Lecture Notes — Packages, Modules, and Jar files

***Packages* are a collection of .class files (i.e., classes, interfaces, enums, and annotations, referred to as *types* or as *package members*) in some directory (folder).** Using packages provides access control (*package-private* access) and namespace management (so your names don't collide with other names).

This material will make more sense if you understand about files and directories, pathnames, and the special names "`.`" and "`..`". If you are unfamiliar with these concepts, remember the "Windows shell" tutorial at the top of our class resources.

A package name should consist of all lowercase letters, digits, and possibly underscores. (Package names must be legal folder/directory names for any system! Best practice is to use lowercase letters and digits only, starting with a letter, and keep the name shorter than 12 characters.) Periods have a special meaning in a package name, as discussed below.

Typically, the DNS name of an organization is used, backwards, with underscores (or nothing) replacing illegal characters (such as hyphens). Each period-separated string is a *path component*, and refers to a directory. For example, one might name a package "`com.example.myapp`". This scheme is not required and for experimenting and learning, shorter names such as "`myapp`" can be used.

To put your class *MyClass* in a package "`foo`", put the `MyClass.class` file in a directory "`foo`", and add this to the top of `MyClass.java`:

```
     package foo;        // directory "foo"
```

Some additional examples:

```
     package foo.bar;    // directory "foo/bar"
     package com.my_company.my_application;  // directory
"com/my_company/my_application"
```

The `package` declaration must go before any other Java statements in a source file, except for comments and blank lines. You can have only one `package` declaration per source file.

Without a `package` declaration, a class becomes part of a nameless package regardless of which directory it is in. (This nameless package is also known as the *default* package.) Thus, all classes are in a package.

Packages can contain other packages in a hierarchy; however, importing one package doesn't import the sub-packages! (Sun described package names as non-hierarchical.)

To refer to a class in another package, you can use its *qualified* name (e.g., "`foo.MyClass`"). Use `import` statements to allow you to refer to the classes in imported packages by their *unqualified* names (e.g., "`MyClass`"). (Review the `import` statement, page **Error! Bookmark not defined.**). A class can always refer to other classes in the same package as itself with an unqualified name; no `import` statement is needed.

It is not possible in Java for a class in a named package to refer to a class in the default, nameless package. (There's no name for the import statement.)

Wildcard imports are convenient to type and have no effect on the resulting code. But they can make your code harder for readers to understand. IDEs such as Eclipse have a feature "Source->Organize Imports" that will expand these to a list of actual class names you used.

Note that importing several packages may lead to name collisions. In that case the compiler will report an *ambiguous* name, and you must then use the *qualified* names. There are few conflicts in the Java standard library; two I know of are `Timer` and `Date`. You can add an explicit import to resolve the conflict:

```
import java.util.*;  // Has a Date class
import java.sql.*;   // Has a Date class
import java.util.Date;
... Date d = new Date();  // Uses java.util.Date
```

If you need to use both `Date` classes in one program, you will need to use a fully-qualified name for one or both of them.

From the JLS#7.4.2: "Unnamed packages are provided by the Java platform principally for convenience when developing small or temporary applications or when just beginning development." **All *production quality* Java classes should be in packages.** Note, it is not possible for a class in a named package to refer to any class in the nameless package!

How Java Finds Packages (when not using modules)

The JVM finds class files using an implementation-defined version of `java.lang.Classloader`. (Oracle's Java procedure is documented in the JDK tools documentation.) In essence, there is a list of directories (folders), jar files, and zip files that are searched, for a given class. This list of places is called the ***classpath***.

A package name is a relative *pathname* (qu: what is that?) to some directory. But what is it relative to? Java looks for packages in several places. A package may be found in some ***directory*** or in some ***jar*** or zip file; let's call these ***package containers*** (not an official term).

The classpath is searched for packages; that is, you list the places containing packages, not the package directory itself. It is also searched for classes in the default, nameless package.

The classpath is constructed from several different settings. The details are mostly unimportant, and have changed often between releases of Java. Java looks in:

1. The package containers listed in the **`bootclasspath`**, which by default contains `rt.jar` and `charsets.jar`, (different names in older versions.) These contain the core class files (the classes in the packages starting with `java.` or `javax`). (Show via 7zip.) Note: A *jar* file is a Java archive file that can contain packages and applications, and will be discussed later. On Unix/Linux/Mac, directories in the list use a colon separator. On Windows a semicolon is used.

   Some packages are not developed by Sun/Oracle but are still part of the standard JRE library. These are called *endorsed APIs.* Such packages may be included in the jars listed in `bootclasspath`, but can also be in jars in the directories listed by another Java property, `java.endorsed.dirs`. As of Java 6 the endorsed packages are the `org.omg.*` ones, plus `javax.rmi.CORBA`. Separating core and endorsed packages this way makes it easier to update the endorsed packages between JRE releases. You could change the location to search for jars containing endorsed packages with the option "`-Djava.endorsed.dirs=`*`directories`*".

   As of Java 8U40, the endorsed directories concept has been deprecated.

2. The ***extension directories*** can contain jar files that will be searched for packages. As of Java 8U40, the extension directories concept has been deprecated.

3. The general **`CLASSPATH`**. This is affected by the `CLASSPATH` environment variable (see below) or specifying `-classpath` (or `-cp`) on the command line of JDK tools, as discussed below. **The `CLASSPATH` is a list of package containers, used to search for packages, classes in the nameless package, or other resource files (e.g., ".gif" files).** This list uses a colon or semi-colon as a separator, just like the `bootclasspath`.

CLASSPATH

`CLASSPATH` is an ***environment variable*** (Qu: what is that?) that tells Java where to look for packages. (Look at `CLASSPATH` now.) It is a semicolon separated list of directories and jar files that contain packages. It is ***not*** a list of packages. (On Unix, it is a colon separated list.) (See the Oracle Java 8 CLASSPATH documentation for all the details.)

(Applets don't use `CLASSPATH` to locate packages. Qu: Why not?)

Classes in the default nameless package are also found by searching `CLASSPATH`.

If you don't set `CLASSPATH` at all, the default path of "`.`" is used (the current directory). This is just what you want most of the time; you'll only need to change this in advanced situations.

Example: Suppose you want Java to find a class named `Foo.class` in the package `util.myApp`. If the path to that directory is `C:\java\MyClasses\util\myApp`, you would set the class path so that it contains `C:\java\MyClasses`.

You can use `java[c]` **`-classpath`** (or `-cp`) ***`dir-list`*** to add to the classpath. (Use `javac -d` *`dir`* to say to put generated class files under *dir*.)

When working from the command line, it is often easier to put the `.java` files in the directory where you want the `.class` file to go. Now you can "`cd`" to the top level of your application (above any packages), and compile the top `.java` file. `javac` will find and compile all `.java` files in all packages automatically, as needed. (And only if needed: if the `.class` file and `.java` file are both found, the timestamps of the two are compared to see if the `.java` file needs to be compiled again.)

(**Demo**: `C:\Java>`**`javac C:\Java\pkg\Foo.java`**) If the source files are kept elsewhere, you can use `javac --sourcepath` *`list`*.

Normally, `javac` puts generated `.class` files into the same directory as the source file. Afterward, you need to create the package folders and move the `.class` files into them. By using `-d` *`dir`* option to `javac`, you can specify where to put packages (or classes in the nameless package). For example, if `Foo.java` has the package statement "`package pkg;`", and the source file is in the current directory (so there is no `pkg` folder yet), running "`javac -d . Foo.java`" will create "`pkg/Foo.class`".

CLASSPATH (since Java 6) can contain the wildcard "`*`" to refer to all JAR files in some directory. To add all the JARs in directory "`foo`" to your CLASSPATH, just use "`.;foo/*`". (Wildcards can't be used in the `Class-Path` JAR manifest header.) Wildcards are thus similar to the extensions mechanism, now deprecated.

Best practice is to create a directory (I call mine "`C:\Java\MyJars`") and put all additional jar files (such as `JUnit.jar`) in there. Then include this directory: `CLASSPATH=.;C:\Java\MyJars\*`. You can also list directories where you install things such as Glassfish, Tomcat, etc., that contain many jars that must otherwise be listed on CLASSPATH.

Java 7 includes a useful non-standard option that can show many things, including the classpath: `java -XshowSettings -version`

Dangers of `CLASSPATH`: Runnable jar files require more work to use *optional* packages, but it is possible. Extension directories have security issues.

Copying jars into a directory listed on CLASSPATH is dangerous since updates won't automatically get copied there. This problem is known as *deployment*; when you change/update any package, you must also remember to deploy it (copy the jar) to the correct directories. You have the same problem for any packages you use. For example, database driver jars; when you update Derby DB (a.k.a. JavaDB), you must remember to copy the new jars. In some cases, I prefer to add additional directories to CLASSPATH; it is for this reason I generally install in directories without any version numbers (so I don't have to remember to update CLASSPATH). For example, "`CLASSPATH=.;C:\Java\db\lib\*;C:\Java\MyExt\*`"

The use of `CLASSPATH` is the result of using the default ***ClassLoader***. Sometimes a custom ClassLoader is used instead (for security or other special purposes), and then none of this section applies. See also `java.lang.ClassLoader` for some more details on the search done.

Extensions found on a system-wide `CLASSPATH` are visible to every Java program. If another program decides that it needs different version of the same classes, then either one version or the other version will not be loaded (that is, both apps will use the same version). **Using the `-cp` option, or per-project `CLASSPATH`, settings will avoid this problem.**

To run a stand-alone application that's in a jar, double-click it or run as `java -jar` *`file.jar`*. But in this case **Java ignores the `CLASSPATH`**. It will only look inside of that JAR. You need to set the `Class-Path:` *`list`* entry in the *manifest* file included in that jar instead. (The `jar` tool is discussed later.)

It is practically impossible to have one `CLASSPATH` that satisfies all possible compilations and all possible applications. Installers will meddle with it, causing programs to work and then mysteriously stop working after you update something unrelated but that changes `CLASSPATH`.

So the **best practice** is to keep your dependencies contained to the app that needs them, by using jars and listing a custom `Class-Path:` *`list`* in the manifest (or using "`-cp` *`list`*" on the command line), and avoid using the (global) `CLASSPATH` environment variable. (Note, this is what IDEs do internally, when you change project settings.) Second-best: use wildcard settings in `CLASSPATH` (to define a JRE-independent extension directory).

Java Web Start lets you skip the JAR `Class-Path` entries in favor of JWS' own "libraries" mechanism. This still relies on being allowed to distribute 3rd party JARs with your application, but it makes managing them and launching the program easier.

Remember: If you put a class in a file without a package statement, it belongs to a "nameless" package by default. Java uses the same mechanisms to find these as it uses to find packages.

Remember: Java source files can have only one `package` statement, which must be the first statement in the file (that means before any `import` statements).

Review the standard Java packages (API). Show the on-line API Docs, have students set a bookmark/favorite to it. **Note**: Never import `java.lang`. Also note that `java.math` (a package) is *not* the same as `java.lang.Math` (a class).

Show **PkgDemo.htm**. Next, try moving your package to another location. Use the "`-cp`" cmd line arg. Next bundle it in a jar and use it via `-cp`. Now move the jar to the JDK's ext dir, then the JRE's ext dir, and then the system-wide ext dir. Use "`java -`

`XshowSettings`" to see the ext directories used for your system. Next, try putting PkgDemo inside a new package. Point out you need to compile from the *base* directory (containing your packages), so `javac` can find your packages by looking in the "`.`" (current) directory. You need to run from there too.

Summary: Using non-standard (that is, not part of the JRE) packages with your code can be a pain. You can bundle these packages with your own code in a single jar file (jar files can contain other jar files), use Java Web Start to deploy your applications, or make users download the packages and install them correctly, themselves. Always consider the problems of deployment and version changes when using non-standard packages.

Ignoring the JRE bundled packages (and the deprecated features of extension and endorsed directories), Java finds other packages (and classes in the nameless package):

1. in the current directory if `CLASSPATH` isn't set (the default);
2. the package containers listed in `CLASSPATH` override the default;
3. the package containers listed with `-cp`, which override any `CLASSPATH` setting;
4. the package containers listed inside a jar file, which overrides all other settings.

Remember, `CLASSPATH` can list wildcard directories, which adds all jar files in those directories to the classpath. The classpath setting inside a jar file cannot contain wildcard directories.

The rules of Java make it difficult to create a self-contained application. A new JDK tool called the Java Packager tool makes this possible.

Modules — *Since Java 9*

All the changes to how Java locates packages over the years discussed above is strong evidence that the whole mechanism is poor. Applications often need bundles of packages, and they need to be specified in a specific order to have a reproducible build. In fact, the issues have become known as Jar Hell.

To address these issues, Java 9 added a new mechanism to Java: modules. Just like a package is a collection of classes, **a *module* is a collection of packages**.

Modules don't use classpath, but rather a *module path*, to locate modules. Modules provide lots of checking for consistent use. For example, only packages that are declared *exported* can be used by other modules.

Modules was known as *project Jigsaw*, started before 2011, and was originally planned for Java 7. It took many years to get it right!

Creating a module is easy. First, create a directory to hold your module. Normally, you name the folder the same as the module, but that's not required:

```
mkdir foo
```

Now you create your packages inside of that folder:

```
mkdir foo\bar
notepad foo\bar\Main.java
```

In this example, my class is named `bar.Main` (the class `Main` in the package `bar`). Assume the code is a basic Hello World app.

Finally, you add a module descriptor to the module's directory. The descriptor is named **`module-info.java`**:

```
notepad foo\module-info.java
```

The contents of that file state the module's name, what other modules it depends on, and which packages it makes visible to other modules. A working and minimal file in this case is:

```
module foo {}
```

That's it! Module `foo` does not depend on ("require") any other modules, and exports no packages. If it depended on a module x.y.z, you would need to add "`requires x.y.z;`" in there.

The compiled module needs to be put somewhere. The convention is to use a directory named "`mods`" for your modules:

```
mkdir mods
javac -d mods\foo foo\module-info.java \
   foo\bar\Main.java
```

To run your module, you must set the module path so it can be found. Then you specify the name of the class to run using the syntax *module/class*:

```
java -p mods -m foo/bar.Main
```

(The "`-m`" is to specify the class from a module; they did that I think to preserve compatibility with older Java, so you can run Java 8 apps the same way we always did.)

Modules are typically packaged in Jar files (discussed below). Such *modular Jar files* are commonly kept in a directory named `mlib` (where "`lib`" is the common name for non-modular Jars):

```
mkdir mlib
jar -cfe mlib\foo.jar bar.Main -C bar .
```

If you set a main class as in the example, you can then run your application with:

```
java -p mlib -m foo
```

Note there is no environment variable for setting the module path. It is also not well documented what can be listed on the module path; clearly, at least directories containing module Jar files or "exploded" modules can be listed.

The JRE has been modularized as well. All Java SE classes are in one or another module, whose names begin with "`java.`", and are found in `JAVA_HOME\jmods`. Other, non-standard modules shipped by your JDK vendor will have names starting with "`jdk.`"

The standard modules are not in Jars, but a newer format with the extension "`.jmod`". You can still open them with a zip utility to see what's in them. Indeed, Java 9 introduced several new formats, including jmod, modular Jars, and multi-version Jars.

Similar to how the `java.lang` package contains the most basic classes and does not need to be imported, **the most basic packages are in the `java.basic` module and that is always available; you do not have to add a `requires` statement for it to your `module-info.java` file.** However, `java.basic` does not include all of the Java SE classes! For more than Hello, world, you may need to add one or more `requires` statements. Note, the Java 9 API docs now also list the module containing a class, and not just the package. You can also use the JDK tool jdeps to find and show all the dependencies of a module.

All packages are in a module. If you don't have a module descriptor (`module-info.java`), then your package ends up in the unnamed module. Similar to the unnamed package, you can use other modules in your code. However, code in named modules cannot use code in the unnamed module (there's no way to "`requires`" it since it has no name).

Eclipse Oxygen 1a supports Java 9 and modules, but not completely yet. To put your packages in a module, create a properly named source folder in Eclipse, named for the module. Put your packages in there rather than "`src`". Right-click the project, choose "configure-->add module-info.java". Edit the generate file so it has the correct exports (if any). That's it!

There's more to know about modules and how they work with legacy packages. More details on modules will be presented in the Advanced course.

Jar Files — Creating and Using

Zip = Jar, except for ***Manifest* file** (and other optional extra files). Describe the `jar` tool:

```
  jar{-c|-t|-x|-u|…}[-v]-f file.jar -e main-class
```

Point out JAR files can hold packages and can be listed on CLASSPATH. Jars are also used to hold WAR files (web archive, a whole Java EE web application with the extension .war instead of .jar), EAR files (enterprise archives, containing an entire enterprise Java EE application including Jars and WARs), create clickable stand-alone applications, or create *signed* classes and/or *sealed* packages (for extra security).

Jar files can also hold media.

Show SmileJar.java. (Add a splash-screen PNG "Miracle Software Co.")

To compile or run some Java that uses a class in a Jar file, you must make sure that Jar file is listed on `CLASSPATH`. If this is a common situation, you can modify `CLASSPATH`. If you follow my suggestion on setting `CLASSPATH`, your `CLASSPATH` is already set to something such as this:

```
C:\Temp>set CLASSPATH
CLASSPATH=.;C:\Java\MyExt\*
```

If so, you only need to ensure Jar files are in the directory `C:\Java\MyExt` and they will be found.

If you don't want to modify `CLASSPATH` permanently, you can compile and run your program using an option to temporarily set the `CLASSPATH`. To compile `MyFile.java` with the `foo.jar` Jar file, use this:

```
C:\Temp>javac -cp.;foo.jar MyFile.java
```

The `java` command also has the "`-cp`" option, for running programs without needing to modify `CLASSPATH`. (Build tools such as Eclipse, NetBeans, Maven, and Ant, use this feature to set a `CLASSPATH` per project.)

(It seems in Java 8, you can get away with just "`-cp foo.jar`", but the docs say you need the dot too.)

Demo downloading a package from the Internet and using it: Download Apache Commons Math packages (in a zip). Show API doc for ...`Primes` class. Extract the JAR file and put into the `Temp` folder. Now use:

```
import org.apache.commons.math3.primes.*;
public class Foo {
    public static void main ( String [] args ) {
        for ( int i = 0; i < 20; ++i )
            System.out.println( i + ": " + Primes.isPrime( i )
);
    }
}
```

Compile it:

```
  javac -cp .;commons-math3-3.6.1.jar Foo.java
```

Run it:

```
  java -cp .;commons-math3-3.6.1.jar Foo
```

Note you need to have the JAR file on your CLASSPATH when compiling and when running.

Finally, show how to use `C:\Java\MyExt` and `CLASSPATH`, to avoid needing to use the `-cp` option each time.