**Java Concurrency Tutorial**

<https://docs.oracle.com/javase/tutorial/index.html>

https://docs.oracle.com/javase/tutorial/essential/index.html

<https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>

**Lesson: Concurrency**

Computer users take it for granted that their systems can do more than one thing at a time. They assume that they can continue to work in a word processor, while other applications download files, manage the print queue, and stream audio. Even a single application is often expected to do more than one thing at a time. For example, that streaming audio application must simultaneously read the digital audio off the network, decompress it, manage playback, and update its display. Even the word processor should always be ready to respond to keyboard and mouse events, no matter how busy it is reformatting text or updating the display. Software that can do such things is known as *concurrent* software.

The Java platform is designed from the ground up to support concurrent programming, with basic concurrency support in the Java programming language and the Java class libraries. Since version 5.0, the Java platform has also included high-level concurrency APIs. This lesson introduces the platform's basic concurrency support and summarizes some of the high-level APIs in the java.util.concurrent packages.

# Processes and Threads

In concurrent programming, there are two basic units of execution: *processes* and *threads*. In the Java programming language, concurrent programming is mostly concerned with threads. However, processes are also important.

A computer system normally has many active processes and threads. This is true even in systems that only have a single execution core, and thus only have one thread actually executing at any given moment. Processing time for a single core is shared among processes and threads through an OS feature called time slicing.

It's becoming more and more common for computer systems to have multiple processors or processors with multiple execution cores. This greatly enhances a system's capacity for concurrent execution of processes and threads — but concurrency is possible even on simple systems, without multiple processors or execution cores.

## Processes

A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.

Processes are often seen as synonymous with programs or applications. However, what the user sees as a single application may in fact be a set of cooperating processes. To facilitate communication between processes, most operating systems support *Inter Process Communication* (IPC) resources, such as pipes and sockets. IPC is used not just for communication between processes on the same system, but processes on different systems.

Most implementations of the Java virtual machine run as a single process. A Java application can create additional processes using a [ProcessBuilder](https://docs.oracle.com/javase/8/docs/api/java/lang/ProcessBuilder.html) object. Multiprocess applications are beyond the scope of this lesson.

## Threads

Threads are sometimes called *lightweight processes*. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.

Threads exist within a process — every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

Multithreaded execution is an essential feature of the Java platform. Every application has at least one thread — or several, if you count "system" threads that do things like memory management and signal handling. But from the application programmer's point of view, you start with just one thread, called the *main thread*. This thread has the ability to create additional threads, as we'll demonstrate in the next section.

# Thread Objects

Each thread is associated with an instance of the class [Thread](https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html). There are two basic strategies for using Thread objects to create a concurrent application.

* To directly control thread creation and management, simply instantiate Thread each time the application needs to initiate an asynchronous task.
* To abstract thread management from the rest of your application, pass the application's tasks to an *executor*.

This section documents the use of Thread objects. Executors are discussed with other [high-level concurrency objects](https://docs.oracle.com/javase/tutorial/essential/concurrency/highlevel.html).

**Defining and Starting a Thread**

An application that creates an instance of Thread must provide the code that will run in that thread. There are two ways to do this:

* *Provide a Runnable object.* The [Runnable](https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html) interface defines a single method, run, meant to contain the code executed in the thread. The Runnable object is passed to the Thread constructor, as in the [HelloRunnable](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/HelloRunnable.java) example:
* public class HelloRunnable implements Runnable {
* public void run() {
* System.out.println("Hello from a thread!");
* }
* public static void main(String args[]) {
* (new Thread(new HelloRunnable())).start();
* }
* }
* *Subclass Thread.* The Thread class itself implements Runnable, though its run method does nothing. An application can subclass Thread, providing its own implementation of run, as in the [HelloThread](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/HelloThread.java) example:
* public class HelloThread extends Thread {
* public void run() {
* System.out.println("Hello from a thread!");
* }
* public static void main(String args[]) {
* (new HelloThread()).start();
* }
* }

Notice that both examples invoke Thread.start in order to start the new thread.

Which of these idioms should you use? The first idiom, which employs a Runnable object, is more general, because the Runnable object can subclass a class other than Thread. The second idiom is easier to use in simple applications, but is limited by the fact that your task class must be a descendant of Thread. This lesson focuses on the first approach, which separates the Runnable task from the Thread object that executes the task. Not only is this approach more flexible, but it is applicable to the high-level thread management APIs covered later.

The Thread class defines a number of methods useful for thread management. These include static methods, which provide information about, or affect the status of, the thread invoking the method. The other methods are invoked from other threads involved in managing the thread and Thread object. We'll examine some of these methods in the following sections.

# Pausing Execution with Sleep

Thread.sleep causes the current thread to suspend execution for a specified period. This is an efficient means of making processor time available to the other threads of an application or other applications that might be running on a computer system. The sleep method can also be used for pacing, as shown in the example that follows, and waiting for another thread with duties that are understood to have time requirements, as with the SimpleThreads example in a later section.

Two overloaded versions of sleep are provided: one that specifies the sleep time to the millisecond and one that specifies the sleep time to the nanosecond. However, these sleep times are not guaranteed to be precise, because they are limited by the facilities provided by the underlying OS. Also, the sleep period can be terminated by interrupts, as we'll see in a later section. In any case, you cannot assume that invoking sleep will suspend the thread for precisely the time period specified.

The [SleepMessages](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/SleepMessages.java) example uses sleep to print messages at four-second intervals:

public class SleepMessages {

public static void main(String args[])

throws InterruptedException {

String importantInfo[] = {

"Mares eat oats",

"Does eat oats",

"Little lambs eat ivy",

"A kid will eat ivy too"

};

for (int i = 0;

i < importantInfo.length;

i++) {

//Pause for 4 seconds

Thread.sleep(4000);

//Print a message

System.out.println(importantInfo[i]);

}

}

}

Notice that main declares that it throws InterruptedException. This is an exception that sleep throws when another thread interrupts the current thread while sleep is active. Since this application has not defined another thread to cause the interrupt, it doesn't bother to catch InterruptedException.

# Interrupts

An *interrupt* is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate. This is the usage emphasized in this lesson.

A thread sends an interrupt by invoking [interrupt](https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html#interrupt--) on the Thread object for the thread to be interrupted. For the interrupt mechanism to work correctly, the interrupted thread must support its own interruption.

## Supporting Interruption

How does a thread support its own interruption? This depends on what it's currently doing. If the thread is frequently invoking methods that throw InterruptedException, it simply returns from the run method after it catches that exception. For example, suppose the central message loop in the SleepMessages example were in the run method of a thread's Runnable object. Then it might be modified as follows to support interrupts:

for (int i = 0; i < importantInfo.length; i++) {

// Pause for 4 seconds

try {

Thread.sleep(4000);

} catch (InterruptedException e) {

// We've been interrupted: no more messages.

return;

}

// Print a message

System.out.println(importantInfo[i]);

}

Many methods that throw InterruptedException, such as sleep, are designed to cancel their current operation and return immediately when an interrupt is received.

What if a thread goes a long time without invoking a method that throws InterruptedException? Then it must periodically invoke Thread.interrupted, which returns true if an interrupt has been received. For example:

for (int i = 0; i < inputs.length; i++) {

heavyCrunch(inputs[i]);

if (Thread.interrupted()) {

// We've been interrupted: no more crunching.

return;

}

}

In this simple example, the code simply tests for the interrupt and exits the thread if one has been received. In more complex applications, it might make more sense to throw an InterruptedException:

if (Thread.interrupted()) {

throw new InterruptedException();

}

This allows interrupt handling code to be centralized in a catch clause.

## The Interrupt Status Flag

The interrupt mechanism is implemented using an internal flag known as the *interrupt status*. Invoking Thread.interrupt sets this flag. When a thread checks for an interrupt by invoking the static method Thread.interrupted, interrupt status is cleared. The non-static isInterrupted method, which is used by one thread to query the interrupt status of another, does not change the interrupt status flag.

By convention, any method that exits by throwing an InterruptedException clears interrupt status when it does so. However, it's always possible that interrupt status will immediately be set again, by another thread invoking interrupt.

# Joins

The join method allows one thread to wait for the completion of another. If t is a Thread object whose thread is currently executing,

t.join();

causes the current thread to pause execution until t's thread terminates. Overloads of join allow the programmer to specify a waiting period. However, as with sleep, join is dependent on the OS for timing, so you should not assume that join will wait exactly as long as you specify.

Like sleep, join responds to an interrupt by exiting with an InterruptedException.

# The SimpleThreads Example

The following example brings together some of the concepts of this section. [SimpleThreads](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/SimpleThreads.java) consists of two threads. The first is the main thread that every Java application has. The main thread creates a new thread from the Runnable object, MessageLoop, and waits for it to finish. If the MessageLoop thread takes too long to finish, the main thread interrupts it.

The MessageLoop thread prints out a series of messages. If interrupted before it has printed all its messages, the MessageLoop thread prints a message and exits.

public class SimpleThreads {

// Display a message, preceded by

// the name of the current thread

static void threadMessage(String message) {

String threadName =

Thread.currentThread().getName();

System.out.format("%s: %s%n",

threadName,

message);

}

private static class MessageLoop

implements Runnable {

public void run() {

String importantInfo[] = {

"Mares eat oats",

"Does eat oats",

"Little lambs eat ivy",

"A kid will eat ivy too"

};

try {

for (int i = 0;

i < importantInfo.length;

i++) {

// Pause for 4 seconds

Thread.sleep(4000);

// Print a message

threadMessage(importantInfo[i]);

}

} catch (InterruptedException e) {

threadMessage("I wasn't done!");

}

}

}

public static void main(String args[])

throws InterruptedException {

// Delay, in milliseconds before

// we interrupt MessageLoop

// thread (default one hour).

long patience = 1000 \* 60 \* 60;

// If command line argument

// present, gives patience

// in seconds.

if (args.length > 0) {

try {

patience = Long.parseLong(args[0]) \* 1000;

} catch (NumberFormatException e) {

System.err.println("Argument must be an integer.");

System.exit(1);

}

}

threadMessage("Starting MessageLoop thread");

long startTime = System.currentTimeMillis();

Thread t = new Thread(new MessageLoop());

t.start();

threadMessage("Waiting for MessageLoop thread to finish");

// loop until MessageLoop

// thread exits

while (t.isAlive()) {

threadMessage("Still waiting...");

// Wait maximum of 1 second

// for MessageLoop thread

// to finish.

t.join(1000);

if (((System.currentTimeMillis() - startTime) > patience)

&& t.isAlive()) {

threadMessage("Tired of waiting!");

t.interrupt();

// Shouldn't be long now

// -- wait indefinitely

t.join();

}

}

threadMessage("Finally!");

}

}

# Synchronization

Threads communicate primarily by sharing access to fields and the objects reference fields refer to. This form of communication is extremely efficient, but makes two kinds of errors possible: *thread interference* and *memory consistency errors*. The tool needed to prevent these errors is *synchronization*.

However, synchronization can introduce thread contention, which occurs when two or more threads try to access the same resource simultaneously and cause the Java runtime to execute one or more threads more slowly, or even suspend their execution. [Starvation and livelock](https://docs.oracle.com/javase/tutorial/essential/concurrency/starvelive.html) are forms of thread contention. See the section [Liveness](https://docs.oracle.com/javase/tutorial/essential/concurrency/liveness.html) for more information.

This section covers the following topics:

* [Thread Interference](https://docs.oracle.com/javase/tutorial/essential/concurrency/interfere.html) describes how errors are introduced when multiple threads access shared data.
* [Memory Consistency Errors](https://docs.oracle.com/javase/tutorial/essential/concurrency/memconsist.html) describes errors that result from inconsistent views of shared memory.
* [Synchronized Methods](https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html) describes a simple idiom that can effectively prevent thread interference and memory consistency errors.
* [Implicit Locks and Synchronization](https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html) describes a more general synchronization idiom, and describes how synchronization is based on implicit locks.
* [Atomic Access](https://docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html) talks about the general idea of operations that can't be interfered with by other threads.

**Thread Interference**

Consider a simple class called [Counter](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/Counter.java)

class Counter {

private int c = 0;

public void increment() {

c++;

}

public void decrement() {

c--;

}

public int value() {

return c;

}

}

Counter is designed so that each invocation of increment will add 1 to c, and each invocation of decrement will subtract 1 from c. However, if a Counter object is referenced from multiple threads, interference between threads may prevent this from happening as expected.

Interference happens when two operations, running in different threads, but acting on the same data, *interleave*. This means that the two operations consist of multiple steps, and the sequences of steps overlap.

It might not seem possible for operations on instances of Counter to interleave, since both operations on c are single, simple statements. However, even simple statements can translate to multiple steps by the virtual machine. We won't examine the specific steps the virtual machine takes — it is enough to know that the single expression c++ can be decomposed into three steps:

1. Retrieve the current value of c.
2. Increment the retrieved value by 1.
3. Store the incremented value back in c.

The expression c-- can be decomposed the same way, except that the second step decrements instead of increments.

Suppose Thread A invokes increment at about the same time Thread B invokes decrement. If the initial value of c is 0, their interleaved actions might follow this sequence:

1. Thread A: Retrieve c.
2. Thread B: Retrieve c.
3. Thread A: Increment retrieved value; result is 1.
4. Thread B: Decrement retrieved value; result is -1.
5. Thread A: Store result in c; c is now 1.
6. Thread B: Store result in c; c is now -1.

Thread A's result is lost, overwritten by Thread B. This particular interleaving is only one possibility. Under different circumstances it might be Thread B's result that gets lost, or there could be no error at all. Because they are unpredictable, thread interference bugs can be difficult to detect and fix.

# Memory Consistency Errors

*Memory consistency errors* occur when different threads have inconsistent views of what should be the same data. The causes of memory consistency errors are complex and beyond the scope of this tutorial. Fortunately, the programmer does not need a detailed understanding of these causes. All that is needed is a strategy for avoiding them.

The key to avoiding memory consistency errors is understanding the *happens-before* relationship. This relationship is simply a guarantee that memory writes by one specific statement are visible to another specific statement. To see this, consider the following example. Suppose a simple int field is defined and initialized:

int counter = 0;

The counter field is shared between two threads, A and B. Suppose thread A increments counter:

counter++;

Then, shortly afterwards, thread B prints out counter:

System.out.println(counter);

If the two statements had been executed in the same thread, it would be safe to assume that the value printed out would be "1". But if the two statements are executed in separate threads, the value printed out might well be "0", because there's no guarantee that thread A's change to counter will be visible to thread B — unless the programmer has established a happens-before relationship between these two statements.

There are several actions that create happens-before relationships. One of them is synchronization, as we will see in the following sections.

We've already seen two actions that create happens-before relationships.

* When a statement invokes Thread.start, every statement that has a happens-before relationship with that statement also has a happens-before relationship with every statement executed by the new thread. The effects of the code that led up to the creation of the new thread are visible to the new thread.
* When a thread terminates and causes a Thread.join in another thread to return, then all the statements executed by the terminated thread have a happens-before relationship with all the statements following the successful join. The effects of the code in the thread are now visible to the thread that performed the join.

For a list of actions that create happens-before relationships, refer to the [Summary page of the java.util.concurrent package.](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html#MemoryVisibility).

# Synchronized Methods

The Java programming language provides two basic synchronization idioms: *synchronized methods* and *synchronized statements*. The more complex of the two, synchronized statements, are described in the next section. This section is about synchronized methods.

To make a method synchronized, simply add the synchronized keyword to its declaration:

public class SynchronizedCounter {

private int c = 0;

public synchronized void increment() {

c++;

}

public synchronized void decrement() {

c--;

}

public synchronized int value() {

return c;

}

}

If count is an instance of SynchronizedCounter, then making these methods synchronized has two effects:

* First, it is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.
* Second, when a synchronized method exits, it automatically establishes a happens-before relationship with *any subsequent invocation* of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

Note that constructors cannot be synchronized — using the synchronized keyword with a constructor is a syntax error. Synchronizing constructors doesn't make sense, because only the thread that creates an object should have access to it while it is being constructed.

**Warning:** When constructing an object that will be shared between threads, be very careful that a reference to the object does not "leak" prematurely. For example, suppose you want to maintain a List called instances containing every instance of class. You might be tempted to add the following line to your constructor:

instances.add(this);

But then other threads can use instances to access the object before construction of the object is complete.

Synchronized methods enable a simple strategy for preventing thread interference and memory consistency errors: if an object is visible to more than one thread, all reads or writes to that object's variables are done through synchronized methods. (An important exception: final fields, which cannot be modified after the object is constructed, can be safely read through non-synchronized methods, once the object is constructed) This strategy is effective, but can present problems with [liveness](https://docs.oracle.com/javase/tutorial/essential/concurrency/liveness.html), as we'll see later in this lesson.

# Intrinsic Locks and Synchronization

Synchronization is built around an internal entity known as the *intrinsic lock* or *monitor lock*. (The API specification often refers to this entity simply as a "monitor.") Intrinsic locks play a role in both aspects of synchronization: enforcing exclusive access to an object's state and establishing happens-before relationships that are essential to visibility.

Every object has an intrinsic lock associated with it. By convention, a thread that needs exclusive and consistent access to an object's fields has to *acquire* the object's intrinsic lock before accessing them, and then *release* the intrinsic lock when it's done with them. A thread is said to *own* the intrinsic lock between the time it has acquired the lock and released the lock. As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock.

When a thread releases an intrinsic lock, a happens-before relationship is established between that action and any subsequent acquisition of the same lock.

## Locks In Synchronized Methods

When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns. The lock release occurs even if the return was caused by an uncaught exception.

You might wonder what happens when a static synchronized method is invoked, since a static method is associated with a class, not an object. In this case, the thread acquires the intrinsic lock for the Class object associated with the class. Thus access to class's static fields is controlled by a lock that's distinct from the lock for any instance of the class.

## Synchronized Statements

Another way to create synchronized code is with *synchronized statements*. Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock:

public void addName(String name) {

synchronized(this) {

lastName = name;

nameCount++;

}

nameList.add(name);

}

In this example, the addName method needs to synchronize changes to lastName and nameCount, but also needs to avoid synchronizing invocations of other objects' methods. (Invoking other objects' methods from synchronized code can create problems that are described in the section on [Liveness](https://docs.oracle.com/javase/tutorial/essential/concurrency/liveness.html).) Without synchronized statements, there would have to be a separate, unsynchronized method for the sole purpose of invoking nameList.add.

Synchronized statements are also useful for improving concurrency with fine-grained synchronization. Suppose, for example, class MsLunch has two instance fields, c1 and c2, that are never used together. All updates of these fields must be synchronized, but there's no reason to prevent an update of c1 from being interleaved with an update of c2 — and doing so reduces concurrency by creating unnecessary blocking. Instead of using synchronized methods or otherwise using the lock associated with this, we create two objects solely to provide locks.

public class MsLunch {

private long c1 = 0;

private long c2 = 0;

private Object lock1 = new Object();

private Object lock2 = new Object();

public void inc1() {

synchronized(lock1) {

c1++;

}

}

public void inc2() {

synchronized(lock2) {

c2++;

}

}

}

Use this idiom with extreme care. You must be absolutely sure that it really is safe to interleave access of the affected fields.

## Reentrant Synchronization

Recall that a thread cannot acquire a lock owned by another thread. But a thread *can* acquire a lock that it already owns. Allowing a thread to acquire the same lock more than once enables *reentrant synchronization*. This describes a situation where synchronized code, directly or indirectly, invokes a method that also contains synchronized code, and both sets of code use the same lock. Without reentrant synchronization, synchronized code would have to take many additional precautions to avoid having a thread cause itself to block.

# Atomic Access

In programming, an *atomic* action is one that effectively happens all at once. An atomic action cannot stop in the middle: it either happens completely, or it doesn't happen at all. No side effects of an atomic action are visible until the action is complete.

We have already seen that an increment expression, such as c++, does not describe an atomic action. Even very simple expressions can define complex actions that can decompose into other actions. However, there are actions you can specify that are atomic:

* Reads and writes are atomic for reference variables and for most primitive variables (all types except long and double).
* Reads and writes are atomic for *all* variables declared volatile (*including* long and double variables).

Atomic actions cannot be interleaved, so they can be used without fear of thread interference. However, this does not eliminate all need to synchronize atomic actions, because memory consistency errors are still possible. Using volatile variables reduces the risk of memory consistency errors, because any write to a volatile variable establishes a happens-before relationship with subsequent reads of that same variable. This means that changes to a volatile variable are always visible to other threads. What's more, it also means that when a thread reads a volatile variable, it sees not just the latest change to the volatile, but also the side effects of the code that led up the change.

Using simple atomic variable access is more efficient than accessing these variables through synchronized code, but requires more care by the programmer to avoid memory consistency errors. Whether the extra effort is worthwhile depends on the size and complexity of the application.

Some of the classes in the [java.util.concurrent](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html) package provide atomic methods that do not rely on synchronization. We'll discuss them in the section on [High Level Concurrency Objects](https://docs.oracle.com/javase/tutorial/essential/concurrency/highlevel.html).

# Liveness

A concurrent application's ability to execute in a timely manner is known as its *liveness*. This section describes the most common kind of liveness problem, [deadlock](https://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html), and goes on to briefly describe two other liveness problems, [starvation and livelock](https://docs.oracle.com/javase/tutorial/essential/concurrency/starvelive.html).

# Deadlock

*Deadlock* describes a situation where two or more threads are blocked forever, waiting for each other. Here's an example.

Alphonse and Gaston are friends, and great believers in courtesy. A strict rule of courtesy is that when you bow to a friend, you must remain bowed until your friend has a chance to return the bow. Unfortunately, this rule does not account for the possibility that two friends might bow to each other at the same time. This example application, [Deadlock](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/Deadlock.java), models this possibility:

public class Deadlock {

static class Friend {

private final String name;

public Friend(String name) {

this.name = name;

}

public String getName() {

return this.name;

}

public synchronized void bow(Friend bower) {

System.out.format("%s: %s"

+ " has bowed to me!%n",

this.name, bower.getName());

bower.bowBack(this);

}

public synchronized void bowBack(Friend bower) {

System.out.format("%s: %s"

+ " has bowed back to me!%n",

this.name, bower.getName());

}

}

public static void main(String[] args) {

final Friend alphonse =

new Friend("Alphonse");

final Friend gaston =

new Friend("Gaston");

new Thread(new Runnable() {

public void run() { alphonse.bow(gaston); }

}).start();

new Thread(new Runnable() {

public void run() { gaston.bow(alphonse); }

}).start();

}

}

When Deadlock runs, it's extremely likely that both threads will block when they attempt to invoke bowBack. Neither block will ever end, because each thread is waiting for the other to exit bow.

# Starvation and Livelock

Starvation and livelock are much less common a problem than deadlock, but are still problems that every designer of concurrent software is likely to encounter.

## Starvation

*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. For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

## Livelock

A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then *livelock* may result. As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked — they are simply too busy responding to each other to resume work. This is comparable to two people attempting to pass each other in a corridor: Alphonse moves to his left to let Gaston pass, while Gaston moves to his right to let Alphonse pass. Seeing that they are still blocking each other, Alphone moves to his right, while Gaston moves to his left. They're still blocking each other, so...

# Guarded Blocks

Threads often have to coordinate their actions. The most common coordination idiom is the *guarded block*. Such a block begins by polling a condition that must be true before the block can proceed. There are a number of steps to follow in order to do this correctly.

Suppose, for example guardedJoy is a method that must not proceed until a shared variable joy has been set by another thread. Such a method could, in theory, simply loop until the condition is satisfied, but that loop is wasteful, since it executes continuously while waiting.

public void guardedJoy() {

// Simple loop guard. Wastes

// processor time. Don't do this!

while(!joy) {}

System.out.println("Joy has been achieved!");

}

A more efficient guard invokes [Object.wait](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#wait--) to suspend the current thread. The invocation of wait does not return until another thread has issued a notification that some special event may have occurred — though not necessarily the event this thread is waiting for:

public synchronized void guardedJoy() {

// This guard only loops once for each special event, which may not

// be the event we're waiting for.

while(!joy) {

try {

wait();

} catch (InterruptedException e) {}

}

System.out.println("Joy and efficiency have been achieved!");

}

**Note:** Always invoke wait inside a loop that tests for the condition being waited for. Don't assume that the interrupt was for the particular condition you were waiting for, or that the condition is still true.

Like many methods that suspend execution, wait can throw InterruptedException. In this example, we can just ignore that exception — we only care about the value of joy.

Why is this version of guardedJoy synchronized? Suppose d is the object we're using to invoke wait. When a thread invokes d.wait, it must own the intrinsic lock for d — otherwise an error is thrown. Invoking wait inside a synchronized method is a simple way to acquire the intrinsic lock.

When wait is invoked, the thread releases the lock and suspends execution. At some future time, another thread will acquire the same lock and invoke [Object.notifyAll](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#notifyAll--), informing all threads waiting on that lock that something important has happened:

public synchronized notifyJoy() {

joy = true;

notifyAll();

}

Some time after the second thread has released the lock, the first thread reacquires the lock and resumes by returning from the invocation of wait.

**Note:** There is a second notification method, notify, which wakes up a single thread. Because notify doesn't allow you to specify the thread that is woken up, it is useful only in massively parallel applications — that is, programs with a large number of threads, all doing similar chores. In such an application, you don't care which thread gets woken up.

Let's use guarded blocks to create a *Producer-Consumer* application. This kind of application shares data between two threads: the *producer*, that creates the data, and the *consumer*, that does something with it. The two threads communicate using a shared object. Coordination is essential: the consumer thread must not attempt to retrieve the data before the producer thread has delivered it, and the producer thread must not attempt to deliver new data if the consumer hasn't retrieved the old data.

In this example, the data is a series of text messages, which are shared through an object of type [Drop](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/Drop.java):

public class Drop {

// Message sent from producer

// to consumer.

private String message;

// True if consumer should wait

// for producer to send message,

// false if producer should wait for

// consumer to retrieve message.

private boolean empty = true;

public synchronized String take() {

// Wait until message is

// available.

while (empty) {

try {

wait();

} catch (InterruptedException e) {}

}

// Toggle status.

empty = true;

// Notify producer that

// status has changed.

notifyAll();

return message;

}

public synchronized void put(String message) {

// Wait until message has

// been retrieved.

while (!empty) {

try {

wait();

} catch (InterruptedException e) {}

}

// Toggle status.

empty = false;

// Store message.

this.message = message;

// Notify consumer that status

// has changed.

notifyAll();

}

}

The producer thread, defined in [Producer](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/Producer.java), sends a series of familiar messages. The string "DONE" indicates that all messages have been sent. To simulate the unpredictable nature of real-world applications, the producer thread pauses for random intervals between messages.

import java.util.Random;

public class Producer implements Runnable {

private Drop drop;

public Producer(Drop drop) {

this.drop = drop;

}

public void run() {

String importantInfo[] = {

"Mares eat oats",

"Does eat oats",

"Little lambs eat ivy",

"A kid will eat ivy too"

};

Random random = new Random();

for (int i = 0;

i < importantInfo.length;

i++) {

drop.put(importantInfo[i]);

try {

Thread.sleep(random.nextInt(5000));

} catch (InterruptedException e) {}

}

drop.put("DONE");

}

}

The consumer thread, defined in [Consumer](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/Consumer.java), simply retrieves the messages and prints them out, until it retrieves the "DONE" string. This thread also pauses for random intervals.

import java.util.Random;

public class Consumer implements Runnable {

private Drop drop;

public Consumer(Drop drop) {

this.drop = drop;

}

public void run() {

Random random = new Random();

for (String message = drop.take();

! message.equals("DONE");

message = drop.take()) {

System.out.format("MESSAGE RECEIVED: %s%n", message);

try {

Thread.sleep(random.nextInt(5000));

} catch (InterruptedException e) {}

}

}

}

Finally, here is the main thread, defined in [ProducerConsumerExample](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/ProducerConsumerExample.java), that launches the producer and consumer threads.

public class ProducerConsumerExample {

public static void main(String[] args) {

Drop drop = new Drop();

(new Thread(new Producer(drop))).start();

(new Thread(new Consumer(drop))).start();

}

}

**Note:** The Drop class was written in order to demonstrate guarded blocks. To avoid re-inventing the wheel, examine the existing data structures in the [Java Collections Framework](https://docs.oracle.com/javase/tutorial/collections/index.html) before trying to code your own data-sharing objects. For more information, refer to the [Questions and Exercises](https://docs.oracle.com/javase/tutorial/essential/concurrency/QandE/questions.html) section.

# Immutable Objects

An object is considered *immutable* if its state cannot change after it is constructed. Maximum reliance on immutable objects is widely accepted as a sound strategy for creating simple, reliable code.

Immutable objects are particularly useful in concurrent applications. Since they cannot change state, they cannot be corrupted by thread interference or observed in an inconsistent state.

Programmers are often reluctant to employ immutable objects, because they worry about the cost of creating a new object as opposed to updating an object in place. The impact of object creation is often overestimated, and can be offset by some of the efficiencies associated with immutable objects. These include decreased overhead due to garbage collection, and the elimination of code needed to protect mutable objects from corruption.

The following subsections take a class whose instances are mutable and derives a class with immutable instances from it. In so doing, they give general rules for this kind of conversion and demonstrate some of the advantages of immutable objects.

# A Synchronized Class Example

The class, [SynchronizedRGB](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/SynchronizedRGB.java), defines objects that represent colors. Each object represents the color as three integers that stand for primary color values and a string that gives the name of the color.

public class SynchronizedRGB {

// Values must be between 0 and 255.

private int red;

private int green;

private int blue;

private String name;

private void check(int red,

int green,

int blue) {

if (red < 0 || red > 255

|| green < 0 || green > 255

|| blue < 0 || blue > 255) {

throw new IllegalArgumentException();

}

}

public SynchronizedRGB(int red,

int green,

int blue,

String name) {

check(red, green, blue);

this.red = red;

this.green = green;

this.blue = blue;

this.name = name;

}

public void set(int red,

int green,

int blue,

String name) {

check(red, green, blue);

synchronized (this) {

this.red = red;

this.green = green;

this.blue = blue;

this.name = name;

}

}

public synchronized int getRGB() {

return ((red << 16) | (green << 8) | blue);

}

public synchronized String getName() {

return name;

}

public synchronized void invert() {

red = 255 - red;

green = 255 - green;

blue = 255 - blue;

name = "Inverse of " + name;

}

}

SynchronizedRGB must be used carefully to avoid being seen in an inconsistent state. Suppose, for example, a thread executes the following code:

SynchronizedRGB color =

new SynchronizedRGB(0, 0, 0, "Pitch Black");

...

int myColorInt = color.getRGB(); //Statement 1

String myColorName = color.getName(); //Statement 2

If another thread invokes color.set after Statement 1 but before Statement 2, the value of myColorInt won't match the value of myColorName. To avoid this outcome, the two statements must be bound together:

synchronized (color) {

int myColorInt = color.getRGB();

String myColorName = color.getName();

}

This kind of inconsistency is only possible for mutable objects — it will not be an issue for the immutable version of SynchronizedRGB.

**A Strategy for Defining Immutable Objects**

The following rules define a simple strategy for creating immutable objects. Not all classes documented as "immutable" follow these rules. This does not necessarily mean the creators of these classes were sloppy — they may have good reason for believing that instances of their classes never change after construction. However, such strategies require sophisticated analysis and are not for beginners.

1. Don't provide "setter" methods — methods that modify fields or objects referred to by fields.
2. Make all fields final and private.
3. Don't allow subclasses to override methods. The simplest way to do this is to declare the class as final. A more sophisticated approach is to make the constructor private and construct instances in factory methods.
4. If the instance fields include references to mutable objects, don't allow those objects to be changed:
   * Don't provide methods that modify the mutable objects.
   * Don't share references to the mutable objects. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.

Applying this strategy to SynchronizedRGB results in the following steps:

1. There are two setter methods in this class. The first one, set, arbitrarily transforms the object, and has no place in an immutable version of the class. The second one, invert, can be adapted by having it create a new object instead of modifying the existing one.
2. All fields are already private; they are further qualified as final.
3. The class itself is declared final.
4. Only one field refers to an object, and that object is itself immutable. Therefore, no safeguards against changing the state of "contained" mutable objects are necessary.

After these changes, we have [ImmutableRGB](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/ImmutableRGB.java):

final public class ImmutableRGB {

// Values must be between 0 and 255.

final private int red;

final private int green;

final private int blue;

final private String name;

private void check(int red,

int green,

int blue) {

if (red < 0 || red > 255

|| green < 0 || green > 255

|| blue < 0 || blue > 255) {

throw new IllegalArgumentException();

}

}

public ImmutableRGB(int red,

int green,

int blue,

String name) {

check(red, green, blue);

this.red = red;

this.green = green;

this.blue = blue;

this.name = name;

}

public int getRGB() {

return ((red << 16) | (green << 8) | blue);

}

public String getName() {

return name;

}

public ImmutableRGB invert() {

return new ImmutableRGB(255 - red,

255 - green,

255 - blue,

"Inverse of " + name);

}

}

# High Level Concurrency Objects

So far, this lesson has focused on the low-level APIs that have been part of the Java platform from the very beginning. These APIs are adequate for very basic tasks, but higher-level building blocks are needed for more advanced tasks. This is especially true for massively concurrent applications that fully exploit today's multiprocessor and multi-core systems.

In this section we'll look at some of the high-level concurrency features introduced with version 5.0 of the Java platform. Most of these features are implemented in the new java.util.concurrent packages. There are also new concurrent data structures in the Java Collections Framework.

* [Lock objects](https://docs.oracle.com/javase/tutorial/essential/concurrency/newlocks.html) support locking idioms that simplify many concurrent applications.
* [Executors](https://docs.oracle.com/javase/tutorial/essential/concurrency/executors.html) define a high-level API for launching and managing threads. Executor implementations provided by java.util.concurrent provide thread pool management suitable for large-scale applications.
* [Concurrent collections](https://docs.oracle.com/javase/tutorial/essential/concurrency/collections.html) make it easier to manage large collections of data, and can greatly reduce the need for synchronization.
* [Atomic variables](https://docs.oracle.com/javase/tutorial/essential/concurrency/atomicvars.html) have features that minimize synchronization and help avoid memory consistency errors.
* [ThreadLocalRandom](https://docs.oracle.com/javase/tutorial/essential/concurrency/threadlocalrandom.html) (in JDK 7) provides efficient generation of pseudorandom numbers from multiple threads.

# Lock Objects

Synchronized code relies on a simple kind of reentrant lock. This kind of lock is easy to use, but has many limitations. More sophisticated locking idioms are supported by the [java.util.concurrent.locks](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/package-summary.html) package. We won't examine this package in detail, but instead will focus on its most basic interface, [Lock](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Lock.html).

Lock objects work very much like the implicit locks used by synchronized code. As with implicit locks, only one thread can own a Lock object at a time. Lock objects also support a wait/notify mechanism, through their associated [Condition](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Condition.html) objects.

The biggest advantage of Lock objects over implicit locks is their ability to back out of an attempt to acquire a lock. The tryLock method backs out if the lock is not available immediately or before a timeout expires (if specified). The lockInterruptibly method backs out if another thread sends an interrupt before the lock is acquired.

Let's use Lock objects to solve the deadlock problem we saw in [Liveness](https://docs.oracle.com/javase/tutorial/essential/concurrency/liveness.html). Alphonse and Gaston have trained themselves to notice when a friend is about to bow. We model this improvement by requiring that our Friend objects must acquire locks for *both* participants before proceeding with the bow. Here is the source code for the improved model, [Safelock](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/Safelock.java). To demonstrate the versatility of this idiom, we assume that Alphonse and Gaston are so infatuated with their newfound ability to bow safely that they can't stop bowing to each other:

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

import java.util.Random;

public class Safelock {

static class Friend {

private final String name;

private final Lock lock = new ReentrantLock();

public Friend(String name) {

this.name = name;

}

public String getName() {

return this.name;

}

public boolean impendingBow(Friend bower) {

Boolean myLock = false;

Boolean yourLock = false;

try {

myLock = lock.tryLock();

yourLock = bower.lock.tryLock();

} finally {

if (! (myLock && yourLock)) {

if (myLock) {

lock.unlock();

}

if (yourLock) {

bower.lock.unlock();

}

}

}

return myLock && yourLock;

}

public void bow(Friend bower) {

if (impendingBow(bower)) {

try {

System.out.format("%s: %s has"

+ " bowed to me!%n",

this.name, bower.getName());

bower.bowBack(this);

} finally {

lock.unlock();

bower.lock.unlock();

}

} else {

System.out.format("%s: %s started"

+ " to bow to me, but saw that"

+ " I was already bowing to"

+ " him.%n",

this.name, bower.getName());

}

}

public void bowBack(Friend bower) {

System.out.format("%s: %s has" +

" bowed back to me!%n",

this.name, bower.getName());

}

}

static class BowLoop implements Runnable {

private Friend bower;

private Friend bowee;

public BowLoop(Friend bower, Friend bowee) {

this.bower = bower;

this.bowee = bowee;

}

public void run() {

Random random = new Random();

for (;;) {

try {

Thread.sleep(random.nextInt(10));

} catch (InterruptedException e) {}

bowee.bow(bower);

}

}

}

public static void main(String[] args) {

final Friend alphonse =

new Friend("Alphonse");

final Friend gaston =

new Friend("Gaston");

new Thread(new BowLoop(alphonse, gaston)).start();

new Thread(new BowLoop(gaston, alphonse)).start();

}

}

# Executors

In all of the previous examples, there's a close connection between the task being done by a new thread, as defined by its Runnable object, and the thread itself, as defined by a Thread object. This works well for small applications, but in large-scale applications, it makes sense to separate thread management and creation from the rest of the application. Objects that encapsulate these functions are known as *executors*. The following subsections describe executors in detail.

* [Executor Interfaces](https://docs.oracle.com/javase/tutorial/essential/concurrency/exinter.html) define the three executor object types.
* [Thread Pools](https://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html) are the most common kind of executor implementation.
* [Fork/Join](https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html) is a framework (new in JDK 7) for taking advantage of multiple processors.

# Executor Interfaces

The java.util.concurrent package defines three executor interfaces:

* Executor, a simple interface that supports launching new tasks.
* ExecutorService, a subinterface of Executor, which adds features that help manage the lifecycle, both of the individual tasks and of the executor itself.
* ScheduledExecutorService, a subinterface of ExecutorService, supports future and/or periodic execution of tasks.

Typically, variables that refer to executor objects are declared as one of these three interface types, not with an executor class type.

## The Executor Interface

The [Executor](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executor.html) interface provides a single method, execute, designed to be a drop-in replacement for a common thread-creation idiom. If r is a Runnable object, and e is an Executor object you can replace

(new Thread(r)).start();

with

e.execute(r);

However, the definition of execute is less specific. The low-level idiom creates a new thread and launches it immediately. Depending on the Executor implementation, execute may do the same thing, but is more likely to use an existing worker thread to run r, or to place r in a queue to wait for a worker thread to become available. (We'll describe worker threads in the section on [Thread Pools](https://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html).)

The executor implementations in java.util.concurrent are designed to make full use of the more advanced ExecutorService and ScheduledExecutorService interfaces, although they also work with the base Executor interface.

## The ExecutorService Interface

The [ExecutorService](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html) interface supplements execute with a similar, but more versatile submit method. Like execute, submit accepts Runnable objects, but also accepts [Callable](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html) objects, which allow the task to return a value. The submit method returns a [Future](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html) object, which is used to retrieve the Callable return value and to manage the status of both Callable and Runnable tasks.

ExecutorService also provides methods for submitting large collections of Callable objects. Finally, ExecutorService provides a number of methods for managing the shutdown of the executor. To support immediate shutdown, tasks should handle [interrupts](https://docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html) correctly.

## The ScheduledExecutorService Interface

The [ScheduledExecutorService](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ScheduledExecutorService.html) interface supplements the methods of its parent ExecutorService with schedule, which executes a Runnable or Callable task after a specified delay. In addition, the interface defines scheduleAtFixedRate and scheduleWithFixedDelay, which executes specified tasks repeatedly, at defined intervals.

**Thread Pools**

Most of the executor implementations in java.util.concurrent use *thread pools*, which consist of *worker threads*. This kind of thread exists separately from the Runnable and Callable tasks it executes and is often used to execute multiple tasks.

Using worker threads minimizes the overhead due to thread creation. Thread objects use a significant amount of memory, and in a large-scale application, allocating and deallocating many thread objects creates a significant memory management overhead.

One common type of thread pool is the *fixed thread pool*. This type of pool always has a specified number of threads running; if a thread is somehow terminated while it is still in use, it is automatically replaced with a new thread. Tasks are submitted to the pool via an internal queue, which holds extra tasks whenever there are more active tasks than threads.

An important advantage of the fixed thread pool is that applications using it *degrade gracefully*. To understand this, consider a web server application where each HTTP request is handled by a separate thread. If the application simply creates a new thread for every new HTTP request, and the system receives more requests than it can handle immediately, the application will suddenly stop responding to *all* requests when the overhead of all those threads exceed the capacity of the system. With a limit on the number of the threads that can be created, the application will not be servicing HTTP requests as quickly as they come in, but it will be servicing them as quickly as the system can sustain.

A simple way to create an executor that uses a fixed thread pool is to invoke the [newFixedThreadPool](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html#newFixedThreadPool-int-) factory method in [java.util.concurrent.Executors](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html) This class also provides the following factory methods:

* The [newCachedThreadPool](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html#newCachedThreadPool-int-) method creates an executor with an expandable thread pool. This executor is suitable for applications that launch many short-lived tasks.
* The [newSingleThreadExecutor](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html#newSingleThreadExecutor-int-) method creates an executor that executes a single task at a time.
* Several factory methods are ScheduledExecutorService versions of the above executors.

If none of the executors provided by the above factory methods meet your needs, constructing instances of [java.util.concurrent.ThreadPoolExecutor](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadPoolExecutor.html) or [java.util.concurrent.ScheduledThreadPoolExecutor](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ScheduledThreadPoolExecutor.html) will give you additional options.

# Fork/Join

The fork/join framework is an implementation of the ExecutorService interface that helps you take advantage of multiple processors. It is designed for work that can be broken into smaller pieces recursively. The goal is to use all the available processing power to enhance the performance of your application.

As with any ExecutorService implementation, the fork/join framework distributes tasks to worker threads in a thread pool. The fork/join framework is distinct because it uses a *work-stealing* algorithm. Worker threads that run out of things to do can steal tasks from other threads that are still busy.

The center of the fork/join framework is the [ForkJoinPool](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html) class, an extension of the AbstractExecutorService class. ForkJoinPool implements the core work-stealing algorithm and can execute [ForkJoinTask](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinTask.html) processes.

## Basic Use

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](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RecursiveTask.html) (which can return a result) or [RecursiveAction](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RecursiveAction.html).

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.

## Blurring for Clarity

To help you understand how the fork/join framework works, consider the following example. Suppose that you want to blur an image. The original *source* image is represented by an array of integers, where each integer contains the color values for a single pixel. The blurred *destination* image is also represented by an integer array with the same size as the source.

Performing the blur is accomplished by working through the source array one pixel at a time. Each pixel is averaged with its surrounding pixels (the red, green, and blue components are averaged), and the result is placed in the destination array. Since an image is a large array, this process can take a long time. You can take advantage of concurrent processing on multiprocessor systems by implementing the algorithm using the fork/join framework. Here is one possible implementation:

public class ForkBlur extends RecursiveAction {

private int[] mSource;

private int mStart;

private int mLength;

private int[] mDestination;

// Processing window size; should be odd.

private int mBlurWidth = 15;

public ForkBlur(int[] src, int start, int length, int[] dst) {

mSource = src;

mStart = start;

mLength = length;

mDestination = dst;

}

protected void computeDirectly() {

int sidePixels = (mBlurWidth - 1) / 2;

for (int index = mStart; index < mStart + mLength; index++) {

// Calculate average.

float rt = 0, gt = 0, bt = 0;

for (int mi = -sidePixels; mi <= sidePixels; mi++) {

int mindex = Math.min(Math.max(mi + index, 0),

mSource.length - 1);

int pixel = mSource[mindex];

rt += (float)((pixel & 0x00ff0000) >> 16)

/ mBlurWidth;

gt += (float)((pixel & 0x0000ff00) >> 8)

/ mBlurWidth;

bt += (float)((pixel & 0x000000ff) >> 0)

/ mBlurWidth;

}

// Reassemble destination pixel.

int dpixel = (0xff000000 ) |

(((int)rt) << 16) |

(((int)gt) << 8) |

(((int)bt) << 0);

mDestination[index] = dpixel;

}

}

...

Now you implement the abstract compute() method, which either performs the blur directly or splits it into two smaller tasks. A simple array length threshold helps determine whether the work is performed or split.

protected static int sThreshold = 100000;

protected void compute() {

if (mLength < sThreshold) {

computeDirectly();

return;

}

int split = mLength / 2;

invokeAll(new ForkBlur(mSource, mStart, split, mDestination),

new ForkBlur(mSource, mStart + split, mLength - split,

mDestination));

}

If the previous methods are in a subclass of the RecursiveAction class, then setting up the task to run in a ForkJoinPool is straightforward, and involves the following steps:

1. Create a task that represents all of the work to be done.
2. // source image pixels are in src
3. // destination image pixels are in dst
4. ForkBlur fb = new ForkBlur(src, 0, src.length, dst);
5. Create the ForkJoinPool that will run the task.
6. ForkJoinPool pool = new ForkJoinPool();
7. Run the task.
8. pool.invoke(fb);

For the full source code, including some extra code that creates the destination image file, see the [ForkBlur](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/ForkBlur.java) example.

## Standard Implementations

Besides using the fork/join framework to implement custom algorithms for tasks to be performed concurrently on a multiprocessor system (such as the ForkBlur.java example in the previous section), there are some generally useful features in Java SE which are already implemented using the fork/join framework. One such implementation, introduced in Java SE 8, is used by the [java.util.Arrays](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html) class for its parallelSort() methods. These methods are similar to sort(), but leverage concurrency via the fork/join framework. Parallel sorting of large arrays is faster than sequential sorting when run on multiprocessor systems. However, how exactly the fork/join framework is leveraged by these methods is outside the scope of the Java Tutorials. For this information, see the Java API documentation.

Another implementation of the fork/join framework is used by methods in the java.util.streams package, which is part of [Project Lambda](http://openjdk.java.net/projects/lambda/) scheduled for the Java SE 8 release. For more information, see the [Lambda Expressions](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html) section.

# Concurrent Collections

The java.util.concurrent package includes a number of additions to the Java Collections Framework. These are most easily categorized by the collection interfaces provided:

* [BlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html) defines a first-in-first-out data structure that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue.
* [ConcurrentMap](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentMap.html) is a subinterface of [java.util.Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html) that defines useful atomic operations. These operations remove or replace a key-value pair only if the key is present, or add a key-value pair only if the key is absent. Making these operations atomic helps avoid synchronization. The standard general-purpose implementation of ConcurrentMap is [ConcurrentHashMap](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html), which is a concurrent analog of [HashMap](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html).
* [ConcurrentNavigableMap](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentNavigableMap.html) is a subinterface of ConcurrentMap that supports approximate matches. The standard general-purpose implementation of ConcurrentNavigableMap is [ConcurrentSkipListMap](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentSkipListMap.html), which is a concurrent analog of [TreeMap](https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html).

All of these collections help avoid [Memory Consistency Errors](https://docs.oracle.com/javase/tutorial/essential/concurrency/memconsist.html) by defining a happens-before relationship between an operation that adds an object to the collection with subsequent operations that access or remove that object.

# Atomic Variables

The [java.util.concurrent.atomic](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html) package defines classes that support atomic operations on single variables. All classes have get and set methods that work like reads and writes on volatile variables. That is, a set has a happens-before relationship with any subsequent get on the same variable. The atomic compareAndSet method also has these memory consistency features, as do the simple atomic arithmetic methods that apply to integer atomic variables.

To see how this package might be used, let's return to the [Counter](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/Counter.java) class we originally used to demonstrate thread interference:

class Counter {

private int c = 0;

public void increment() {

c++;

}

public void decrement() {

c--;

}

public int value() {

return c;

}

}

One way to make Counter safe from thread interference is to make its methods synchronized, as in [SynchronizedCounter](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/SynchronizedCounter.java):

class SynchronizedCounter {

private int c = 0;

public synchronized void increment() {

c++;

}

public synchronized void decrement() {

c--;

}

public synchronized int value() {

return c;

}

}

For this simple class, synchronization is an acceptable solution. But for a more complicated class, we might want to avoid the liveness impact of unnecessary synchronization. Replacing the int field with an AtomicInteger allows us to prevent thread interference without resorting to synchronization, as in [AtomicCounter](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/AtomicCounter.java):

import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {

private AtomicInteger c = new AtomicInteger(0);

public void increment() {

c.incrementAndGet();

}

public void decrement() {

c.decrementAndGet();

}

public int value() {

return c.get();

}

}

# Concurrent Random Numbers

In JDK 7, [java.util.concurrent](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html) includes a convenience class, [ThreadLocalRandom](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadLocalRandom.html), for applications that expect to use random numbers from multiple threads or ForkJoinTasks.

For concurrent access, using ThreadLocalRandom instead of Math.random() results in less contention and, ultimately, better performance.

All you need to do is call ThreadLocalRandom.current(), then call one of its methods to retrieve a random number. Here is one example:

int r = ThreadLocalRandom.current() .nextInt(4, 77);

# For Further Reading

* *Concurrent Programming in Java: Design Principles and Pattern (2nd Edition)* by Doug Lea. A comprehensive work by a leading expert, who's also the architect of the Java platform's concurrency framework.
* *Java Concurrency in Practice* by Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. A practical guide designed to be accessible to the novice.
* *Effective Java Programming Language Guide (2nd Edition)* by Joshua Bloch. Though this is a general programming guide, its chapter on threads contains essential "best practices" for concurrent programming.
* *Concurrency: State Models & Java Programs (2nd Edition)*, by Jeff Magee and Jeff Kramer. An introduction to concurrent programming through a combination of modeling and practical examples.
* [*Java Concurrent Animated*](http://sourceforge.net/projects/javaconcurrenta/)*:* Animations that show usage of concurrency features.

# Questions and Exercises: Concurrency

## Questions

1. Can you pass a Thread object to Executor.execute? Would such an invocation make sense?

## Exercises

1. Compile and run [BadThreads.java](https://docs.oracle.com/javase/tutorial/essential/concurrency/QandE/BadThreads.java):
2. public class BadThreads {
3. static String message;
4. private static class CorrectorThread
5. extends Thread {
6. public void run() {
7. try {
8. sleep(1000);
9. } catch (InterruptedException e) {}
10. // Key statement 1:
11. message = "Mares do eat oats.";
12. }
13. }
14. public static void main(String args[])
15. throws InterruptedException {
16. (new CorrectorThread()).start();
17. message = "Mares do not eat oats.";
18. Thread.sleep(2000);
19. // Key statement 2:
20. System.out.println(message);
21. }
22. }

The application should print out "Mares do eat oats." Is it guaranteed to always do this? If not, why not? Would it help to change the parameters of the two invocations of Sleep? How would you guarantee that all changes to message will be visible in the main thread?

1. Modify the producer-consumer example in [Guarded Blocks](https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html) to use a standard library class instead of the Drop class.

# Answers to Questions and Exercises: Concurrency

## Questions

1. **Question:** Can you pass a Thread object to Executor.execute? Would such an invocation make sense? Why or why not?

**Answer:** Thread implements the Runnable interface, so you can pass an instance of Thread to Executor.execute. However it doesn't make sense to use Thread objects this way. If the object is directly instantiated from Thread, its run method doesn't do anything. You can define a subclass of Thread with a useful run method — but such a class would implement features that the executor would not use.

## Exercises

1. **Exercise:** Compile and run [BadThreads.java](https://docs.oracle.com/javase/tutorial/essential/concurrency/QandE/BadThreads.java):
2. public class BadThreads {
3. static String message;
4. private static class CorrectorThread
5. extends Thread {
6. public void run() {
7. try {
8. sleep(1000);
9. } catch (InterruptedException e) {}
10. // Key statement 1:
11. message = "Mares do eat oats.";
12. }
13. }
14. public static void main(String args[])
15. throws InterruptedException {
16. (new CorrectorThread()).start();
17. message = "Mares do not eat oats.";
18. Thread.sleep(2000);
19. // Key statement 2:
20. System.out.println(message);
21. }
22. }

The application should print out "Mares do eat oats." Is it guaranteed to always do this? If not, why not? Would it help to change the parameters of the two invocations of Sleep? How would you guarantee that all changes to message will be visible to the main thread?

**Solution:** The program will almost always print out "Mares do eat oats." However, this result is not guaranteed, because there is no happens-before relationship between "Key statement 1" and "Key statment 2". This is true even if "Key statement 1" actually executes before "Key statement 2" — remember, a happens-before relationship is about visibility, not sequence.

There are two ways you can guarantee that all changes to message will be visible to the main thread:

* + In the main thread, retain a reference to the CorrectorThread instance. Then invoke join on that instance before referring to message
  + Encapsulate message in an object with synchronized methods. Never reference message except through those methods.

Both of these techniques establish the necessary happens-before relationship, making changes to message visible.

A third technique is to simply declare message as volatile. This guarantees that any write to message (as in "Key statement 1") will have a happens-before relationship with any subsequent reads of message (as in "Key statement 2"). But it does not guarantee that "Key statement 1" will *literally* happen before "Key statement 2". They will *probably* happen in sequence, but because of scheduling uncertainities and the unknown granularity of sleep, this is not guaranteed.

Changing the arguments of the two sleep invocations does not help either, since this does nothing to guarantee a happens-before relationship.

1. **Exercise:** Modify the producer-consumer example in [Guarded Blocks](https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html) to use a standard library class instead of the Drop class.

**Solution:** The [java.util.concurrent.BlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html) interface defines a get method that blocks if the queue is empty, and a put methods that blocks if the queue is full. These are effectively the same operations defined by Drop — except that Drop is not a queue! However, there's another way of looking at Drop: it's a queue with a capacity of zero. Since there's no room in the queue for *any* elements, every get blocks until the corresponding take and every take blocks until the corresponding get. There is an implementation of BlockingQueue with precisely this behavior: [java.util.concurrent.SynchronousQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/SynchronousQueue.html).

BlockingQueue is almost a drop-in replacement for Drop. The main problem in [Producer](https://docs.oracle.com/javase/tutorial/essential/concurrency/QandE/Producer.java) is that with BlockingQueue, the put and get methods throw InterruptedException. This means that the existing try must be moved up a level:

import java.util.Random;

import java.util.concurrent.BlockingQueue;

public class Producer implements Runnable {

private BlockingQueue<String> drop;

public Producer(BlockingQueue<String> drop) {

this.drop = drop;

}

public void run() {

String importantInfo[] = {

"Mares eat oats",

"Does eat oats",

"Little lambs eat ivy",

"A kid will eat ivy too"

};

Random random = new Random();

try {

for (int i = 0;

i < importantInfo.length;

i++) {

drop.put(importantInfo[i]);

Thread.sleep(random.nextInt(5000));

}

drop.put("DONE");

} catch (InterruptedException e) {}

}

}

Similar changes are required for [Consumer](https://docs.oracle.com/javase/tutorial/essential/concurrency/QandE/Consumer.java):

import java.util.Random;

import java.util.concurrent.BlockingQueue;

public class Consumer implements Runnable {

private BlockingQueue<String> drop;

public Consumer(BlockingQueue<String> drop) {

this.drop = drop;

}

public void run() {

Random random = new Random();

try {

for (String message = drop.take();

! message.equals("DONE");

message = drop.take()) {

System.out.format("MESSAGE RECEIVED: %s%n",

message);

Thread.sleep(random.nextInt(5000));

}

} catch (InterruptedException e) {}

}

}

For [ProducerConsumerExample](https://docs.oracle.com/javase/tutorial/essential/concurrency/QandE/ProducerConsumerExample.java), we simply change the declaration for the drop object:

import java.util.concurrent.BlockingQueue;

import java.util.concurrent.SynchronousQueue;

public class ProducerConsumerExample {

public static void main(String[] args) {

BlockingQueue<String> drop =

new SynchronousQueue<String> ();

(new Thread(new Producer(drop))).start();

(new Thread(new Consumer(drop))).start();

}

}