

5th Edition  
Covers Java 5



# JAVA™

## IN A NUTSHELL

*A Desktop Quick Reference*

O'REILLY®

*David Flanagan*

# Threads and Concurrency

The Java platform has supported multithreaded or *concurrent* programming with the `Thread` class and `Runnable` interface since Java 1.0. Java 5.0 bolsters that support with a comprehensive set of new utilities for concurrent programming.

## Creating, Running, and Manipulating Threads

Java makes it easy to define and work with multiple threads of execution within a program. `java.lang.Thread` is the fundamental thread class in the Java API. There are two ways to define a thread. One is to subclass `Thread`, override the `run()` method and then instantiate your `Thread` subclass. The other is to define a class that implements the `Runnable` method (i.e., define a `run()` method) and then pass an instance of this `Runnable` object to the `Thread()` constructor. In either case, the result is a `Thread` object, where the `run()` method is the body of the thread. When you call the `start()` method of the `Thread` object, the interpreter creates a new thread to execute the `run()` method. This new thread continues to run until the `run()` method exits. Meanwhile, the original thread continues running itself, starting with the statement following the `start()` method. The following code demonstrates:

```
final List list; // Some long unsorted list of objects; initialized elsewhere

/** A Thread class for sorting a List in the background */
class BackgroundSorter extends Thread {
    List l;
    public BackgroundSorter(List l) { this.l = l; } // Constructor
    public void run() { Collections.sort(l); } // Thread body
}

// Create a BackgroundSorter thread
Thread sorter = new BackgroundSorter(list);
// Start it running; the new thread runs the run() method above while
// the original thread continues with whatever statement comes next.
sorter.start();

// Here's another way to define a similar thread
Thread t = new Thread(new Runnable() { // Create a new thread
    public void run() { Collections.sort(list); } // to sort the list of objects.
});
t.start(); // Start it running
```

### Thread lifecycle

A thread can be in one of six states. In Java 5.0, these states are represented by the `Thread.State` enumerated type, and the state of a thread can be queried with the `getState()` method. A listing of the `Thread.State` constants provides a good overview of the lifecycle of a thread:

#### NEW

The `Thread` has been created but its `start()` method has not yet been called. All threads start in this state.

## RUNNABLE

The thread is running or is available to run when the operating system schedules it.

## BLOCKED

The thread is not running because it is waiting to acquire a lock so that it can enter a synchronized method or block. We'll see more about synchronized methods and blocks later in this section.

## WAITING

The thread is not running because it has called `Object.wait()` or `Thread.join()`.

## TIMED\_WAITING

The thread is not running because it has called `Thread.sleep()` or has called `Object.wait()` or `Thread.join()` with a timeout value.

## TERMINATED

The thread has completed execution. Its `run()` method has exited normally or by throwing an exception.

## Thread priorities

Threads can run at different priority levels. A thread at a given priority level does not typically run unless no higher-priority threads are waiting to run. Here is some code you can use when working with thread priorities:

```
// Set a thread t to lower-than-normal priority
t.setPriority(Thread.NORM_PRIORITY-1);

// Set a thread to lower priority than the current thread
t.setPriority(Thread.currentThread().getPriority() - 1);

// Threads that don't pause for I/O should explicitly yield the CPU
// to give other threads with the same priority a chance to run.
Thread t = new Thread(new Runnable() {
    public void run() {
        for(int i = 0; i < data.length; i++) { // Loop through a bunch of data
            process(data[i]);                // Process it
            if ((i % 10) == 0)                // But after every 10 iterations,
                Thread.yield();              // pause to let other threads run.
        }
    }
});
```

## Handling uncaught exceptions

A thread terminates normally when it reaches the end of its `run()` method or when it executes a `return` statement in that method. A thread can also terminate by throwing an exception, however. When a thread exits in this way, the default behavior is to print the name of the thread, the type of the exception, the exception message, and a stack trace. In Java 5.0, you can install a custom handler for uncaught exceptions in a thread. For example:

```
// This thread just throws an exception
Thread t = new Thread() {
```

```

        public void run() {throw new UnsupportedOperationException();}
    };

    // Giving threads a name helps with debugging
    t.setName("My Broken Thread");

    // Here's a handler for the error.
    t.setUncaughtExceptionHandler(new Thread.UncaughtExceptionHandler() {
        public void uncaughtException(Thread t, Throwable e) {
            System.err.printf("Exception in thread %d '%s':" +
                "%s at line %d of %s%n",
                t.getId(), // Thread id
                t.getName(), // Thread name
                e.toString(), // Exception name and message
                e.getStackTrace()[0].getLineNumber(), // line #
                e.getStackTrace()[0].getFileName()); // filename
        }
    });

```

## Making a Thread Sleep

Often, threads are used to perform some kind of repetitive task at a fixed interval. This is particularly true when doing graphical programming that involves animation or similar effects. The key to doing this is making a thread *sleep*, or stop running, for a specified amount of time. This is done with the static `Thread.sleep()` method, or, in Java 5.0, with utility methods of enumerated constants of the `TimeUnit` class:

```

import static java.util.concurrent.TimeUnit.SECONDS; // utility class

public class Clock extends Thread {
    // This field is volatile because two different threads may access it
    volatile boolean keepRunning = true;

    public Clock() { // The constructor
        setDaemon(true); // Daemon thread: interpreter can exit while it runs
    }

    public void run() { // The body of the thread
        while(keepRunning) { // This thread runs until asked to stop
            long now = System.currentTimeMillis(); // Get current time
            System.out.printf("%tr%n", now); // Print it out
            try { Thread.sleep(1000); } // Wait 1000 milliseconds
            catch (InterruptedException e) { return; } // Quit on interrupt
        }
    }

    // Ask the thread to stop running. An alternative to interrupt().
    public void pleaseStop() { keepRunning = false; }

    // This method demonstrates how to use the Clock class
    public static void main(String[] args) {
        Clock c = new Clock(); // Create a Clock thread
        c.start(); // Start it
    }
}

```

```

        try { SECONDS.sleep(10); } // Wait 10 seconds
        catch(InterruptedException ignore) {} // Ignore interrupts
        // Now stop the clock thread. We could also use c.interrupt()
        c.pleaseStop();
    }
}

```

Notice the `pleaseStop()` method in this example: it is designed to stop the clock thread in a controlled way. The example is coded so that it can also be stopped by calling the `interrupt()` method it inherits from `Thread`. The `Thread` class defines a `stop()` method, but it is deprecated.

## Running and Scheduling Tasks

Java provides a number of ways to run tasks asynchronously or to schedule them for future execution without having to explicitly create `Thread` objects. The following sections illustrate the `Timer` class added in Java 1.3 and the executors framework of the Java 5.0 `java.util.concurrent` package.

### Scheduling tasks with Timer

Added in Java 1.3, the `java.util.Timer` and `java.util.TimerTask` classes make it easy to run repetitive tasks. Here is some code that behaves much like the `Clock` class shown earlier:

```

import java.util.*;

// Define the time-display task
TimerTask displayTime = new TimerTask() {
    public void run() { System.out.printf("%tr%n", System.currentTimeMillis()); }
};
// Create a timer object to run the task (and possibly others)
Timer timer = new Timer();
// Now schedule that task to be run every 1,000 milliseconds, starting now
timer.schedule(displayTime, 0, 1000);

// To stop the time-display task
displayTime.cancel();

```

### The Executor interface

In Java 5.0, the `java.util.concurrent` package includes the `Executor` interface. An `Executor` is an object that can execute a `Runnable` object. A user of an `Executor` often does not need to be aware of just how the `Executor` accomplishes this: it just needs to know that the `Runnable` will, at some point, run. `Executor` implementations can be created to use a number of different threading strategies, as the following code makes clear. (Note that this example also demonstrates the use of a `BlockingQueue`.)

```

import java.util.concurrent.*;

/** Execute a Runnable in the current thread. */
class CurrentThreadExecutor implements Executor {
    public void execute(Runnable r) { r.run(); }
}

```

```

}

/** Execute each Runnable using a newly created thread */
class NewThreadExecutor implements Executor {
    public void execute(Runnable r) { new Thread(r).start(); }
}

/**
 * Queue up the Runnables and execute them in order using a single thread
 * created for that purpose.
 */
class SingleThreadExecutor extends Thread implements Executor {
    BlockingQueue<Runnable> q = new LinkedBlockingQueue<Runnable>();

    public void execute(Runnable r) {
        // Don't execute the Runnable here; just put it on the queue.
        // Our queue is effectively unbounded, so this should never block.
        // Since it never blocks, it should never throw InterruptedException.
        try { q.put(r); }
        catch(InterruptedException never) { throw new AssertionError(never); }
    }

    // This is the body of the thread that actually executes the Runnables
    public void run() {
        for(;;) { // Loop forever
            try {
                Runnable r = q.take(); // Get next Runnable, or wait
                r.run(); // Run it!
            }
            catch(InterruptedException e) {
                // If interrupted, stop executing queued Runnables.
                return;
            }
        }
    }
}
}

```

These sample implementations help demonstrate how an `Executor` works and how it separates the notion of executing a task from the scheduling policy and threading details of the implementation. It is rarely necessary to actually implement your own `Executor`, however, since `java.util.concurrent` provides the flexible and powerful `ThreadPoolExecutor` class. This class is typically used via one of the static factory methods in the `Executors` class:

```

Executor oneThread = Executors.newSingleThreadExecutor(); // pool size of 1
Executor fixedPool = Executors.newFixedThreadPool(10); // 10 threads in pool
Executor unboundedPool = Executors.newCachedThreadPool(); // as many as needed

```

In addition to these convenient factory methods, you can also explicitly create a `ThreadPoolExecutor` if you want to specify a minimum and maximum size for the thread pool or want to specify the queue type (bounded, unbounded, priority-sorted, or synchronized, for example) to use for tasks that cannot immediately be run by a thread.

## ExecutorService

If you've looked at the signature for `ThreadPoolExecutor` or for the `Executors` factory methods cited above, you'll see that it is an `ExecutorService`. The `ExecutorService` interface extends `Executor` and adds the ability to execute `Callable` objects. `Callable` is something like a `Runnable`. Instead of encapsulating arbitrary code in a `run()` method, however, a `Callable` puts that code in a `call()` method. `call()` differs from `run()` in two important ways: it returns a result, and it is allowed to throw exceptions.

Because `call()` returns a result, the `Callable` interface takes the result type as a parameter. A time-consuming chunk of code that computes a large prime number, for example, could be wrapped in a `Callable<BigInteger>`:

```
import java.util.concurrent.*;
import java.math.BigInteger;
import java.util.Random;
import java.security.SecureRandom;

/** This is a Callable implementation for computing big primes. */
public class RandomPrimeSearch implements Callable<BigInteger> {
    static Random prng = new SecureRandom(); // self-seeding
    int n;
    public RandomPrimeSearch(int bitsize) { n = bitsize; }
    public BigInteger call() { return BigInteger.probablePrime(n, prng); }
}
```

You can invoke the `call()` method of any `Callable` object directly, of course, but to execute it using an `ExecutorService`, you pass it to the `submit()` method. Because `ExecutorService` implementations typically run tasks asynchronously, the `submit()` method cannot simply return the result of the `call()` method. Instead, `submit()` returns a `Future` object. A `Future` is simply the promise of a result some-time in the future. It is parameterized with the type of the result, as shown in this code snippet:

```
// Try to compute two primes at the same time
ExecutorService threadpool = Executors.newFixedThreadPool(2);
Future<BigInteger> p = threadpool.submit(new RandomPrimeSearch(512));
Future<BigInteger> q = threadpool.submit(new RandomPrimeSearch(512));
```

Once you have a `Future` object, what can you do with it? You can call `isDone()` to see if the `Callable` has finished running. You can call `cancel()` to cancel execution of the `Callable` and can call `isCancelled()` to see if the `Callable` was canceled before it completed. But most of the time, you simply call `get()` to get the result of the `call()` method. `get()` blocks, if necessary, to wait for the `call()` method to complete. Here is code you might use with the `Future` objects shown above:

```
BigInteger product = p.get().multiply(q.get());
```

Note that the `get()` method may throw an `ExecutionException`. Recall that `Callable.call()` can throw any kind of exception. If this happens, the `Future` wraps that exception in an `ExecutionException` and throws it from `get()`. Note that the `Future.isDone()` method considers a `Callable` to be “done,” even if the `call()` method terminated abnormally with an exception.

## ScheduledExecutorService

`ScheduledExecutorService` is an extension of `ExecutorService` that adds Timer-like scheduling capabilities. It allows you to schedule a `Runnable` or `Callable` to be executed once after a specified time delay or to schedule a `Runnable` for repeated execution. In each case, the result of scheduling a task for future execution is a `ScheduledFuture` object. This is simply a `Future` that also implements the `Delay` interface and provides a `getDelay()` method that can be used to query the remaining time before execution of the task.

The easiest way to obtain a `ScheduledExecutorService` is with factory methods of the `Executors` class. The following code uses a `ScheduledExecutorService` to repeatedly perform an action and also to cancel the repeated action after a fixed interval.

```
/**
 * Print random ASCII characters at a rate of cps characters per second
 * for a total of totalSeconds seconds.
 */
public static void spew(int cps, int totalSeconds) {
    final Random rng = new Random(System.currentTimeMillis());
    final ScheduledExecutorService executor =
        Executors.newSingleThreadScheduledExecutor();
    final ScheduledFuture<?> spewer =
        executor.scheduleAtFixedRate(new Runnable() {
            public void run() {
                System.out.print((char)(rng.nextInt('~' - ' ') + ' '));
                System.out.flush();
            }
        },
        0, 1000000/cps, TimeUnit.MICROSECONDS);
    executor.schedule(new Runnable() {
        public void run() {
            spewer.cancel(false);
            executor.shutdown();
            System.out.println();
        }
    },
    totalSeconds, TimeUnit.SECONDS);
}
```

## Exclusion and Locks

When using multiple threads, you must be very careful if you allow more than one thread to access the same data structure. Consider what would happen if one thread was trying to loop through the elements of a `List` while another thread was sorting those elements. Preventing this kind of unwanted concurrency is one of the central problems of multithreaded computing. The basic technique for preventing two threads from accessing the same object at the same time is to require a thread to obtain a lock on the object before the thread can modify it. While any one thread holds the lock, another thread that requests the lock has to wait until the first thread is done and releases the lock. Every Java object has the fundamental ability to provide such a locking capability.

The easiest way to keep objects threadsafe is to declare all sensitive methods synchronized. A thread must obtain a lock on an object before it can execute any of its synchronized methods, which means that no other thread can execute any other synchronized method at the same time. (If a static method is declared synchronized, the thread must obtain a lock on the class, and this works in the same manner.) To do finer-grained locking, you can specify synchronized blocks of code that hold a lock on a specified object for a short time:

```
// This method swaps two array elements in a synchronized block
public static void swap(Object[] array, int index1, int index2) {
    synchronized(array) {
        Object tmp = array[index1];
        array[index1] = array[index2];
        array[index2] = tmp;
    }
}

// The Collection, Set, List, and Map implementations in java.util do
// not have synchronized methods (except for the legacy implementations
// Vector and Hashtable). When working with multiple threads, you can
// obtain synchronized wrapper objects.
List synclist = Collections.synchronizedList(list);
Map syncmap = Collections.synchronizedMap(map);
```

### The `java.util.concurrent.locks` package

Note that when you use the `synchronized` modifier or statement, the lock you acquire is block-scoped, and is automatically released when the thread exits the method or block. The `java.util.concurrent.locks` package in Java 5.0 provides an alternative: a `Lock` object that you explicitly lock and unlock. `Lock` objects are not automatically block-scoped and you must be careful to use `try/finally` constructs to ensure that locks are always released. On the other hand, `Lock` enables algorithms that are simply not possible with block-scoped locks, such as the following “hand-over-hand” linked list traversal:

```
import java.util.concurrent.locks.*; // New in Java 5.0

/**
 * A partial implementation of a linked list of values of type E.
 * It demonstrates hand-over-hand locking with Lock
 */
public class LinkedList<E> {
    E value; // The value of this node of the list
    LinkedList<E> rest; // The rest of the list
    Lock lock; // A lock for this node

    public LinkedList(E value) { // Constructor for a list
        this.value = value; // Node value
        rest = null; // This is the only node in the list
        lock = new ReentrantLock(); // We can lock this node
    }

    /**
```

```

* Append a node to the end of the list, traversing the list using
* hand-over-hand locking. This method is threadsafe: multiple threads
* may traverse different portions of the list at the same time.
**/
public void append(E value) {
    LinkedList<E> node = this; // Start at this node
    node.lock.lock();        // Lock it.

    // Loop 'till we find the last node in the list
    while(node.rest != null) {
        LinkedList<E> next = node.rest;

        // This is the hand-over-hand part. Lock the next node and then
        // unlock the current node. We use a try/finally construct so
        // that the current node is unlocked even if the lock on the
        // next node fails with an exception.
        try { next.lock.lock(); } // lock the next node
        finally { node.lock.unlock(); } // unlock the current node
        node = next;
    }

    // At this point, node is the final node in the list, and we have
    // a lock on it. Use a try/finally to ensure that we unlock it.
    try {
        node.rest = new LinkedList<E>(value); // Append new node
    }
    finally { node.lock.unlock(); }
}
}

```

## Deadlock

When you are using locking to prevent threads from accessing the same data at the same time, you must be careful to avoid *deadlock*, which occurs when two threads end up waiting for each other to release a lock they need. Since neither can proceed, neither one can release the lock it holds, and they both stop running. The following code is prone to deadlock. Whether or not a deadlock actually occurs may vary from system to system and from execution to execution.

```

// When two threads try to lock two objects, deadlock can occur unless
// they always request the locks in the same order.
final Object resource1 = new Object(); // Here are two objects to lock
final Object resource2 = new Object();
Thread t1 = new Thread(new Runnable() { // Locks resource1 then resource2
    public void run() {
        synchronized(resource1) {
            synchronized(resource2) { compute(); }
        }
    }
});

Thread t2 = new Thread(new Runnable() { // Locks resource2 then resource1
    public void run() {
        synchronized(resource2) {

```

```

        synchronized(resource1) { compute(); }
    }
}
});

t1.start(); // Locks resource1
t2.start(); // Locks resource2 and now neither thread can progress!

```

## Coordinating Threads

It is common in multithreaded programming to require one thread to wait for another thread to take some action. The Java platform provides a number of ways to coordinate threads, including methods built into the `Object` and `Thread` classes, as well as “synchronizer” utility classes introduced in Java 5.0.

### `wait()` and `notify()`

Sometimes a thread needs to stop running and wait until some kind of event occurs, at which point it is told to continue running. This is done with the `wait()` and `notify()` methods. These aren’t methods of the `Thread` class, however; they are methods of `Object`. Just as every Java object has a lock associated with it, every object can maintain a list of waiting threads. When a thread calls the `wait()` method of an object, any locks the thread holds are temporarily released, and the thread is added to the list of waiting threads for that object and stops running. When another thread calls the `notifyAll()` method of the same object, the object wakes up the waiting threads and allows them to continue running:

```

import java.util.*;

/**
 * A queue. One thread calls push() to put an object on the queue.
 * Another calls pop() to get an object off the queue. If there is no
 * data, pop() waits until there is some, using wait()/notify().
 * wait() and notify() must be used within a synchronized method or
 * block. In Java 5.0, use a java.util.concurrent.BlockingQueue instead.
 */
public class WaitingQueue<E> {
    LinkedList<E> q = new LinkedList<E>(); // Where objects are stored
    public synchronized void push(E o) {
        q.add(o); // Append the object to the end of the list
        this.notifyAll(); // Tell waiting threads that data is ready
    }
    public synchronized E pop() {
        while(q.size() == 0) {
            try { this.wait(); }
            catch (InterruptedException ignore) {}
        }
        return q.remove(0);
    }
}

```

Note that such a class is not necessary in Java 5.0 because `java.util.concurrent` defines the `BlockingQueue` interface and general-purpose implementations such as `ArrayBlockingQueue`.

### Waiting on a Condition

Java 5.0 provides an alternative to the `wait()` and `notifyAll()` methods of `Object`. `java.util.concurrent.locks` defines a `Condition` object with `await()` and `signalAll()` methods. `Condition` objects are always associated with `Lock` objects and are used in much the same way as the locking and waiting capability built into each Java object. The primary benefit is that it is possible to have more than one `Condition` for each `Lock`, something that is not possible with `Object`-based locking and waiting.

### Waiting for a thread to finish

Sometimes one thread needs to stop and wait for another thread to complete. You can accomplish this with the `join()` method:

```
List list; // A long list of objects to be sorted; initialized elsewhere

// Define a thread to sort the list: lower its priority, so it runs only
// when the current thread is waiting for I/O and then start it running.
Thread sorter = new BackgroundSorter(list); // Defined earlier
sorter.setPriority(Thread.currentThread.getPriority()-1); // Lower priority
sorter.start(); // Start sorting

// Meanwhile, in this original thread, read data from a file
byte[] data = readData(); // Method defined elsewhere

// Before we can proceed, we need the list to be fully sorted, so
// we must wait for the sorter thread to exit, if it hasn't already.
try { sorter.join(); } catch(InterruptedException e) {}
```

### Synchronizer utilities

`java.util.concurrent` includes four “synchronizer” classes that help to synchronize the state of a concurrent program by making threads wait until certain conditions hold:

#### Semaphore

The `Semaphore` class models semaphores, a traditional concurrent programming construct. Conceptually, a semaphore represents one or more “permits.” A thread that needs a permit calls `acquire()` and then calls `release()` when done with it. `acquire()` blocks if no permits are available, suspending the thread until another thread releases a permit.

#### CountDownLatch

A *latch* is conceptually any variable or concurrency construct that has two possible states and transitions from its initial state to its final state only once. Once the transition occurs, it remains in that final state forever. `CountDownLatch` is a concurrency utility that can exist in two states, closed and open. In its initial closed state, any threads that call the `await()` method

block and cannot proceed until it transitions to its latched open state. Once this transition occurs, all waiting threads proceed, and any threads that call `await()` in the future will not block at all. The transition from closed to open occurs when a specified number of calls to `countDown()` have occurred.

### Exchanger

An `Exchanger` is a utility that allows two threads to *rendezvous* and exchange values. The first thread to call the `exchange()` method blocks until a second thread calls the same method. When this happens, the argument passed to the `exchange()` method by the first thread becomes the return value of the method for the second thread and vice-versa. When the two `exchange()` invocations return, both threads are free to continue running concurrently. `Exchanger` is a generic type and uses its type parameter to specify the type of values to be exchanged.

### CyclicBarrier

A `CyclicBarrier` is a utility that enables a group of `N` threads to wait for each other to reach a synchronization point. The number of threads is specified when the `CyclicBarrier` is first created. Threads call the `await()` method to block until the last thread calls `await()`, at which point all threads resume again. Unlike a `CountDownLatch`, a `CyclicBarrier` resets its count and is ready for immediate reuse. `CyclicBarrier` is useful in parallel algorithms in which a computation is decomposed into parts, and each part is handled by a separate thread. In such algorithms, the threads must typically rendezvous so that their partial solutions can be merged into a complete solution. To facilitate this, the `CyclicBarrier` constructor allows you to specify a `Runnable` object to be executed by the last thread that calls `await()` before any of the other threads are woken up and allowed to resume. This `Runnable` can provide the coordination required to assemble a solution from the threads' computations or to assign a new computation to each of the threads.

## Thread Interruption

In the examples illustrating the `sleep()`, `join()`, and `wait()` methods, you may have noticed that calls to each of these methods are wrapped in a `try` statement that catches an `InterruptedException`. This is necessary because the `interrupt()` method allows one thread to interrupt the execution of another. The outcome of an interrupt depends on how you handle the `InterruptedException`. The response that is usually preferred is for an interrupted thread to stop running. On the other hand, if you simply catch and ignore the `InterruptedException`, an interrupt simply stops a thread from blocking.

If the `interrupt()` method is called on a thread that is not blocked, the thread continues running, but its “interrupt status” is set to indicate that an interrupt has been requested. A thread can test its own interrupt status by calling the static `Thread.interrupted()` method, which returns `true` if the thread has been interrupted and, as a side effect, clears the interrupt status. One thread can test the interrupt status of another thread with the instance method `isInterrupted()`, which queries the status but does not clear it.

If a thread calls `sleep()`, `join()`, or `wait()` while its interrupt status is set, it does not block but immediately throws an `InterruptedException` (the interrupt status is cleared as a side effect of throwing the exception). Similarly, if the `interrupt()` method is called on a thread that is already blocked in a call to `sleep()`, `join()`, or `wait()`, that thread stops blocking by throwing an `InterruptedException`.

One of the most common times that threads block is while doing input/output; a thread often has to pause and wait for data to become available from the file-system or from the network. (The `java.io`, `java.net`, and `java.nio` APIs for performing I/O operations are discussed later in this chapter.) Unfortunately, the `interrupt()` method does not wake up a thread blocked in an I/O method of the `java.io` package. This is one of the shortcomings of `java.io` that is cured by the New I/O API in `java.nio`. If a thread is interrupted while blocked in an I/O operation on any channel that implements `java.nio.channels.InterruptibleChannel`, the channel is closed, the thread's interrupt status is set, and the thread wakes up by throwing a `java.nio.channels.ClosedByInterruptException`. The same thing happens if a thread tries to call a blocking I/O method while its interrupt status is set. Similarly, if a thread is interrupted while it is blocked in the `select()` method of a `java.nio.channels.Selector` (or if it calls `select()` while its interrupt status is set), `select()` will stop blocking (or will never start) and will return immediately. No exception is thrown in this case; the interrupted thread simply wakes up, and the `select()` call returns.

## Blocking Queues

As noted in “The Queue and BlockingQueue Interfaces” earlier in this chapter, a *queue* is a collection in which elements are inserted at the “tail” and removed at the “head.” The `Queue` interface and various implementations were added to `java.util` as part of Java 5.0. `java.util.concurrent` extends the `Queue` interface: `BlockingQueue` defines `put()` and `take()` methods that allow you to add and remove elements of the queue, blocking if necessary until the queue has room, or until there is an element to be removed. The use of blocking queues is a common pattern in multithreaded programming: one thread produces objects and places them on a queue for consumption by another thread which removes them from the queue.

`java.util.concurrent` provides five implementations of `BlockingQueue`:

### `ArrayBlockingQueue`

This implementation is based on an array, and, like all arrays, has a fixed capacity established when it is created. At the cost of reduced throughput, this queue can operate in a “fair” mode in which threads blocking to `put()` or `take()` an element are served in the order in which they arrived.

### `LinkedBlockingQueue`

This implementation is based on a linked-list data structure. It may have a maximum size specified, but, by default, it is essentially unbounded.

### `PriorityBlockingQueue`

This unbounded queue does not implement FIFO (first-in, first-out) ordering. Instead, it orders its elements based on a specified `Comparator` object, or based on their natural ordering if they are `Comparable` objects and no `Comparator` is

specified. The element returned by `take()` is the smallest element according to the `Comparator` or `Comparable` ordering. See also `java.util.PriorityQueue` for a nonblocking version.

### DelayQueue

A `DelayQueue` is like a `PriorityBlockingQueue` for elements that implement the `Delayed` interface. `Delayed` is `Comparable` and orders elements by how long they are delayed. But `DelayQueue` is more than just an unbounded queue that sorts its elements. It also restricts `take()` and related methods so that elements cannot be removed from the queue until their delay has elapsed.

### SynchronousQueue

This class implements the degenerate case of a `BlockingQueue` with a capacity of zero. A call to `put()` blocks until some other thread calls `take()`, and a call to `take()` blocks until some other thread calls `put()`.

## Atomic Variables

The `java.util.concurrent.atomic` package contains utility classes that permit *atomic* operations on fields without locking. An atomic operation is one that is indivisible: no other thread can observe an atomic variable in the middle of an atomic operation on it. These utility classes define `get()` and `set()` accessor methods that have the properties of `volatile` fields but also define compound operations such as `compare-and-set` and `get-and-increment` that behave atomically. The code below demonstrates the use of `AtomicInteger` and contrasts it with the use of a traditional synchronized method:

```
// The count1(), count2() and count3() methods are all threadsafe. Two
// threads can call these methods at the same time, and they will never
// see the same return value.
public class Counters {
    // A counter using a synchronized method and locking
    int count1 = 0;
    public synchronized int count1() { return count1++; }

    // A counter using an atomic increment on an AtomicInteger
    AtomicInteger count2 = new AtomicInteger(0);
    public int count2() { return count2.getAndIncrement(); }

    // An optimistic counter using compareAndSet()
    AtomicInteger count3 = new AtomicInteger(0);
    public int count3() {
        // Get the counter value with get() and set it with compareAndSet().
        // If compareAndSet() returns false, try again until we get
        // through the loop without interference.
        int result;
        do {
            result = count3.get();
        } while(!count3.compareAndSet(result, result+1));
        return result;
    }
}
```

# Files and Directories

The `java.io.File` class represents a file or a directory and defines a number of important methods for manipulating files and directories. Note, however, that none of these methods allow you to read the contents of a file; that is the job of `java.io.FileInputStream`, which is just one of the many types of I/O streams used in Java and discussed in the next section. Here are some things you can do with `File`:

```
import java.io.*;
import java.util.*;

// Get the name of the user's home directory and represent it with a File
File homedir = new File(System.getProperty("user.home"));
// Create a File object to represent a file in that directory
File f = new File(homedir, ".configfile");

// Find out how big a file is and when it was last modified
long filelength = f.length();
Date lastModified = new java.util.Date(f.lastModified());

// If the file exists, is not a directory, and is readable,
// move it into a newly created directory.
if (f.exists() && f.isFile() && f.canRead()) { // Check config file
    File configdir = new File(homedir, ".configdir"); // A new config directory
    configdir.mkdir(); // Create that directory
    f.renameTo(new File(configdir, ".config")); // Move the file into it
}

// List all files in the home directory
String[] allfiles = homedir.list();

// List all files that have a ".java" suffix
String[] sourcecode = homedir.list(new FilenameFilter() {
    public boolean accept(File d, String name) { return name.endsWith(".java"); }
});
```

The `File` class gained some important additional functionality as of Java 1.2:

```
// List all filesystem root directories; on Windows, this gives us
// File objects for all drive letters (Java 1.2 and later).
File[] rootdirs = File.listRoots();

// Atomically, create a lock file, then delete it (Java 1.2 and later)
File lock = new File(configdir, ".lock");
if (lock.createNewFile()) {
    // We successfully created the file. Now arrange to delete it on exit
    lock.deleteOnExit();

    // Now run the application secure in the knowledge that no one else
    // is running it at the same time
    ...
}
else {
```

```

    // We didn't create the file; someone else has a lock
    System.err.println("Can't create lock file; exiting.");
    System.exit(1);
}

// Create a temporary file to use during processing (Java 1.2 and later)
File temp = File.createTempFile("app", ".tmp"); // Filename prefix and suffix
// Do something with the temp file
...
// And delete it when we're done
temp.delete();

```

## RandomAccessFile

The `java.io` package also defines a `RandomAccessFile` class that allows you to read binary data from arbitrary locations in a file. This can be useful in certain situations, but most applications read files sequentially, using the stream classes described in the next section. Here is a short example of using `RandomAccessFile`:

```

// Open a file for read/write ("rw") access
File datafile = new File(configdir, "datafile");
RandomAccessFile f = new RandomAccessFile(datafile, "rw");
f.seek(100); // Move to byte 100 of the file
byte[] data = new byte[100]; // Create a buffer to hold data
f.read(data); // Read 100 bytes from the file
int i = f.readInt(); // Read a 4-byte integer from the file
f.seek(100); // Move back to byte 100
f.writeInt(i); // Write the integer first
f.write(data); // Then write the 100 bytes
f.close(); // Close file when done with it

```

## Input/Output with java.io

The `java.io` package defines a large number of classes for reading and writing streaming, or sequential, data. The `InputStream` and `OutputStream` classes are for reading and writing streams of bytes while the `Reader` and `Writer` classes are for reading and writing streams of characters. Streams can be nested, meaning you might read characters from a `FilterReader` object that reads and processes characters from an underlying `Reader` stream. This underlying `Reader` stream might read bytes from an `InputStream` and convert them to characters.

### Reading Console Input

You can perform a number of common operations with streams. One is to read lines of input the user types at the console:

```

import java.io.*;

BufferedReader console = new BufferedReader(new InputStreamReader(System.in));
System.out.print("What is your name: ");
String name = null;
try {
    name = console.readLine();
}

```

```

}
catch (IOException e) { name = "<" + e + ">"; } // This should never happen
System.out.println("Hello " + name);

```

## Reading Lines from a Text File

Reading lines of text from a file is a similar operation. The following code reads an entire text file and quits when it reaches the end:

```

String filename = System.getProperty("user.home") + File.separator + ".cshrc";
try {
    BufferedReader in = new BufferedReader(new FileReader(filename));
    String line;
    while((line = in.readLine()) != null) { // Read line, check for end-of-file
        System.out.println(line);        // Print the line
    }
    in.close(); // Always close a stream when you are done with it
}
catch (IOException e) {
    // Handle FileNotFoundException, etc. here
}

```

## Writing Text to a File

Throughout this book, you've seen the use of the `System.out.println()` method to display text on the console. `System.out` simply refers to an output stream. You can print text to any output stream using similar techniques. The following code shows how to output text to a file:

```

try {
    File f = new File(homedir, ".config");
    PrintWriter out = new PrintWriter(new FileWriter(f));
    out.println("## Automatically generated config file. DO NOT EDIT!");
    out.close(); // We're done writing
}
catch (IOException e) { /* Handle exceptions */ }

```

## Reading a Binary File

Not all files contain text, however. The following lines of code treat a file as a stream of bytes and read the bytes into a large array:

```

try {
    File f; // File to read; initialized elsewhere
    int filesize = (int) f.length(); // Figure out the file size
    byte[] data = new byte[filesize]; // Create an array that is big enough
    // Create a stream to read the file
    DataInputStream in = new DataInputStream(new FileInputStream(f));
    in.readFully(data); // Read file contents into array
    in.close();
}
catch (IOException e) { /* Handle exceptions */ }

```

## Compressing Data

Various other packages of the Java platform define specialized stream classes that operate on streaming data in some useful way. The following code shows how to use stream classes from `java.util.zip` to compute a checksum of data and then compress the data while writing it to a file:

```
import java.io.*;
import java.util.zip.*;

try {
    File f;           // File to write to; initialized elsewhere
    byte[] data;     // Data to write; initialized elsewhere
    Checksum check = new Adler32(); // An object to compute a simple checksum

    // Create a stream that writes bytes to the file f
    FileOutputStream fos = new FileOutputStream(f);
    // Create a stream that compresses bytes and writes them to fos
    GZIPOutputStream gzos = new GZIPOutputStream(fos);
    // Create a stream that computes a checksum on the bytes it writes to gzos
    CheckedOutputStream cos = new CheckedOutputStream(gzos, check);

    cos.write(data); // Now write the data to the nested streams
    cos.close();     // Close down the nested chain of streams
    long sum = check.getValue(); // Obtain the computed checksum
}
catch (IOException e) { /* Handle exceptions */ }
```

## Reading ZIP Files

The `java.util.zip` package also contains a `ZipFile` class that gives you random access to the entries of a ZIP archive and allows you to read those entries through a stream:

```
import java.io.*;
import java.util.zip.*;

String filename; // File to read; initialized elsewhere
String entryname; // Entry to read from the ZIP file; initialized elsewhere
ZipFile zipfile = new ZipFile(filename); // Open the ZIP file
ZipEntry entry = zipfile.getEntry(entryname); // Get one entry
InputStream in = zipfile.getInputStream(entry); // A stream to read the entry
BufferedInputStream bis = new BufferedInputStream(in); // Improves efficiency
// Now read bytes from bis...
// Print out contents of the ZIP file
for(java.util.Enumeration e = zipfile.entries(); e.hasMoreElements();) {
    ZipEntry zipentry = (ZipEntry) e.nextElement();
    System.out.println(zipentry.getName());
}
```

## Computing Message Digests

If you need to compute a cryptographic-strength checksum (also known as a message digest), use one of the stream classes of the `java.security` package. For example:

```
import java.io.*;
import java.security.*;
import java.util.*;

File f;           // File to read and compute digest on; initialized elsewhere
List text = new ArrayList(); // We'll store the lines of text here

// Get an object that can compute an SHA message digest
MessageDigest digester = MessageDigest.getInstance("SHA");
// A stream to read bytes from the file f
FileInputStream fis = new FileInputStream(f);
// A stream that reads bytes from fis and computes an SHA message digest
DigestInputStream dis = new DigestInputStream(fis, digester);
// A stream that reads bytes from dis and converts them to characters
InputStreamReader isr = new InputStreamReader(dis);
// A stream that can read a line at a time
BufferedReader br = new BufferedReader(isr);
// Now read lines from the stream
for(String line; (line = br.readLine()) != null; text.add(line)) ;
// Close the streams
br.close();
// Get the message digest
byte[] digest = digester.digest();
```

## Streaming Data to and from Arrays

So far, we've used a variety of stream classes to manipulate streaming data, but the data itself ultimately comes from a file or is written to the console. The `java.io` package defines other stream classes that can read data from and write data to arrays of bytes or strings of text:

```
import java.io.*;

// Set up a stream that uses a byte array as its destination
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream out = new DataOutputStream(baos);
out.writeUTF("hello");           // Write some string data out as bytes
out.writeDouble(Math.PI);       // Write a floating-point value out as bytes
byte[] data = baos.toByteArray(); // Get the array of bytes we've written
out.close();                     // Close the streams

// Set up a stream to read characters from a string
Reader in = new StringReader("Now is the time!");
// Read characters from it until we reach the end
int c;
while((c = in.read()) != -1) System.out.print((char) c);
```

Other classes that operate this way include `ByteArrayInputStream`, `StringWriter`, `CharArrayReader`, and `CharArrayWriter`.

## Thread Communication with Pipes

`PipedInputStream` and `PipedOutputStream` and their character-based counterparts, `PipedReader` and `PipedWriter`, are another interesting set of streams defined by `java.io`. These streams are used in pairs by two threads that want to communicate. One thread writes bytes to a `PipedOutputStream` or characters to a `PipedWriter`, and another thread reads bytes or characters from the corresponding `PipedInputStream` or `PipedReader`:

```
// A pair of connected piped I/O streams forms a pipe. One thread writes
// bytes to the PipedOutputStream, and another thread reads them from the
// corresponding PipedInputStream. Or use PipedWriter/PipedReader for chars.
final PipedOutputStream writeEndOfPipe = new PipedOutputStream();
final PipedInputStream readEndOfPipe = new PipedInputStream(writeEndOfPipe);

// This thread reads bytes from the pipe and discards them
Thread devnull = new Thread(new Runnable() {
    public void run() {
        try { while(readEndOfPipe.read() != -1); }
        catch (IOException e) {} // ignore it
    }
});
devnull.start();
```