual IDE notes any properties different than the defaults and saves these in an XML file, something like this:

```
JFrame frame = new JFrame();
frame.setTitle("Foo" );
frame.setVisible( true );
```

The result might be:

```
<object class="jakarta.swing.JFrame">
  <void property="title">
      <string>foo</string>
   </void>
   <void property="visible">
      <boolean>true</boolean>
   </void>
</object>
```

When next loading the bean, the constructer creates a default object and then the XML is used to change the properties to the saved values.

Lecture X — SJWToolkit (Sun Java Wireless Toolkit, a.k.a. JavaME)

The *Sun Java Wireless Toolkit for CLDC* (formerly known as Java 2 Platform, Micro Edition (J2ME, now "Java ME" or "JavaME") Wireless Toolkit) is a toolbox for developing wireless applications that are based on Java ME's *Connected Limited Device Configuration* (CLDC) and *Mobile Information Device Profile* (MIDP), and designed to run on cell phones, personal digital assistants (PDAs), and other small mobile devices.

See http://developers.sun.com/mobility/getstart/ and http://developers.sun.com/mobility/learn/midp/lifecycle/

JavaME (Java Micro Edition) is a version of Java designed for embedded devices such as PDAs and cell-phones. JavaME uses *configurations* and *profiles* to customize the Java Runtime Environment (JRE). This is necessary because of the wide range of possible devices on which a JavaME application might run. Tablets, embedded controllers, gaming consoles, PDAs, and cell-phones all have different capabilities.

Because mobile devices vary much more than desktop computers JavaME is available in different configurations.

> First there are basic hardware *configurations*. For each, one or more *profiles* has been defined. On top of the configuration/profile platform, there are a number of *optional* add-on packages available, such as for Bluetooth.

The ***configuration*** defines the basic run-time environment as a set of core classes and a specific JVM that run on specific types of devices. JavaME current ships with two configurations:

- ***Connected Limited Device Configuration*** (**CLDC**) is used specifically with the KVM for 16-bit or 32-bit devices with limited amounts of memory. This is the configuration (and the virtual machine) used for developing small JavaME applications, say for PDAs, pagers, etc. *This is the current version, and only available on Windows*.

  CLDC outlines the most basic set of libraries and Java virtual machine features required for each implementation of JavaME on highly constrained devices. CLDC targets devices with slow network connections, limited power (often battery operated), 128 KB or more of non-volatile memory, and 32 KB or more of volatile memory. (Volatile memory is non-persistent and has no write protection. With non-volatile memory, contents are persistent and write protected.)

  CLDC devices use non-volatile memory to store the run-time libraries and KVM, or another virtual machine created for a particular device. Volatile memory is used for allocating run-time memory.

- ***Connected Device Configuration*** (**CDC**) is used with the C virtual machine (CVM) and is used for 32-bit architectures requiring more than 2 MB of memory. An example of such a device is a Net TV or set-top box. Further discussion of CDC is beyond the scope of this course.

The ***profile*** defines the type of application. Specifically, it adds domain-specific classes to the JavaME configuration to define certain uses for devices. Currently there are two available; both are built on top of the CLDC: ***KJava*** and the more popular ***Mobile Information Device Profile*** (**MIDP**). These profiles are geared toward smaller devices. Thus the most common JavaME platform is CLDC/MIDP.

Compiling JavaME Applications

When developing for CLDC (smaller) devices, always keep in mind how limited are memory and other resources (such as battery life). In addition not all the normal Java SE classes are available. In addition to the **`jakarta.microedition`** package, the CLDC API consists of subsets of the `java.io`, `java.lang`, and `java.util` packages.

The device specific classes you need to use are determined by the configuration and the profile. The **KJava virtual machine**, **KVM**, accepts the same byte codes and class file format as the classic J2SE virtual machine. KJava contains a Sun-specific API that **runs on the Palm OS**. The KJava API has a great deal in common with the J2SE Abstract Windowing Toolkit (AWT). However, because it is not a standard JavaME package, its main package is **`com.sun.kjava`**.

KJava is a proprietary Sun API. It was not intended as a complete fully featured profile, but rather as a demonstration of how a profile could work with CLDC. Instead you should consider using the MIDP profile. The specification of MIDP for Palm OS was defined by the Java Community Process (JCP) and is, therefore, vendor independent.

The first step is downloading the JavaME development environment. This include the CLDC APIs (packages) and special tools. These are available from

With JavaME you use the Java SE compiler and need to invoke it with the appropriate options. You need to use the **`-bootclasspath`** option to instruct the compiler to use the JavaME classes, not the Java SE classes. Do not place the configuration classes in the compiler's `CLASSPATH`. This approach will lead to run-time errors, because the compiler automatically searches the Java SE core classes first, regardless of what is in the `CLASSPATH`. In other words, the compiler will not catch any references to classes or methods that are missing from a particular JavaME configuration, resulting in run-time errors when you try to run your application.

Deploying JavaME applications

Because JavaME is designed for small devices with limited memory, much of the usual Java verification has been removed from the virtual machine to allow for a smaller footprint. As a result, it is necessary to ***preverify*** your JavaME application before deployment. An additional check is made at run time to make sure that the class has not changed since preverification.

Exactly how to perform the preverification, or checking the classes for correctness, depends on the toolkit. CLDC comes with a command-line utility called **`preverify`**, which does the actual verification and inserts extra information into the class files. MIDP uses the wireless toolkit which comes with a GUI tool although this too can be run from the command line.

Deployment depends on the platform to which you are deploying. The application must be packaged and deployed in a format suitable for the type of JavaME device, as defined by the profile.

Java Application Descriptor **(JAD) files** describe the MIDlets (JavaME applications) that are distributed as Jar files. JAD files are commonly used to package Java applications or games that can be downloaded to mobile phones, similar to how a JNLP file describes the Jars of an application to download.

On may mobile devices you need to navigate (or dial) to a JAD file. The device will recognize that, read it, and download the associated Jar files to the correct location in your device's memory. (Usually some special Java application folder.) Occasionally you must manually download the JAD and Jars to the correct location via a USB cable, Bluetooth, or an IR connection, using vendor supplied software.

You can deploy your code in several ways:

- You upload your software over the Internet to distribution site like cell.photo.net. Your app gets assigned a phone number. Anyone wanting to load the software dials than number on their cell-phone.
- If the phone has Internet access, it is possible to download the jad (Jar?) file from an ordinary website given the URL.
- If the phone has WAP access, you upload the jad file to a host like waphq.net. Your app gets assigned a number. Anyone wanting to load the software uses their cell-phone wap browser to navigate to `http://waphq.net/wap.php` and enters the number.
- A Palm PDA has software that is called a conduit, which communicates with the device for installing software as well as transferring data for backup or update.
- Nokia cell/mobile/handy telephones are supplied with Nokia PC-Suite. You can use this to select Jar files to be installed on the phone via a USB connector.

Hello World JavaME application (MIDlet).

To run the toolkit itself, select the KToolbar shortcut. The JavaME Wireless Toolkit works with projects, where the end result of each project is one MIDlet suite. The toolkit works with one project at a time. You can change properties of the current project, build the project, and run the project in a device emulator. Several example projects come installed with the toolkit.

Click on New Project in the button bar. The toolkit prompts you for a project name and the name of a ***MIDlet*** class in the project. Fill in `HelloSuite` and `HelloMIDlet`.

The toolkit gives now you a chance to edit the project settings. Accepting the defaults is fine. Press OK to finish creating the new project. In the text output pane of the Java ME Wireless Toolkit, you'll see several helpful messages telling you where to store the project's source files.

The toolkit stores each project in a subdirectory of the application directory. The name of the directory is the same as the name of the project. Here, the toolkit has created a new directory, say in `C:\WTK22\apps\HelloSuite`. Each project directory has a standard set of subdirectories: The **`bin`** directory contains the compiled MIDlet suite (a `.jar` file) and the MIDlet suite descriptor (a `.jad` file). The **`lib`** directory is the location for any additional Jar files you would like to include in your project. **`res`** is the location for resource files, like images or text files that should be bundled with your MIDlet suite. The **`src`** directory is the place where your source code should be saved (in sub-directories if you use packages).

The Java ME Wireless Toolkit creates additional directories: **`classes`**, **`tmpclasses`**, and **`tmplib`**. For the most part you can ignore these directories; the toolkit uses them internally.

Save the following code as `HelloMIDlet.java` in the `src` directory of your project. Next, press the **`Build`** button in KToolbar. The toolkit will attempt to compile your project. If there are any compiler errors, you'll see them in the text output area of KToolbar. Fix the errors until the project builds successfully.

```
import jakarta.microedition.lcdui.*;
import jakarta.microedition.midlet.*;
public class HelloMIDlet extends MIDlet
                         implements CommandListener
{ private Form mMainForm;
  public HelloMIDlet() {
    mMainForm = new Form("HelloMIDlet");
    mMainForm.append(
      new StringItem(null, "Hello, MIDP!") );
    mMainForm.addCommand( new Command("Exit",
      Command.EXIT, 0) );
    mMainForm.setCommandListener(this);
 }
 public void startApp()
 { Display.getDisplay(this).setCurrent(mMainForm);
 }
 public void pauseApp() {}
 public void destroyApp( boolean unconditional ) {}
 public void commandAction ( Command c, Displayable s )
 { notifyDestroyed(); }
}
```

Now you're ready to test your MIDlet suite. Click on the **Run** button. You should see a mobile phone emulator pop up. The emulator is showing a list of MIDlets in the MIDlet suite. This example shows only one MIDlet. Although the name you see here is `HelloSuite`, the class that will be run is `HelloMIDlet`. To see where this mapping occurs, go to KToolbar and select `Settings...` then click on the `MIDlets` tab to see a list of the MIDlets in the project.

Back in the emulator, click on the soft button below Launch to start up the MIDlet. It will display a simple screen like the one below. Click on Exit to leave the MIDlet. Close the emulator window or hit the Escape key to end the emulator session.

The default emulator is `DefaultColorPhone`. The Java ME Wireless Toolkit has other emulators as well. Try running `HelloMIDlet` on some other devices

to see how the user interface adapts. Simply select the emulator you'd like in the combo box in KToolbar, then click on `Run` again.

The final step is to upload your MIDlet to the selected device. This usually requires you to connect your device to your computer, and to run device specific software to upload files and applications.

WebStart (old notes)

[*Adopted in part from Cay Horstmann's Core Java vol. I, ch. 10*]

> **Oracle announced in 2/2018 that WebStart is deprecated in Java 10 and will not be included in Java 11 SE.** *(So, skip to Jar Files.)*

WebStart is an alternate delivery mechanism for "thick" clients. Previously, thick clients were applets with many limitations, or applications that had no security limitations at all and needed to be deployed (downloaded and installed by the end-users) and updated regularly.

WebStart works by having (almost) regular Java applications that are cached on the local system automatically when first launched from a URL. Thus they launch and deploy like Applets but are in fact applications.

Although the JRE comes with a WebStart cache viewer (which allows an easy way to launch and create desktop shortcuts to them), they are typically launched by clicking a URL in a browser. Note the launched application is not an Applet and doesn't require the browser to run. The cache viewer (for Oracle's Java) is in the control panel somplace. (You can launch it directly with a `.bat` file of "`javaws.exe -viewer`".) (I've noticed that as of Java 9, the cache initially shows as empty. You need to click "System" then "User" to make it refresh.)

WebStart uses a more relaxed security model than applets, but there is still security. A digitally signed application is granted full access. An unsigned application can still access protected resources, but only with the user's permission. (The user is asked to allow each restricted access, but there is a checkbox to *remember this decision* so you don't have to keep clicking through these dialogs.) This works by having your Java WebStart application use different APIs for dangerous operations than normal Java SE. If your application uses the normal APIs then the security model is enforced as for applets. The API provides access methods for reading/writing files, clipboard access, network access, printing, opening the (default) browser to show an HTML document, and other features.

This API is in the extension Jar "**`javaws.jar`**" which must be in the *classpath*. To use the WebStart services, the application creates the appropriate *service manager* object to do the access. Here's an example of saving a file:

```
FileSaveService fss = (FileSaveService)

ServiceManager.lookup( "jakarta.jnlp.FileSaveService" )
```

```
;
fss.saveFileDialog( ".", new String[]{"txt"},
    dataStream, "MyApp" );
```

(See Oracle's Java WebStart Developer's Guide.)

To create a WebStart application you compile your application (make sure `javaws.jar` is on CLASSPATH), put it into a clickable Jar file, and create a *deployment descriptor* to be placed on a web page. When the user clicks the link for the deployment descriptor, the web browser launches `javaws.exe`, which checks if the latest version of the application is downloaded, and then runs it.

JNLP Files

This descriptor is an XML file in ***Java Network Launch Protocol*** (*JNLP*) format. It describes the application, lists the required Jars, and optionally asks for full desktop permissions. Note that the web server must report the correct MIME type of this file, or your browser won't know it is a deployment descriptor and very likely will just display it as text. (The same is true for PDF and other document types.):

```
mkdir mypackage
notepad mypackage/MyApp.java
javac mypackage/MyApp.java
jar -cvfe MyApp.jar mypackage.MyApp mypackage
notepad MyApp.jnlp
jarsigner ...   sign the jar; discussed later
javaws MyApp.jnlp
```

A sample JNLP 1.0 file might look like this:

```
<?xml version="1.0">
<jnlp spec="1.0+"
      codebase="http://localhost:8080/myapp/
      href="MyApp.jnlp">
<information>
<title>My Demo Application</title>
<vendor>Hymie Piffl</vendor>
<description>This application does nada.</description>
<offline-allowed/>
</information>
<resources>
  <j2se version="8.0+"/>
  <jar href="MyApp.jar"/>
</resources>
<application-desc/>
</jnlp>
```

Note that this syntax has been enhanced several times, so you could see additional tags if you use a newer version. (The WebStartDemo in the class resources uses version 6 syntax, which includes more information.)

> JNLP doesn't allow relative URLs. You must specify a full one (or, as show above, specify the full URL to the codebase). That doesn't work well in reality, since you might have your web pages served from a cluster of different web servers or hosted on a *content distribution network*.
>
> Jakarta EE has a work-around for that: A Jakarta EE application can generate JNLP files from a template and put in the correct URL.

The WebStart environment provides the `PersistenceService` class that allows your application to store safely data, without having full access to your hard drive. The service limits the data size, and only your application can read or write the data. This means that other applications on your hard drive will not be able to access information stored by your application. WebStart provides many such services.

Using JNLP to Deploy Applets

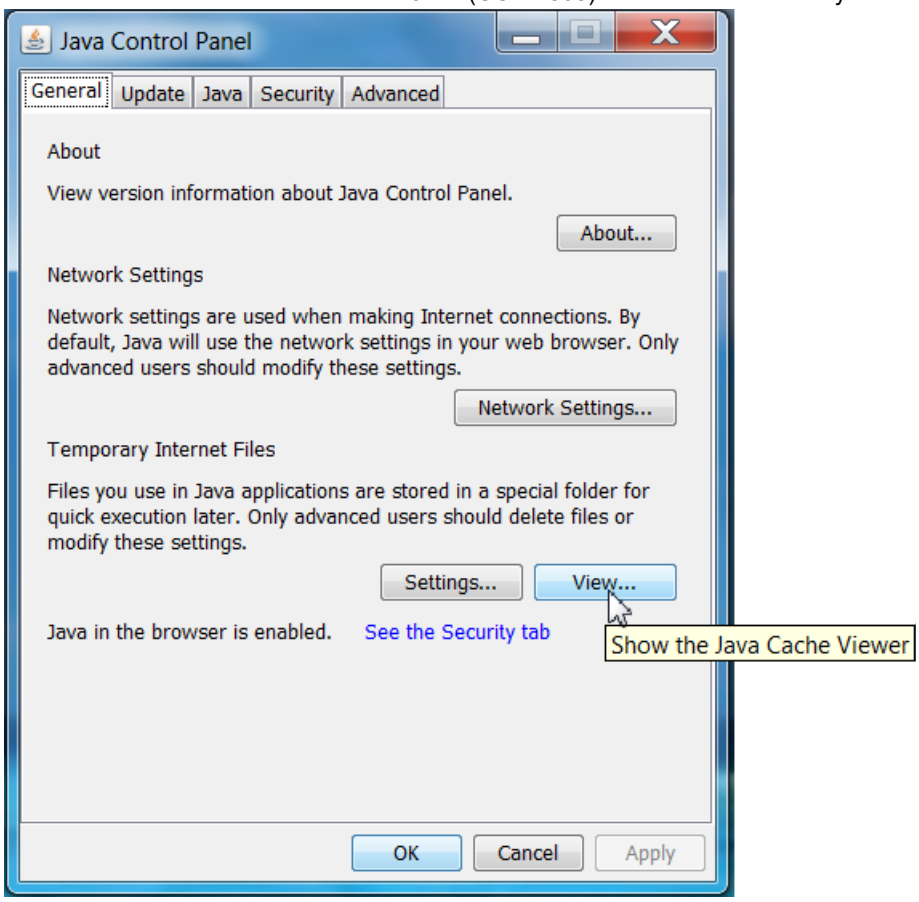
This can be done as of Java 6u10. To deploy applets using JNLP, specify the `jnlp_href` parameter in the applet tag as follows:

```
<applet width="300" height="300"
   code="com.mycompany.MyApplet">
   <param name="jnlp_href" value="my_applet.jnlp">
</applet>
```

In this example, `my_applet.jnlp` has the following deployment information:

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp href="my_applet.jnlp">
 <information>
  <title>My Applet</title>
  <vendor>My Company, Inc.</vendor>
  <offline-allowed />
 </information>
 <resources>
  <j2se version="1.4+"
    href="http://java.sun.com/products/autodl/j2se" />
  <jar href="my_applet.jar" main="true" />
  <extension name="SomeExtension"
   href="http://some.server.com/some_extension.jnlp" />
 </resources>
 <applet-desc
  name="My Applet"
  main-class="com.mycompany.MyApplet"
  width="300"  height="200">
 </applet-desc>
</jnlp>
```

You can view the local cache of WebStart downloaded applications from the Java control panel. In the "General" tab, click the "View" button in the "Temporary Internet Files" section (or, run "`javaws -viewer`" from the command line):



By default, applets and WebStart applications run in what is called a *sandbox*, which is an attempt to restrict what untrusted code can do on your system. While users can install policy files in the JRE that grant applets more access, it is difficult for end users to do so. With Java WebStart, users can do other actions (using JNLP services), but untrusted code causes pop-up dialogs whenever the user tries using your app. (Launching, caching, creating icons, opening files, etc., all causes security popups.)

> To run outside of the sandbox, WebStart and applet Jar files must be ***digitally signed***. This means the Jar file contains a PKI ***certificate*** that identifies the author and application, and is very hard to forge. (Such certificates usually cost money.) In 2012, a number of security flaws in the applet sandbox were uncovered. Many of those were patched, but there are probably others. These flaws only affect applet sandbox. This is why few people require Java applets enabled anymore. The Java 7 control panel includes a setting to easily disable applets.
>
> Regular Java applications don't run in a sandbox and remain as secure as any other downloaded program you run.

> The applet sandbox flaw doesn't affect WebStart applications, but digitally signing your Jar files will give users the perception your WebStart apps are more secure.

The JNLP file or the Jar's manifest file can contain a *security* section stating the additional permissions needed, such as "`all-permissions`".

Starting with Java 7u51, no unsigned applets or WebStart apps are allowed to run by default. However, system administrators can set up Java with exceptions (deployment rule sets) on PCs. End-users can also add exceptions to the security rules, by "allowlisting" locations from where unsigned RIAs should still be allowed. (You merely copy the URL of the directory containing the RIA into the Java control panel's Security-->Manage Exceptions dialog.)

> If using JNLP to launch an Applet or WebStart app, an identical copy of the deployment descriptor must exist in the Jar file. The JNLP file must be named `APPLICATION.JNLP` and is included in the `JNLP-INF` subdirectory. See the Signed JNLP Files section of the Java Deployment Guide for details. (I got WebStart to work without this step in Java 9.)