Multithreading (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.

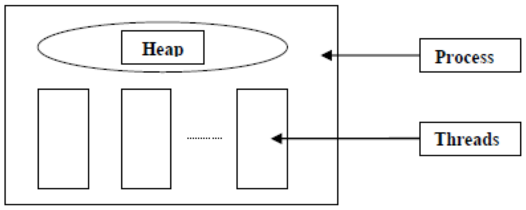
Running multiple tasks at the same time on different hardware is a type of concurrency known as *parallelism*.

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. 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 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.

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 sharable (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.

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*). Less well-known liveness issues include *livelock* and *thread starvation*. ***Performance* refers to having a program complete a task in a timely manner.** 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 operationsError: Reference source not found") 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.)

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: `javax.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 `javax.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 `long`s and `double`s 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 **`javax.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 28).

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 wioth per-thread field values).

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 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. Or use some framework that handles security and transaction details for you. (These likely use `ThreadLocal`s internally but correctly.)

Using a local variable in `run` is private to a thread (as it's on the stack), but 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.

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 when you think they might.

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 [*Quoting from InfoWorld article, 3/6/2024*]

"`ScopedValues` is a new way [a Java preview feature since Java 21] 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.

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 `javax.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 `javax.swing.*`!