

statement matches the exception. Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause.

Here is an example program that shows three methods that exit in various ways, none without executing their **finally** clauses:

```
// Demonstrate finally.
class FinallyDemo {
    // Throw an exception out of the method.
    static void procA() {
        try {
            System.out.println("inside procA");
            throw new RuntimeException("demo");
        } finally {
            System.out.println("procA's finally");
        }
    }

    // Return from within a try block.
    static void procB() {
        try {
            System.out.println("inside procB");
            return;
        } finally {
            System.out.println("procB's finally");
        }
    }

    // Execute a try block normally.
    static void procC() {
        try {
            System.out.println("inside procC");
        } finally {
            System.out.println("procC's finally");
        }
    }

    public static void main(String[] args) {
        try {
            procA();
        } catch (Exception e) {
            System.out.println("Exception caught");
        }

        procB();
        procC();
    }
}
```

```
}
```

In this example, **procA()** prematurely breaks out of the **try** by throwing an exception. The **finally** clause is executed on the way out. **procB()**'s **try** statement is exited via a **return** statement. The **finally** clause is executed before **procB()** returns. In **procC()**, the **try** statement executes normally, without error. However, the **finally** block is still executed.

Remember If a **finally** block is associated with a **try**, the **finally** block will be executed upon conclusion of the **try**.

Here is the output generated by the preceding program:

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

Java's Built-in Exceptions

Inside the standard package **java.lang**, Java defines several exception classes. A few have been used by the preceding examples. The most general of these exceptions are subclasses of the standard type

RuntimeException. As previously explained, these exceptions need not be included in any method's **throws** list. In the language of Java, these are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions. The unchecked exceptions defined in **java.lang** are listed in [Table 10-1](#). [Table 10-2](#) lists those exceptions defined by **java.lang** that must be included in a method's **throws** list if that method can generate one of these exceptions and does not handle it itself. These are called *checked exceptions*. In addition to the exceptions in **java.lang**, Java defines several more that relate to its other standard packages.

Exception	Meaning
ArithmaticException	Arithmatic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalCallerException	A method cannot be legally executed by the calling code.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
LayerInstantiationException	A module layer cannot be created.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.
TypeNotFoundException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Table 10-1 Java's Unchecked **RuntimeException** Subclasses Defined in **java.lang**

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.
ReflectiveOperationException	Superclass of reflection-related exceptions.

Table 10-2 Java’s Checked Exceptions Defined in `java.lang`

Creating Your Own Exception Subclasses

Although Java’s built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications. This is quite easy to do: just define a subclass of **Exception** (which is, of course, a subclass of **Throwable**). Your subclasses don’t need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions.

The **Exception** class does not define any methods of its own. It does, of course, inherit those methods provided by **Throwable**. Thus, all exceptions, including those that you create, have the methods defined by **Throwable** available to them. They are shown in [Table 10-3](#). You may also wish to override one or more of these methods in exception classes that you create.

Method	Description
final void addSuppressed(Throwable <i>exc</i>)	Adds <i>exc</i> to the list of suppressed exceptions associated with the invoking exception. Primarily for use by the try-with-resources statement.
Throwable fillInStackTrace()	Returns a Throwable object that contains a completed stack trace. This object can be rethrown.
Throwable getCause()	Returns the exception that underlies the current exception. If there is no underlying exception, null is returned.
String getLocalizedMessage()	Returns a localized description of the exception.
String getMessage()	Returns a description of the exception.
StackTraceElement[] getStackTrace()	Returns an array that contains the stack trace, one element at a time, as an array of StackTraceElement . The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array. The StackTraceElement class gives your program access to information about each element in the trace, such as its method name.
final Throwable[] getSuppressed()	Obtains the suppressed exceptions associated with the invoking exception and returns an array that contains the result. Suppressed exceptions are primarily generated by the try-with-resources statement.
Throwable initCause(Throwable <i>causeExc</i>)	Associates <i>causeExc</i> with the invoking exception as a cause of the invoking exception. Returns a reference to the exception.
void printStackTrace()	Displays the stack trace.
void printStackTrace(PrintStream <i>stream</i>)	Sends the stack trace to the specified stream.
void printStackTrace(PrintWriter <i>stream</i>)	Sends the stack trace to the specified stream.
void setStackTrace(StackTraceElement[] <i>elements</i>)	Sets the stack trace to the elements passed in <i>elements</i> . This method is for specialized applications, not normal use.

String toString()	Returns a String object containing a description of the exception. This method is called by println() when outputting a Throwable object.
----------------------------------	---

Table 10-3 The Methods Defined by **Throwable**

Exception defines four public constructors. Two support chained exceptions, described in the next section. The other two are shown here:

Exception()
Exception(String msg)

The first form creates an exception that has no description. The second form lets you specify a description of the exception.

Although specifying a description when an exception is created is often useful, sometimes it is better to override **toString()**. Here's why: The version of **toString()** defined by **Throwable** (and inherited by **Exception**) first displays the name of the exception followed by a colon, which is then followed by your description. By overriding **toString()**, you can prevent the exception name and colon from being displayed. This makes for a cleaner output, which is desirable in some cases.

The following example declares a new subclass of **Exception** and then uses that subclass to signal an error condition in a method. It overrides the **toString()** method, allowing a carefully tailored description of the exception to be displayed.

```

// This program creates a custom exception type.
class MyException extends Exception {
    private int detail;

    MyException(int a) {
        detail = a;
    }

    public String toString() {
        return "MyException[" + detail + "]";
    }
}

class ExceptionDemo {
    static void compute(int a) throws MyException {
        System.out.println("Called compute(" + a + ")");
        if(a > 10)
            throw new MyException(a);
        System.out.println("Normal exit");
    }

    public static void main(String[] args) {
        try {
            compute(1);
            compute(20);
        } catch (MyException e) {
            System.out.println("Caught " + e);
        }
    }
}

```

This example defines a subclass of **Exception** called **MyException**. This subclass is quite simple: It has only a constructor plus an overridden **toString()** method that displays the value of the exception. The **ExceptionDemo** class defines a method named **compute()** that throws a **MyException** object. The exception is thrown when **compute()**'s integer parameter is greater than 10. The **main()** method sets up an exception handler for **MyException**, then calls **compute()** with a legal value (less than 10) and an illegal one to show both paths through the code. Here is the result:

```
Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]
```

Chained Exceptions

A number of years ago, a feature was incorporated into the exception subsystem: *chained exceptions*. The chained exception feature allows you to associate another exception with an exception. This second exception describes the cause of the first exception. For example, imagine a situation in which a method throws an **ArithmeticException** because of an attempt to divide by zero. However, the actual cause of the problem was that an I/O error occurred, which caused the divisor to be set improperly.

Although the method must certainly throw an **ArithmeticException**, since that is the error that occurred, you might also want to let the calling code know that the underlying cause was an I/O error. Chained exceptions let you handle this, and any other situation in which layers of exceptions exist.

To allow chained exceptions, two constructors and two methods were added to **Throwable**. The constructors are shown here:

```
Throwable(Throwable causeExc)
Throwable(String msg, Throwable causeExc)
```

In the first form, *causeExc* is the exception that causes the current exception. That is, *causeExc* is the underlying reason that an exception occurred. The second form allows you to specify a description at the same time that you specify a cause exception. These two constructors have also been added to the **Error**, **Exception**, and **RuntimeException** classes.

The chained exception methods supported by **Throwable** are **getCause()** and **initCause()**. These methods are shown in [Table 10-3](#) and are repeated here for the sake of discussion.

```
Throwable getCause()
Throwable initCause(Throwable causeExc)
```

The **getCause()** method returns the exception that underlies the current exception. If there is no underlying exception, **null** is returned. The **initCause()** method associates *causeExc* with the invoking exception and returns a reference to the exception. Thus, you can associate a cause with an exception after the exception has been created. However, the cause exception can be set only once. This means that you can call **initCause()** only once for each exception object. Furthermore, if the cause exception was set by a constructor, then you can't set it again using **initCause()**. In general, **initCause()** is used to set a cause for legacy exception classes that don't support the two additional constructors described earlier.

Here is an example that illustrates the mechanics of handling chained exceptions:

```
// Demonstrate exception chaining.
class ChainExcDemo {
    static void demoproc() {

        // create an exception
        NullPointerException e =
            new NullPointerException("top layer");

        // add a cause
        e.initCause(new ArithmeticException("cause"));

        throw e;
    }

    public static void main(String[] args) {
        try {
            demoproc();
        } catch(NullPointerException e) {
            // display top level exception
            System.out.println("Caught: " + e);

            // display cause exception
            System.out.println("Original cause: " +
                e.getCause());
        }
    }
}
```

The output from the program is shown here:

```
Caught: java.lang.NullPointerException: top layer
Original cause: java.lang.ArithmetricException: cause
```

In this example, the top-level exception is **NullPointerException**. To it is added a cause exception, **ArithmetricException**. When the exception is thrown out of **demoproc()**, it is caught by **main()**. There, the top-level exception is displayed, followed by the underlying exception, which is obtained by calling **getCause()**.

Chained exceptions can be carried on to whatever depth is necessary. Thus, the cause exception can, itself, have a cause. Be aware that overly long chains of exceptions may indicate poor design.

Chained exceptions are not something that every program will need. However, in cases in which knowledge of an underlying cause is useful, they offer an elegant solution.

Three Additional Exception Features

Beginning with JDK 7, three interesting and useful features have been part of the exception system. The first automates the process of releasing a resource, such as a file, when it is no longer needed. It is based on an expanded form of the **try** statement called **try-with-resources**, and is described in [Chapter 13](#) when files are introduced. The second feature is called **multi-catch**, and the third is sometimes referred to as *final rethrow* or *more precise rethrow*. These two features are described here.

The multi-catch feature allows two or more exceptions to be caught by the same **catch** clause. It is not uncommon for two or more exception handlers to use the same code sequence even though they respond to different exceptions. Instead of having to catch each exception type individually, you can use a single **catch** clause to handle all of the exceptions without code duplication.

To use a multi-catch, separate each exception type in the **catch** clause with the OR operator. Each multi-catch parameter is implicitly **final**. (You can explicitly specify **final**, if desired, but it is not necessary.) Because

each multi-catch parameter is implicitly **final**, it can't be assigned a new value.

Here is a **catch** statement that uses the multi-catch feature to catch both **ArithmeticeException** and **ArrayIndexOutOfBoundsException**:

```
catch(ArithmeticeException | ArrayIndexOutOfBoundsException e) {
```

The following program shows the multi-catch feature in action:

```
// Demonstrate the multi-catch feature.
class MultiCatch {
    public static void main(String[] args) {
        int a=10, b=0;
        int[] vals = { 1, 2, 3 };

        try {
            int result = a / b; // generate an ArithmeticeException

//          vals[10] = 19; // generate an ArrayIndexOutOfBoundsException

            // This catch clause catches both exceptions.
        } catch(ArithmeticeException | ArrayIndexOutOfBoundsException e) {
            System.out.println("Exception caught: " + e);
        }

        System.out.println("After multi-catch.");
    }
}
```

The program will generate an **ArithmeticeException** when the division by zero is attempted. If you comment out the division statement and remove the comment symbol from the next line, an

ArrayIndexOutOfBoundsException is generated. Both exceptions are caught by the single **catch** statement.

The more precise rethrow feature restricts the type of exceptions that can be rethrown to only those checked exceptions that the associated **try** block throws, that are not handled by a preceding **catch** clause, and that are a subtype or supertype of the parameter. Although this capability might not be needed often, it is now available for use. For the more precise

rethrow feature to be in force, the **catch** parameter must be either effectively **final**, which means that it must not be assigned a new value inside the **catch** block, or explicitly declared **final**.

Using Exceptions

Exception handling provides a powerful mechanism for controlling complex programs that have many dynamic run-time characteristics. It is important to think of **try**, **throw**, and **catch** as clean ways to handle errors and unusual boundary conditions in your program's logic. Instead of using error return codes to indicate failure, use Java's exception handling capabilities. Thus, when a method can fail, have it throw an exception. This is a cleaner way to handle failure modes.

One last point: Java's exception-handling statements should not be considered a general mechanism for nonlocal branching. If you do so, it will only confuse your code and make it hard to maintain.

CHAPTER 11

Multithreaded Programming

Java provides built-in support for *multithreaded programming*. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a *thread*, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

You are almost certainly acquainted with multitasking because it is supported by virtually all modern operating systems. However, there are two distinct types of multitasking: process-based and thread-based. It is important to understand the difference between the two. For many readers, process-based multitasking is the more familiar form. A *process* is, in essence, a program that is executing. Thus, *process-based* multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor or visiting a web site. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

In a *thread-based* multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads. Thus, process-based multitasking deals with the “big picture,” and thread-based multitasking handles the details.

Multitasking threads require less overhead than multitasking processes. Processes are heavyweight tasks that require their own separate address spaces. Interprocess communication is expensive and limited. Context

switching from one process to another is also costly. Threads, on the other hand, are lighter weight. They share the same address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is lower in cost. While Java programs make use of process-based multitasking environments, process-based multitasking is not under Java's direct control. However, multithreaded multitasking is.

Multithreading enables you to write efficient programs that make maximum use of the processing power available in the system. One important way multithreading achieves this is by keeping idle time to a minimum. This is especially important for the interactive, networked environment in which Java operates because idle time is common. For example, the transmission rate of data over a network is much slower than the rate at which the computer can process it. Even local file system resources are read and written at a much slower pace than they can be processed by the CPU. And, of course, user input is much slower than the computer. In a single-threaded environment, your program has to wait for each of these tasks to finish before it can proceed to the next one—even though most of the time the program is idle, waiting for input. Multithreading helps you reduce this idle time because another thread can run when one is waiting.

If you have programmed for operating systems such as Windows, then you are already familiar with multithreaded programming. However, the fact that Java manages threads makes multithreading especially convenient because many of the details are handled for you.

The Java Thread Model

The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind. In fact, Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles.

The value of a multithreaded environment is best understood in contrast to its counterpart. Single-threaded systems use an approach called an *event loop* with *polling*. In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next. Once

this polling mechanism returns with, say, a signal that a network file is ready to be read, then the event loop dispatches control to the appropriate event handler. Until this event handler returns, nothing else can happen in the program. This wastes CPU time. It can also result in one part of a program dominating the system and preventing any other events from being processed. In general, in a single-threaded environment, when a thread *blocks* (that is, suspends execution) because it is waiting for some resource, the entire program stops running.

The benefit of Java's multithreading is that the main loop/polling mechanism is eliminated. One thread can pause without stopping other parts of your program. For example, the idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere. Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause. When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run.

As most readers know, over the past few years, multicore systems have become commonplace. Of course, single-core systems are still in widespread use. It is important to understand that Java's multithreading features work in both types of systems. In a single-core system, concurrently executing threads share the CPU, with each thread receiving a slice of CPU time. Therefore, in a single-core system, two or more threads do not actually run at the same time, but idle CPU time is utilized. However, in multicore systems, it is possible for two or more threads to actually execute simultaneously. In many cases, this can further improve program efficiency and increase the speed of certain operations.

Note In addition to the multithreading features described in this chapter, you will also want to explore the Fork/Join Framework. It provides a powerful means of creating multithreaded applications that automatically scale to make best use of multicore environments. The Fork/Join Framework is part of Java's support for *parallel programming*, which is the name commonly given to the techniques that optimize some types of algorithms for parallel execution in systems that have more than one CPU. For a discussion of the Fork/Join Framework and other concurrency utilities, see [Chapter 29](#). Java's traditional multithreading capabilities are described here.

Threads exist in several states. Here is a general description. A thread can be *running*. It can be *ready to run* as soon as it gets CPU time. A running thread can be *suspended*, which temporarily halts its activity. A suspended thread can then be *resumed*, allowing it to pick up where it left off. A thread can be *blocked* when waiting for a resource. At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed.

Thread Priorities

Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. Thread priorities are integers that specify the relative priority of one thread to another. As an absolute value, a priority is meaningless; a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running. Instead, a thread's priority is used to decide when to switch from one running thread to the next. This is called a *context switch*. The rules that determine when a *context switch* takes place are simple:

- *A thread can voluntarily relinquish control.* This occurs when explicitly yielding, sleeping, or when blocked. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.
- *A thread can be preempted by a higher-priority thread.* In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing—by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called *preemptive multitasking*.

In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated. For some operating systems, threads of equal priority are time-sliced automatically in round-robin fashion. For other types of operating systems, threads of equal priority must voluntarily yield control to their peers. If they don't, the other threads will not run.

Caution Portability problems can arise from the differences in the way that operating systems context-switch threads of equal priority.

Synchronization

Because multithreading introduces an asynchronous behavior to your programs, there must be a way for you to enforce synchronicity when you need it. For example, if you want two threads to communicate and share a complicated data structure, such as a linked list, you need some way to ensure that they don't conflict with each other. That is, you must prevent one thread from writing data while another thread is in the middle of reading it. For this purpose, Java implements an elegant twist on an age-old model of interprocess synchronization: the *monitor*. The monitor is a control mechanism first defined by C.A.R. Hoare. You can think of a monitor as a very small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor. In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.

In Java, there is no class "Monitor"; instead, each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object. This enables you to write very clear and concise multithreaded code, because synchronization support is built into the language.

Messaging

After you divide your program into separate threads, you need to define how they will communicate with each other. When programming with some other languages, you must depend on the operating system to establish communication between threads. This, of course, adds overhead. By contrast, Java provides a clean, low-cost way for two or more threads to talk to each other, via calls to predefined methods that all objects have. Java's messaging system allows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out.

The Thread Class and the Runnable Interface

Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**. **Thread** encapsulates a thread of execution. Since you can't directly refer to the ethereal state of a running thread, you will deal with it through its proxy, the **Thread** instance that spawned it. To create a new thread, your program will either extend **Thread** or implement the **Runnable** interface.

The **Thread** class defines several methods that help manage threads. Several of those used in this chapter are shown here:

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

Thus far, all the examples in this book have used a single thread of execution. The remainder of this chapter explains how to use **Thread** and **Runnable** to create and manage threads, beginning with the one thread that all Java programs have: the main thread.

The Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons:

- It is the thread from which other “child” threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions.

Although the main thread is created automatically when your program is started, it can be controlled through a **Thread** object. To do so, you must obtain a reference to it by calling the method **currentThread()**, which is a **public static** member of **Thread**. Its general form is shown here:

```
static Thread currentThread()
```

This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread.

Let's begin by reviewing the following example:

```
// Controlling the main Thread.  
class CurrentThreadDemo {  
    public static void main(String[] args) {  
        Thread t = Thread.currentThread();  
  
        System.out.println("Current thread: " + t);  
  
        // change the name of the thread  
        t.setName("My Thread");  
        System.out.println("After name change: " + t);  
  
        try {  
            for(int n = 5; n > 0; n--) {  
                System.out.println(n);  
                Thread.sleep(1000);  
            }  
        } catch (InterruptedException e) {  
            System.out.println("Main thread interrupted");  
        }  
    }  
}
```

In this program, a reference to the current thread (the main thread, in this case) is obtained by calling **currentThread()**, and this reference is stored in the local variable **t**. Next, the program displays information about the thread. The program then calls **setName()** to change the internal name of the thread. Information about the thread is then redisplayed. Next,

a loop counts down from five, pausing one second between each line. The pause is accomplished by the `sleep()` method. The argument to `sleep()` specifies the delay period in milliseconds. Notice the `try/catch` block around this loop. The `sleep()` method in `Thread` might throw an **InterruptedException**. This would happen if some other thread wanted to interrupt this sleeping one. This example just prints a message if it gets interrupted. In a real program, you would need to handle this differently. Here is the output generated by this program:

```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

Notice the output produced when `t` is used as an argument to `println()`. This displays, in order: the name of the thread, its priority, and the name of its group. By default, the name of the main thread is **main**. Its priority is 5, which is the default value, and **main** is also the name of the group of threads to which this thread belongs. A *thread group* is a data structure that controls the state of a collection of threads as a whole. After the name of the thread is changed, `t` is again output. This time, the new name of the thread is displayed.

Let's look more closely at the methods defined by `Thread` that are used in the program. The `sleep()` method causes the thread from which it is called to suspend execution for the specified period of milliseconds. Its general form is shown here:

```
static void sleep(long milliseconds) throws InterruptedException
```

The number of milliseconds to suspend is specified in *milliseconds*. This method may throw an **InterruptedException**.

The `sleep()` method has a second form, shown next, which allows you to specify the period in terms of milliseconds and nanoseconds:

```
static void sleep(long milliseconds, int nanoseconds) throws
InterruptedException
```

This second form is useful only in environments that allow timing periods as short as nanoseconds.

As the preceding program shows, you can set the name of a thread by using `setName()`. You can obtain the name of a thread by calling `getName()` (but note that this is not shown in the program). These methods are members of the **Thread** class and are declared like this:

```
final void setName(String threadName)
final String getName()
```

Here, *threadName* specifies the name of the thread.

Creating a Thread

In the most general sense, you create a thread by instantiating an object of type **Thread**. Java defines two ways in which this can be accomplished:

- You can implement the **Runnable** interface.
- You can extend the **Thread** class, itself.

The following two sections look at each method, in turn.

Implementing Runnable

The easiest way to create a thread is to create a class that implements the **Runnable** interface. **Runnable** abstracts a unit of executable code. You can construct a thread on any object that implements **Runnable**. To implement **Runnable**, a class need only implement a single method called `run()`, which is declared like this:

```
public void run()
```

Inside `run()`, you will define the code that constitutes the new thread. It is important to understand that `run()` can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that `run()` establishes the entry point for another, concurrent

thread of execution within your program. This thread will end when **run()** returns.

After you create a class that implements **Runnable**, you will instantiate an object of type **Thread** from within that class. **Thread** defines several constructors. The one that we will use is shown here:

```
Thread(Runnable threadOb, String threadName)
```

In this constructor, *threadOb* is an instance of a class that implements the **Runnable** interface. This defines where execution of the thread will begin. The name of the new thread is specified by *threadName*.

After the new thread is created, it will not start running until you call its **start()** method, which is declared within **Thread**. In essence, **start()** initiates a call to **run()**. The **start()** method is shown here:

```
void start()
```

Here is an example that creates a new thread and starts it running:

```
// Create a second thread.
class NewThread implements Runnable {
    Thread t;

    NewThread() {
        // Create a new, second thread
        t = new Thread(this, "Demo Thread");
        System.out.println("Child thread: " + t);
    }

    // This is the entry point for the second thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}

class ThreadDemo {
    public static void main(String[] args) {
        NewThread nt = new NewThread(); // create a new thread

        nt.t.start(); // Start the thread

        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}
```

Inside **NewThread**'s constructor, a new **Thread** object is created by the following statement:

```
t = new Thread(this, "Demo Thread");
```

Passing **this** as the first argument indicates that you want the new thread to call the **run()** method on **this** object. Inside **main()**, **start()** is called, which starts the thread of execution beginning at the **run()** method. This causes the child thread's **for** loop to begin. Next the main thread enters its **for** loop. Both threads continue running, sharing the CPU in single-core systems, until their loops finish. The output produced by this program is as follows.(Your output may vary based upon the specific execution environment.)

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

As mentioned earlier, in a multithreaded program, it is often useful for the main thread to be the last thread to finish running. The preceding program ensures that the main thread finishes last, because the main thread sleeps for 1,000 milliseconds between iterations, but the child thread sleeps for only 500 milliseconds. This causes the child thread to terminate earlier than the main thread. Shortly, you will see a better way to wait for a thread to finish.

Extending Thread

The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class. The extending class must override the **run()** method, which is the entry point for the new thread. As before, a call to **start()** begins execution of the new thread. Here is the preceding program rewritten to extend **Thread**:

```
// Create a second thread by extending Thread
class NewThread extends Thread {

    NewThread() {
        // Create a new, second thread
        super("Demo Thread");
        System.out.println("Child thread: " + this);
    }

    // This is the entry point for the second thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}

class ExtendThread {
    public static void main(String[] args) {
        NewThread nt = new NewThread(); // create a new thread

        nt.start(); // start the thread

        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}
```

This program generates the same output as the preceding version. As you can see, the child thread is created by instantiating an object of **NewThread**, which is derived from **Thread**.

Notice the call to `super()` inside **NewThread**. This invokes the following form of the **Thread** constructor:

```
public Thread(String threadName)
```

Here, *threadName* specifies the name of the thread.

Choosing an Approach

At this point, you might be wondering why Java has two ways to create child threads, and which approach is better. The answers to these questions turn on the same point. The **Thread** class defines several methods that can be overridden by a derived class. Of these methods, the only one that *must* be overridden is `run()`. This is, of course, the same method required when you implement **Runnable**. Many Java programmers feel that classes should be extended only when they are being enhanced or adapted in some way. So, if you will not be overriding any of **Thread**'s other methods, it is probably best simply to implement **Runnable**. Also, by implementing **Runnable**, your thread class does not need to inherit **Thread**, making it free to inherit a different class. Ultimately, which approach to use is up to you. However, throughout the rest of this chapter, we will create threads by using classes that implement **Runnable**.

Creating Multiple Threads

So far, you have been using only two threads: the main thread and one child thread. However, your program can spawn as many threads as it needs. For example, the following program creates three child threads:

```
// Create multiple threads.
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
    }

    // This is the entry point for thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println(name + " Interrupted");
        }
        System.out.println(name + " exiting.");
    }
}

class MultiThreadDemo {
    public static void main(String[] args) {
        NewThread nt1 = new NewThread("One");
        NewThread nt2 = new NewThread("Two");
        NewThread nt3 = new NewThread("Three");

        // Start the threads.
        nt1.t.start();
        nt2.t.start();
        nt3.t.start();

        try {
            // wait for other threads to end
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }
    }
}
```

```
        System.out.println("Main thread exiting.");
    }
}
```

Sample output from this program is shown here. (Your output may vary based upon the specific execution environment.)

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

As you can see, once started, all three child threads share the CPU. Notice the call to **sleep(10000)** in **main()**. This causes the main thread to sleep for ten seconds and ensures that it will finish last.

Using **isAlive()** and **join()**

As mentioned, often you will want the main thread to finish last. In the preceding examples, this is accomplished by calling **sleep()** within **main()**

), with a long enough delay to ensure that all child threads terminate prior to the main thread. However, this is hardly a satisfactory solution, and it also raises a larger question: How can one thread know when another thread has ended? Fortunately, **Thread** provides a means by which you can answer this question.

Two ways exist to determine whether a thread has finished. First, you can call **isAlive()** on the thread. This method is defined by **Thread**, and its general form is shown here:

```
final boolean isAlive()
```

The **isAlive()** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise.

While **isAlive()** is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called **join()**, shown here:

```
final void join() throws InterruptedException
```

This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread *joins* it. Additional forms of **join()** allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

Here is an improved version of the preceding example that uses **join()** to ensure that the main thread is the last to stop. It also demonstrates the **isAlive()** method.

```
// Using join() to wait for threads to finish.
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
    }

    // This is the entry point for thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println(name + " interrupted.");
        }
        System.out.println(name + " exiting.");
    }
}
```

```
class DemoJoin {
    public static void main(String[] args) {
        NewThread nt1 = new NewThread("One");
        NewThread nt2 = new NewThread("Two");
        NewThread nt3 = new NewThread("Three");

        // Start the threads.
        nt1.t.start();
        nt2.t.start();
        nt3.t.start();

        System.out.println("Thread One is alive: "
                           + nt1.t.isAlive());
        System.out.println("Thread Two is alive: "
                           + nt2.t.isAlive());
        System.out.println("Thread Three is alive: "
                           + nt3.t.isAlive());
        // wait for threads to finish
        try {
            System.out.println("Waiting for threads to finish.");
            nt1.t.join();
            nt2.t.join();
            nt3.t.join();
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }

        System.out.println("Thread One is alive: "
                           + nt1.t.isAlive());
        System.out.println("Thread Two is alive: "
                           + nt2.t.isAlive());
        System.out.println("Thread Three is alive: "
                           + nt3.t.isAlive());

        System.out.println("Main thread exiting.");
    }
}
```

Sample output from this program is shown here. (Your output may vary based upon the specific execution environment.)

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.

One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
Two exiting.
Three exiting.
One exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.
```

As you can see, after the calls to **join()** return, the threads have stopped executing.

Thread Priorities

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, over a given period of time, higher-priority threads get more CPU time than lower-priority threads. In practice, the amount of CPU time that a thread gets often depends on

several factors besides its priority. (For example, how an operating system implements multitasking can affect the relative availability of CPU time.) A higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower-priority thread.

In theory, threads of equal priority should get equal access to the CPU. But you need to be careful. Remember, Java is designed to work in a wide range of environments. Some of those environments implement multitasking fundamentally differently than others. For safety, threads that share the same priority should yield control once in a while. This ensures that all threads have a chance to run under a nonpreemptive operating system. In practice, even in nonpreemptive environments, most threads still get a chance to run, because most threads inevitably encounter some blocking situation, such as waiting for I/O. When this happens, the blocked thread is suspended and other threads can run. But, if you want smooth multithreaded execution, you are better off not relying on this. Also, some types of tasks are CPU-intensive. Such threads dominate the CPU. For these types of threads, you want to yield control occasionally so that other threads can run.

To set a thread's priority, use the **setPriority()** method, which is a member of **Thread**. This is its general form:

```
final void setPriority(int level)
```

Here, *level* specifies the new priority setting for the calling thread. The value of *level* must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify **NORM_PRIORITY**, which is currently 5. These priorities are defined as **static final** variables within **Thread**.

You can obtain the current priority setting by calling the **getPriority()** method of **Thread**, shown here:

```
final int getPriority()
```

Implementations of Java may have radically different behavior when it comes to scheduling. Most of the inconsistencies arise when you have

threads that are relying on preemptive behavior, instead of cooperatively giving up CPU time. The safest way to obtain predictable, cross-platform behavior with Java is to use threads that voluntarily give up control of the CPU.

Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization*. As you will see, Java provides unique, language-level support for it.

Key to synchronization is the concept of the monitor. A *monitor* is an object that is used as a mutually exclusive lock. Only one thread can *own* a monitor at a given time. When a thread acquires a lock, it is said to have *entered* the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread *exits* the monitor. These other threads are said to be *waiting* for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires.

You can synchronize your code in either of two ways. Both involve the use of the **synchronized** keyword, and both are examined here.

Using Synchronized Methods

Synchronization is easy in Java, because all objects have their own implicit monitor associated with them. To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword. While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait. To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

To understand the need for synchronization, let's begin with a simple example that does not use it—but should. The following program has three simple classes. The first one, **Callme**, has a single method named **call()**. The **call()** method takes a **String** parameter called **msg**. This method tries to print the **msg** string inside of square brackets. The interesting thing to

notice is that after **call()** prints the opening bracket and the **msg** string, it calls **Thread.sleep(1000)**, which pauses the current thread for one second.

The constructor of the next class, **Caller**, takes a reference to an instance of the **Callme** class and a **String**, which are stored in **target** and **msg**, respectively. The constructor also creates a new thread that will call this object's **run()** method. The **run()** method of **Caller** calls the **call()** method on the **target** instance of **Callme**, passing in the **msg** string. Finally, the **Synch** class starts by creating a single instance of **Callme**, and three instances of **Caller**, each with a unique message string. The same instance of **Callme** is passed to each **Caller**.

```
// This program is not synchronized.

class Callme {
    void call(String msg) {
        System.out.print("[ " + msg);
        try {
            Thread.sleep(1000);
        } catch(InterruptedException e) {
            System.out.println("Interrupted");
        }
        System.out.println("] ");
    }
}

class Caller implements Runnable {
    String msg;
    Callme target;
    Thread t;

    public Caller(Callme targ, String s) {
        target = targ;
        msg = s;
        t = new Thread(this);
    }

    public void run() {
        target.call(msg);
    }
}

class Synch {
    public static void main(String[] args) {
        Callme target = new Callme();
        Caller ob1 = new Caller(target, "Hello");
        Caller ob2 = new Caller(target, "Synchronized");
        Caller ob3 = new Caller(target, "World");

        // Start the threads.
        ob1.t.start();
        ob2.t.start();
        ob3.t.start();

        // wait for threads to end
    }
}
```

```
// ...  
try {  
    ob1.t.join();  
    ob2.t.join();  
    ob3.t.join();  
} catch(InterruptedException e) {  
    System.out.println("Interrupted");  
}  
}  
}
```

Here is the output produced by this program:

```
[Hello [Synchronized [World]]  
]  
]  
]
```

As you can see, by calling `sleep()`, the `call()` method allows execution to switch to another thread. This results in the mixed-up output of the three message strings. In this program, nothing exists to stop all three threads from calling the same method, on the same object, at the same time. This is known as a *race condition*, because the three threads are racing each other to complete the method. This example used `sleep()` to make the effects repeatable and obvious. In most situations, a race condition is more subtle and less predictable, because you can't be sure when the context switch will occur. This can cause a program to run right one time and wrong the next.

To fix the preceding program, you must *serialize* access to `call()`. That is, you must restrict its access to only one thread at a time. To do this, you simply need to precede `call()`'s definition with the keyword **synchronized**, as shown here:

```
class Callme {  
    synchronized void call(String msg) {  
        System.out.println(msg);  
    }  
}
```

This prevents other threads from entering `call()` while another thread is using it. After `synchronized` has been added to `call()`, the output of the program is as follows:

```
[Hello]  
[Synchronized]  
[World]
```

Any time that you have a method, or group of methods, that manipulates the internal state of an object in a multithreaded situation, you should use the **synchronized** keyword to guard the state from race conditions. Remember, once a thread enters any synchronized method on an instance, no other thread can enter any other synchronized method on the same instance. However, nonsynchronized methods on that instance will continue to be callable.

The synchronized Statement

While creating **synchronized** methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases. To understand why, consider the following. Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use **synchronized** methods. Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add **synchronized** to the appropriate methods within the class. How can access to an object of this class be synchronized? Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a **synchronized** block.

This is the general form of the **synchronized** statement:

```
synchronized(objRef) {  
    // statements to be synchronized  
}
```

Here, *objRef* is a reference to the object being synchronized. A synchronized block ensures that a call to a synchronized method that is a member of *objRef's class* occurs only after the current thread has successfully entered *objRef's monitor*.

Here is an alternative version of the preceding example, using a synchronized block within the **run()** method:

```
// This program uses a synchronized block.
class Callme {
    void call(String msg) {
        System.out.print("[ " + msg);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
        System.out.println("] ");
    }
}

class Caller implements Runnable {
    String msg;
    Callme target;
    Thread t;

    public Caller(Callme targ, String s) {
        target = targ;
        msg = s;
        t = new Thread(this);
    }

    // synchronize calls to call()
    public void run() {
        synchronized(target) { // synchronized block
            target.call(msg);
        }
    }
}

class Synch1 {
    public static void main(String[] args) {
        Callme target = new Callme();
        Caller ob1 = new Caller(target, "Hello");
        Caller ob2 = new Caller(target, "Synchronized");
        Caller ob3 = new Caller(target, "World");

        // Start the threads.
        ob1.t.start();
        ob2.t.start();
    }
}
```

```
    ob3.t.start();

    // wait for threads to end
    try {
        ob1.t.join();
        ob2.t.join();
        ob3.t.join();
    } catch(InterruptedException e) {
        System.out.println("Interrupted");
    }
}
```

Here, the `call()` method is not modified by `synchronized`. Instead, the `synchronized` statement is used inside `Caller's run()` method. This causes the same correct output as the preceding example, because each thread waits for the prior one to finish before proceeding.

Interthread Communication

The preceding examples unconditionally blocked other threads from asynchronous access to certain methods. This use of the implicit monitors in Java objects is powerful, but you can achieve a more subtle level of control through interprocess communication. As you will see, this is especially easy in Java.

As discussed earlier, multithreading replaces event loop programming by dividing your tasks into discrete, logical units. Threads also provide a secondary benefit: they do away with polling. Polling is usually implemented by a loop that is used to check some condition repeatedly. Once the condition is true, appropriate action is taken. This wastes CPU time. For example, consider the classic queuing problem, where one thread is producing some data and another is consuming it. To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data. In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce. Once the producer was finished, it would start polling, wasting

more CPU cycles waiting for the consumer to finish, and so on. Clearly, this situation is undesirable.

To avoid polling, Java includes an elegant interprocess communication mechanism via the **wait()**, **notify()**, and **notifyAll()** methods. These methods are implemented as **final** methods in **Object**, so all classes have them. All three methods can be called only from within a **synchronized** context. Although conceptually advanced from a computer science perspective, the rules for using these methods are actually quite simple:

- **wait()** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify() or notifyAll()**.
- **notify()** wakes up a thread that called **wait()** on the same object.
- **notifyAll()** wakes up all the threads that called **wait()** on the same object. One of the threads will be granted access.

These methods are declared within **Object**, as shown here:

```
final void wait() throws InterruptedException  
final void notify()  
final void notifyAll()
```

Additional forms of **wait()** exist that allow you to specify a period of time to wait.

Before working through an example that illustrates interthread communication, an important point needs to be made. Although **wait()** normally waits until **notify()** or **notifyAll()** is called, there is a possibility that in very rare cases the waiting thread could be awakened due to a *spurious wakeup*. In this case, a waiting thread resumes without **notify()** or **notifyAll()** having been called. (In essence, the thread resumes for no apparent reason.) Because of this remote possibility, the Java API documentation recommends that calls to **wait()** should take place within a loop that checks the condition on which the thread is waiting. The following example shows this technique.

Let's now work through an example that uses **wait()** and **notify()**. To begin, consider the following sample program that incorrectly implements a simple form of the producer/consumer problem. It consists of four

classes: **Q**, the queue that you're trying to synchronize; **Producer**, the threaded object that is producing queue entries; **Consumer**, the threaded object that is consuming queue entries; and **PC**, the tiny class that creates the single **Q**, **Producer**, and **Consumer**.

```
// An incorrect implementation of a producer and consumer.
class Q {
    int n;

    synchronized int get() {
        System.out.println("Got: " + n);
        return n;
    }

    synchronized void put(int n) {
        this.n = n;
        System.out.println("Put: " + n);
    }
}

class Producer implements Runnable {
    Q q;
    Thread t;

    Producer(Q q) {
        this.q = q;
        t = new Thread(this, "Producer");
    }

    public void run() {
        int i = 0;

        while(true) {
            q.put(i++);
        }
    }
}
```

```

        }
    }
}

class Consumer implements Runnable {
    Q q;
    Thread t;

    Consumer(Q q) {
        this.q = q;
        t = new Thread(this, "Consumer");
    }

    public void run() {
        while(true) {
            q.get();
        }
    }
}

class PC {
    public static void main(String[] args) {
        Q q = new Q();
        Producer p = new Producer(q);
        Consumer c = new Consumer(q);

        // Start the threads.
        p.t.start();
        c.t.start();

        System.out.println("Press Control-C to stop.");
    }
}

```

Although the **put()** and **get()** methods on **Q** are synchronized, nothing stops the producer from overrunning the consumer, nor will anything stop the consumer from consuming the same queue value twice. Thus, you get the erroneous output shown here (the exact output will vary with processor speed and task load):

```
Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Got: 1
Put: 2
Put: 3
Put: 4
Put: 5
Put: 6
Put: 7
Got: 7
```

As you can see, after the producer put 1, the consumer started and got the same 1 five times in a row. Then, the producer resumed and produced 2 through 7 without letting the consumer have a chance to consume them.

The proper way to write this program in Java is to use **wait()** and **notify()** to signal in both directions, as shown here:

```
// A correct implementation of a producer and consumer.
class Q {
    int n;
    boolean valueSet = false;

    synchronized int get() {
        while(!valueSet)
            try {
                wait();
            } catch(InterruptedException e) {
                System.out.println("InterruptedException caught");
            }

        System.out.println("Got: " + n);
        valueSet = false;
        notify();
        return n;
    }

    synchronized void put(int n) {
        while(valueSet)
            try {
                wait();
            } catch(InterruptedException e) {
                System.out.println("InterruptedException caught");
            }

        this.n = n;
        valueSet = true;
        System.out.println("Put: " + n);
        notify();
    }
}

class Producer implements Runnable {
    Q q;
    Thread t;

    Producer(Q q) {
        this.q = q;
        t = new Thread(this, "Producer");
    }
}
```

```
    } // now we can add,  execute ...
```

```
}
```

```
public void run() {
    int i = 0;
```

```

    while(true) {
        q.put(i++);
    }
}
```

```
}
```

```
class Consumer implements Runnable {
    Q q;
    Thread t;
```

```

    Consumer(Q q) {
        this.q = q;
        t = new Thread(this, "Consumer");
    }
}
```

```

    public void run() {
        while(true) {
            q.get();
        }
    }
}
```

```
}
```

```
class PCFixed {
    public static void main(String[] args) {
        Q q = new Q();
        Producer p = new Producer(q);
        Consumer c = new Consumer(q);

        // Start the threads.
        p.t.start();
        c.t.start();

        System.out.println("Press Control-C to stop.");
    }
}
```

Inside `get()`, `wait()` is called. This causes its execution to suspend until **Producer** notifies you that some data is ready. When this happens, execution inside `get()` resumes. After the data has been obtained, `get()` calls `notify()`. This tells **Producer** that it is okay to put more data in the queue. Inside `put()`, `wait()` suspends execution until **Consumer** has removed the item from the queue. When execution resumes, the next item of data is put in the queue, and `notify()` is called. This tells **Consumer** that it should now remove it.

Here is some output from this program, which shows the clean synchronous behavior:

```
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
```

Deadlock

A special type of error that you need to avoid that relates specifically to multitasking is *deadlock*, which occurs when two threads have a circular dependency on a pair of synchronized objects. For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete. Deadlock is a difficult error to debug for two reasons:

- In general, it occurs only rarely, when the two threads time-slice in just the right way.

- It may involve more than two threads and two synchronized objects.
(That is, deadlock can occur through a more convoluted sequence of events than just described.)

To understand deadlock fully, it is useful to see it in action. The next example creates two classes, **A** and **B**, with methods **foo()** and **bar()**, respectively, which pause briefly before trying to call a method in the other class. The main class, named **Deadlock**, creates an **A** and a **B** instance, and then calls **deadlockStart()** to start a second thread that sets up the deadlock condition. The **foo()** and **bar()** methods use **sleep()** as a way to force the deadlock condition to occur.

```
// An example of deadlock.
class A {
    synchronized void foo(B b) {
        String name = Thread.currentThread().getName();

        System.out.println(name + " entered A.foo");

        try {
            Thread.sleep(1000);
        } catch(Exception e) {
            System.out.println("A Interrupted");
        }

        System.out.println(name + " trying to call B.last()");
        b.last();
    }

    synchronized void last() {
        System.out.println("Inside A.last");
    }
}

class B {
    synchronized void bar(A a) {
        String name = Thread.currentThread().getName();
        System.out.println(name + " entered B.bar");

        try {
            Thread.sleep(1000);
        } catch(Exception e) {
```

```
        System.out.println("B Interrupted");
    }

    System.out.println(name + " trying to call A.last()");
    a.last();
}

synchronized void last() {
    System.out.println("Inside B.last");
}
}

class Deadlock implements Runnable {
    A a = new A();
    B b = new B();
    Thread t;

    Deadlock() {
        Thread.currentThread().setName("MainThread");
        t = new Thread(this, "RacingThread");
    }

    void deadlockStart() {
        t.start();
        a.foo(b); // get lock on a in this thread.
        System.out.println("Back in main thread");
    }

    public void run() {
        b.bar(a); // get lock on b in other thread.
        System.out.println("Back in other thread");
    }
}

public static void main(String[] args) {
    Deadlock dl = new Deadlock();

    dl.deadlockStart();
}
}
```

When you run this program, you will see the output shown here, although whether **A.foo()** or **B.bar()** executes first will vary based on the specific execution environment.

```
MainThread entered A.foo
RacingThread entered B.bar
MainThread trying to call B.last()
RacingThread trying to call A.last()
```

Because the program has deadlocked, you need to press CTRL-C to end the program. You can see a full thread and monitor cache dump by pressing CTRL-BREAK on a PC. You will see that **RacingThread** owns the monitor on **b**, while it is waiting for the monitor on **a**. At the same time, **MainThread** owns **a** and is waiting to get **b**. This program will never complete. As this example illustrates, if your multithreaded program locks up occasionally, deadlock is one of the first conditions that you should check for.

Suspending, Resuming, and Stopping Threads

Sometimes, suspending execution of a thread is useful. For example, a separate thread can be used to display the time of day. If the user doesn't want a clock, then its thread can be suspended. Whatever the case, suspending a thread is a simple matter. Once suspended, restarting the thread is also a simple matter.

The mechanisms to suspend, stop, and resume threads differ between early versions of Java, such as Java 1.0, and more modern versions, beginning with Java 2. Prior to Java 2, a program used **suspend()**, **resume()**, and **stop()**, which are methods defined by **Thread**, to pause, restart, and stop the execution of a thread. Although these methods seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they must not be used for new Java programs. Here's why. The **suspend()** method of the **Thread** class was deprecated by Java 2 several years ago. This was done because **suspend()** can sometimes cause serious system failures. Assume that a thread has obtained locks on

critical data structures. If that thread is suspended at that point, those locks are not relinquished. Other threads that may be waiting for those resources can be deadlocked.

The **resume()** method is also deprecated. It does not cause problems, but cannot be used without the **suspend()** method as its counterpart.

The **stop()** method of the **Thread** class, too, was deprecated by Java 2. This was done because this method can sometimes cause serious system failures. Assume that a thread is writing to a critically important data structure and has completed only part of its changes. If that thread is stopped at that point, that data structure might be left in a corrupted state. The trouble is that **stop()** causes any lock the calling thread holds to be released. Thus, the corrupted data might be used by another thread that is waiting on the same lock.

Because you can't now use the **suspend()**, **resume()**, or **stop()** methods to control a thread, you might be thinking that no way exists to pause, restart, or terminate a thread. But, fortunately, this is not true. Instead, a thread must be designed so that the **run()** method periodically checks to determine whether that thread should suspend, resume, or stop its own execution. Typically, this is accomplished by establishing a flag variable that indicates the execution state of the thread. As long as this flag is set to "running," the **run()** method must continue to let the thread execute. If this variable is set to "suspend," the thread must pause. If it is set to "stop," the thread must terminate. Of course, a variety of ways exist in which to write such code, but the central theme will be the same for all programs.

The following example illustrates how the **wait()** and **notify()** methods that are inherited from **Object** can be used to control the execution of a thread. Let us consider its operation. The **NewThread** class contains a **boolean** instance variable named **suspendFlag**, which is used to control the execution of the thread. It is initialized to **false** by the constructor. The **run()** method contains a **synchronized** statement block that checks **suspendFlag**. If that variable is **true**, the **wait()** method is invoked to suspend the execution of the thread. The **mysuspend()** method sets **suspendFlag** to **true**. The **myresume()** method sets **suspendFlag** to **false** and invokes **notify()** to wake up the thread. Finally, the **main()**

method has been modified to invoke the **mysuspend()** and **myresume()** methods.

```
// Suspending and resuming a thread the modern way.
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;
    boolean suspendFlag;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        suspendFlag = false;
    }

    // This is the entry point for thread.
    public void run() {
        try {
            for(int i = 15; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(200);
                synchronized(this) {
                    while(suspendFlag) {
                        wait();
                    }
                }
            }
        } catch (InterruptedException e) {
            System.out.println(name + " interrupted.");
        }
        System.out.println(name + " exiting.");
    }

    synchronized void mysuspend() {
        suspendFlag = true;
    }

    synchronized void myresume() {
        suspendFlag = false;
        notify();
    }
}
```

```
        }
    }

class SuspendResume {
    public static void main(String[] args) {
        NewThread ob1 = new NewThread("One");
        NewThread ob2 = new NewThread("Two");

        ob1.t.start(); // Start the thread
        ob2.t.start(); // Start the thread

        try {
            Thread.sleep(1000);
            ob1.mysuspend();
            System.out.println("Suspending thread One");
            Thread.sleep(1000);
            ob1.myresume();
            System.out.println("Resuming thread One");
            ob2.mysuspend();
            System.out.println("Suspending thread Two");
            Thread.sleep(1000);

            ob2.myresume();
            System.out.println("Resuming thread Two");
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }

        // wait for threads to finish
        try {
            System.out.println("Waiting for threads to finish.");
            ob1.t.join();
            ob2.t.join();
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }

        System.out.println("Main thread exiting.");
    }
}
```

When you run the program, you will see the threads suspend and resume. Later in this book, you will see more examples that use the modern mechanism of thread control. Although this mechanism may not appear as simple to use as the old way, nevertheless, it is the way required to ensure that run-time errors don't occur. It is the approach that *must* be used for all new code.

Obtaining a Thread's State

As mentioned earlier in this chapter, a thread can exist in a number of different states. You can obtain the current state of a thread by calling the `getState()` method defined by **Thread**. It is shown here:

```
Thread.State getState()
```

It returns a value of type **Thread.State** that indicates the state of the thread at the time at which the call was made. **State** is an enumeration defined by **Thread**. (An enumeration is a list of named constants. It is discussed in detail in [Chapter 12](#).) Here are the values that can be returned by `getState()`:

Value	State
BLOCKED	A thread that has suspended execution because it is waiting to acquire a lock.
NEW	A thread that has not begun execution.
RUNNABLE	A thread that either is currently executing or will execute when it gains access to the CPU.
TERMINATED	A thread that has completed execution.
TIMED_WAITING	A thread that has suspended execution for a specified period of time, such as when it has called <code>sleep()</code> . This state is also entered when a timeout version of <code>wait()</code> or <code>join()</code> is called.
WAITING	A thread that has suspended execution because it is waiting for some action to occur. For example, it is waiting because of a call to a non-timeout version of <code>wait()</code> or <code>join()</code> .

Figure 11-1 diagrams how the various thread states relate.

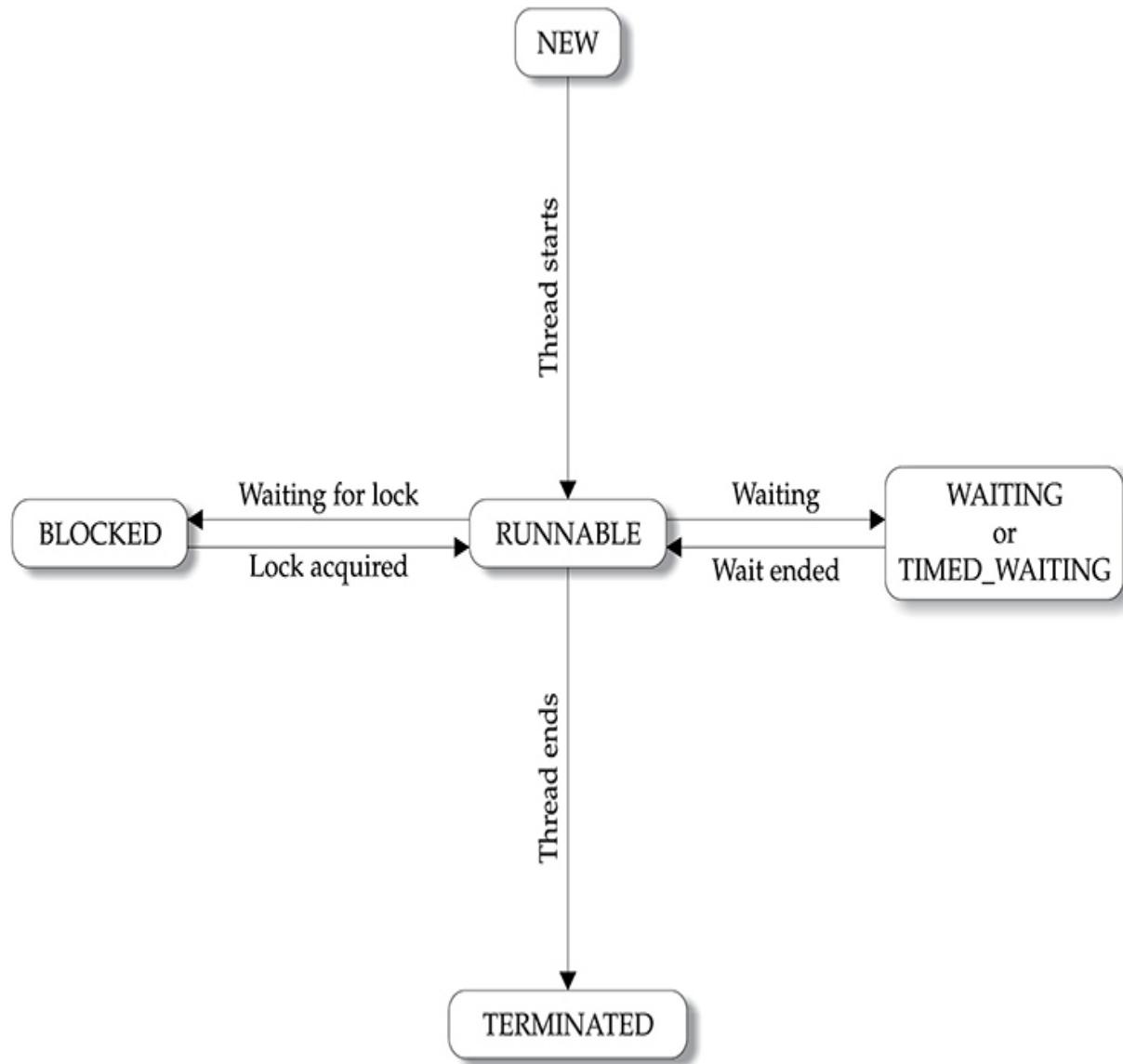


Figure 11-1 Thread states

Given a **Thread** instance, you can use **getState()** to obtain the state of a thread. For example, the following sequence determines if a thread called **thrd** is in the **RUNNABLE** state at the time **getState()** is called:

```
Thread.State ts = thrd.getState();  
if(ts == Thread.State.RUNNABLE) // ...
```

It is important to understand that a thread's state may change after the call to `getState()`. Thus, depending on the circumstances, the state obtained by calling `getState()` may not reflect the actual state of the thread only a moment later. For this (and other) reasons, `getState()` is not intended to provide a means of synchronizing threads. It's primarily used for debugging or for profiling a thread's run-time characteristics.

Using a Factory Method to Create and Start a Thread

In some cases, it is not necessary to separate the creation of a thread from the start of its execution. In other words, sometimes it is convenient to create and start a thread at the same time. One way to do this is to use a static factory method. A *factory method* is a method that returns an object of a class. Typically, factory methods are *static* methods of a class. They are used for a variety of reasons, such as to set an object to some initial state prior to use, to configure a specific type of object, or in some cases to enable an object to be reused. As it relates to creating and starting a thread, a factory method will create the thread, call `start()` on the thread, and then return a reference to the thread. With this approach, you can create and start a thread through a single method call, thus streamlining your code.

For example, assuming the **ThreadDemo** program shown near the start of this chapter, adding the following factory method to **NewThread** enables you to create and start a thread in a single step:

```
// A factory method that creates and starts a thread.
public static NewThread createAndStart() {
    NewThread myThrd = new NewThread();
    myThrd.t.start();
    return myThrd;
}
```

Using `createAndStart()`, you can now replace this sequence:

```
NewThread nt = new NewThread(); // create a new thread  
nt.t.start(); // Start the thread
```

with

```
NewThread nt = NewThread.createAndStart();
```

Now the thread is created and started in one step.

In cases in which you don't need to keep a reference to the executing thread, you can sometimes create and start a thread with one line of code, without the use of a factory method. For example, again assuming the **ThreadDemo** program, the following creates and starts a **NewThread** thread:

```
new NewThread().t.start();
```

However, in real-world applications, you will usually need to keep a reference to the thread, so the factory method is often a good choice.

Using Multithreading

The key to utilizing Java's multithreading features effectively is to think concurrently rather than serially. For example, when you have two subsystems within a program that can execute concurrently, make them individual threads. With the careful use of multithreading, you can create very efficient programs. A word of caution is in order, however: If you create too many threads, you can actually degrade the performance of your program rather than enhance it. Remember, some overhead is associated with context switching. If you create too many threads, more CPU time will be spent changing contexts than executing your program! One last point: To create compute-intensive applications that can automatically scale to make use of the available processors in a multicore system, consider using the Fork/Join Framework, which is described in [Chapter 29](#).

CHAPTER 12

Enumerations, Autoboxing, and Annotations

This chapter examines three features that were not originally part of Java, but over time each has become a near indispensable aspect of Java programming: enumerations, autoboxing, and annotations. Originally added by JDK 5, each is a feature upon which Java programmers have come to rely because each offers a streamlined approach to handling common programming tasks. This chapter also discusses Java's type wrappers and introduces reflection.

Enumerations

In its simplest form, an *enumeration* is a list of named constants that define a new data type and its legal values. Thus, an enumeration object can hold only a value that was declared in the list. Other values are not allowed. In other words, an enumeration gives you a way to explicitly specify the only values that a data type can legally have. Enumerations are commonly used to define a set of values that represent a collection of items. For example, you might use an enumeration to represent the error codes that can result from some operation, such as success, failed, or pending; or a list of the states that a device might be in, such as running, stopped, or paused. In early versions of Java, such values were defined using **final** variables, but enumerations offer a far superior approach.

Although Java enumerations might, at first glance, appear similar to enumerations in other languages, this similarity may be only skin deep

because, in Java, an enumeration defines a class type. By making enumerations into classes, the capabilities of the enumeration are greatly expanded. For example, in Java, an enumeration can have constructors, methods, and instance variables. Because of their power and flexibility, enumerations are widely used throughout the Java API library.

Enumeration Fundamentals

An enumeration is created using the **enum** keyword. For example, here is a simple enumeration that lists various apple varieties:

```
// An enumeration of apple varieties.  
enum Apple {  
    Jonathan, GoldenDel, RedDel, Winesap, Cortland  
}
```

The identifiers **Jonathan**, **GoldenDel**, and so on, are called *enumeration constants*. Each is implicitly declared as a public, static final member of **Apple**. Furthermore, their type is the type of the enumeration in which they are declared, which is **Apple** in this case. Thus, in the language of Java, these constants are called *self-typed*, in which “self” refers to the enclosing enumeration.

Once you have defined an enumeration, you can create a variable of that type. However, even though enumerations define a class type, you do not instantiate an **enum** using **new**. Instead, you declare and use an enumeration variable in much the same way as you do one of the primitive types. For example, this declares **ap** as a variable of enumeration type **Apple**:

```
Apple ap;
```

Because **ap** is of type **Apple**, the only values that it can be assigned (or can contain) are those defined by the enumeration. For example, this assigns **ap** the value **RedDel**:

```
ap = Apple.RedDel;
```

Notice that the symbol **RedDel** is preceded by **Apple**.

Two enumeration constants can be compared for equality by using the `=` relational operator. For example, this statement compares the value in `ap` with the **GoldenDel** constant:

```
if(ap == Apple.GoldenDel) // ...
```

An enumeration value can also be used to control a **switch** statement. Of course, all of the **case** statements must use constants from the same **enum** as that used by the **switch** expression. For example, this **switch** is perfectly valid:

```
// Use an enum to control a switch statement.
switch(ap) {
    case Jonathan:
        // ...
    case Winesap:
        // ...
```

Notice that in the **case** statements, the names of the enumeration constants are used without being qualified by their enumeration type name. That is, **Winesap**, not **Apple.Winesap**, is used. This is because the type of the enumeration in the **switch** expression has already implicitly specified the **enum** type of the **case** constants. There is no need to qualify the constants in the **case** statements with their **enum** type name. In fact, attempting to do so will cause a compilation error.

When an enumeration constant is displayed, such as in a **println()** statement, its name is output. For example, given this statement:

```
System.out.println(Apple.Winesap);
```

the name **Winesap** is displayed.

The following program puts together all of the pieces and demonstrates the **Apple** enumeration:

```
// An enumeration of apple varieties.
enum Apple {
    Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo {
    public static void main(String[] args)
    {
        Apple ap;

        ap = Apple.RedDel;

        // Output an enum value.
        System.out.println("Value of ap: " + ap);
        System.out.println();

        ap = Apple.GoldenDel;

        // Compare two enum values.
        if(ap == Apple.GoldenDel)
            System.out.println("ap contains GoldenDel.\n");

        // Use an enum to control a switch statement.
        switch(ap) {
            case Jonathan:
                System.out.println("Jonathan is red.");
                break;
            case GoldenDel:
                System.out.println("Golden Delicious is yellow.");
                break;
            case RedDel:
                System.out.println("Red Delicious is red.");
                break;
            case Winesap:
                System.out.println("Winesap is red.");
                break;
            case Cortland:
                System.out.println("Cortland is red.");
                break;
        }
    }
}
```

,

The output from the program is shown here:

```
// Use the built-in enumeration methods.

// An enumeration of apple varieties.
enum Apple {
    Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo2 {
    public static void main(String[] args)
    {
        Apple ap;

        System.out.println("Here are all Apple constants:");

        // use values()
        Apple[] allapples = Apple.values();
        for(Apple a : allapples)
            System.out.println(a);

        System.out.println();

        // use valueOf()
        ap = Apple.valueOf("Winesap");
        System.out.println("ap contains " + ap);

    }
}
```

The **values()** and **valueOf()** Methods

All enumerations automatically contain two predefined methods: **values()** and **valueOf()**. Their general forms are shown here:

```
public static enum-type [ ] values()
public static enum-type valueOf(String str)
```

The **values()** method returns an array that contains a list of the enumeration constants. The **valueOf()** method returns the enumeration constant whose value corresponds to the string passed in *str*. In both cases, *enum-type* is the type of the enumeration. For example, in the case of the **Apple** enumeration shown earlier, the return type of **Apple.valueOf("Winesap")** is **Winesap**.

The following program demonstrates the **values()** and **valueOf()** methods:

```
// Use the built-in enumeration methods.

// An enumeration of apple varieties.
enum Apple {
    Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo2 {
    public static void main(String[] args)
    {
        Apple ap;

        System.out.println("Here are all Apple constants:");

        // use values()
        Apple[] allapples = Apple.values();
        for(Apple a : allapples)
            System.out.println(a);

        System.out.println();

        // use valueOf()
        ap = Apple.valueOf("Winesap");
        System.out.println("ap contains " + ap);
    }
}
```

The output from the program is shown here:

```
Here are all Apple constants:
```

```
Jonathan  
GoldenDel  
RedDel  
Winesap  
Cortland
```

```
ap contains Winesap
```

Notice that this program uses a for-each style **for** loop to cycle through the array of constants obtained by calling **values()**. For the sake of illustration, the variable **allapples** was created and assigned a reference to the enumeration array. However, this step is not necessary because the **for** could have been written as shown here, eliminating the need for the **allapples** variable:

```
for(Apple a : Apple.values())  
    System.out.println(a);
```

Now, notice how the value corresponding to the name **Winesap** was obtained by calling **valueOf()**.

```
ap = Apple.valueOf("Winesap");
```

As explained, **valueOf()** returns the enumeration value associated with the name of the constant represented as a string.

Java Enumerations Are Class Types

As mentioned, a Java enumeration is a class type. Although you don't instantiate an **enum** using **new**, it otherwise has much the same capabilities as other classes. The fact that **enum** defines a class gives the Java enumeration extraordinary power. For example, you can give them constructors, add instance variables and methods, and even implement interfaces.

It is important to understand that each enumeration constant is an object of its enumeration type. Thus, when you define a constructor for an **enum**, the constructor is called when each enumeration constant is created. Also, each enumeration constant has its own copy of any instance

variables defined by the enumeration. For example, consider the following version of **Apple**:

```
// Use an enum constructor, instance variable, and method.
enum Apple {
    Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);

    private int price; // price of each apple

    // Constructor
    Apple(int p) { price = p; }

    int getPrice() { return price; }
}

class EnumDemo3 {
    public static void main(String[] args)
    {
        Apple ap;

        // Display price of Winesap.
        System.out.println("Winesap costs " +
                           Apple.Winesap.getPrice() +
                           " cents.\n");

        // Display all apples and prices.
        System.out.println("All apple prices:");
        for(Apple a : Apple.values())
            System.out.println(a + " costs " + a.getPrice() +
                               " cents.");
    }
}
```

The output is shown here:

```
Winesap costs 15 cents.
```

```
All apple prices:  
Jonathan costs 10 cents.  
GoldenDel costs 9 cents.  
RedDel costs 12 cents.  
Winesap costs 15 cents.  
Cortland costs 8 cents.
```

This version of **Apple** adds three things. The first is the instance variable **price**, which is used to hold the price of each variety of apple. The second is the **Apple** constructor, which is passed the price of an apple. The third is the method **getPrice()**, which returns the value of **price**.

When the variable **ap** is declared in **main()**, the constructor for **Apple** is called once for each constant that is specified. Notice how the arguments to the constructor are specified, by putting them inside parentheses after each constant, as shown here:

```
Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);
```

These values are passed to the **p** parameter of **Apple()**, which then assigns this value to **price**. Again, the constructor is called once for each constant.

Because each enumeration constant has its own copy of **price**, you can obtain the price of a specified type of apple by calling **getPrice()**. For example, in **main()** the price of a Winesap is obtained by the following call:

```
Apple.Winesap.getPrice()
```

The prices of all varieties are obtained by cycling through the enumeration using a **for** loop. Because there is a copy of **price** for each enumeration constant, the value associated with one constant is separate and distinct from the value associated with another constant. This is a powerful concept, which is only available when enumerations are implemented as classes, as Java does.

Although the preceding example contains only one constructor, an **enum** can offer two or more overloaded forms, just as can any other class. For example, this version of **Apple** provides a default constructor that initializes the price to -1, to indicate that no price data is available:

```

// Use an enum constructor.
enum Apple {
    Jonathan(10), GoldenDel(9), RedDel, Winesap(15), Cortland(8);

    private int price; // price of each apple

    // Constructor
    Apple(int p) { price = p; }

    // Overloaded constructor
    Apple() { price = -1; }

    int getPrice() { return price; }
}

```

Notice that in this version, **RedDel** is not given an argument. This means that the default constructor is called, and **RedDel**'s price variable is given the value **-1**.

Here are two restrictions that apply to enumerations. First, an enumeration can't inherit another class. Second, an **enum** cannot be a superclass. This means that an **enum** can't be extended. Otherwise, **enum** acts much like any other class type. The key is to remember that each of the enumeration constants is an object of the class in which it is defined.

Enumerations Inherit Enum

Although you can't inherit a superclass when declaring an **enum**, all enumerations automatically inherit one: **java.lang.Enum**. This class defines several methods that are available for use by all enumerations. The **Enum** class is described in detail in [Part II](#), but three of its methods warrant a discussion at this time.

You can obtain a value that indicates an enumeration constant's position in the list of constants. This is called its *ordinal value*, and it is retrieved by calling the **ordinal()** method, shown here:

```
final int ordinal()
```

It returns the ordinal value of the invoking constant. Ordinal values begin at zero. Thus, in the **Apple** enumeration, **Jonathan** has an ordinal value of

zero, **GoldenDel** has an ordinal value of 1, **RedDel** has an ordinal value of 2, and so on.

You can compare the ordinal value of two constants of the same enumeration by using the **compareTo()** method. It has this general form:

```
final int compareTo(enum-type e)
```

Here, *enum-type* is the type of the enumeration, and *e* is the constant being compared to the invoking constant. Remember, both the invoking constant and *e* must be of the same enumeration. If the invoking constant has an ordinal value less than *e*'s, then **compareTo()** returns a negative value. If the two ordinal values are the same, then zero is returned. If the invoking constant has an ordinal value greater than *e*'s, then a positive value is returned.

You can compare for equality an enumeration constant with any other object by using **equals()**, which overrides the **equals()** method defined by **Object**. Although **equals()** can compare an enumeration constant to any other object, those two objects will be equal only if they both refer to the same constant, within the same enumeration. Simply having ordinal values in common will not cause **equals()** to return true if the two constants are from different enumerations.

Remember, you can compare two enumeration references for equality by using `==`.

The following program demonstrates the **ordinal()**, **compareTo()**, and **equals()** methods:

```
// Demonstrate ordinal(), compareTo(), and equals().  
  
// An enumeration of apple varieties.  
enum Apple {  
    Jonathan, GoldenDel, RedDel, Winesap, Cortland  
}  
  
class EnumDemo4 {  
    public static void main(String[] args)  
    {  
        Apple ap, ap2, ap3;  
  
        // Obtain all ordinal values using ordinal().  
        System.out.println("Here are all apple constants" +  
                           " and their ordinal values: ");  
        for(Apple a : Apple.values())  
            System.out.println(a + " " + a.ordinal());  
  
        ap = Apple.RedDel;  
        ap2 = Apple.GoldenDel;  
        ap3 = Apple.RedDel;  
  
        System.out.println();  
  
        // Demonstrate compareTo() and equals()  
        if(ap.compareTo(ap2) < 0)  
            System.out.println(ap + " comes before " + ap2);  
  
        if(ap.compareTo(ap2) > 0)  
            System.out.println(ap2 + " comes before " + ap);  
  
        if(ap.compareTo(ap3) == 0)  
            System.out.println(ap + " equals " + ap3);  
  
        System.out.println();  
  
        if(ap.equals(ap2))  
            System.out.println("Error!");  
  
        if(ap.equals(ap3))  
            System.out.println(ap + " equals " + ap3);
```

```
    if(ap == ap3)
        System.out.println(ap + " == " + ap3);

    }
}
```

The output from the program is shown here:

```
Here are all apple constants and their ordinal values:
Jonathan 0
GoldenDel 1
RedDel 2
Winesap 3
Cortland 4

GoldenDel comes before RedDel
RedDel equals RedDel

RedDel equals RedDel
RedDel == RedDel
```

Another Enumeration Example

Before moving on, we will look at a different example that uses an **enum**. In [Chapter 9](#), an automated “decision maker” program was created. In that version, variables called **NO**, **YES**, **MAYBE**, **LATER**, **SOON**, and **NEVER** were declared within an interface and used to represent the possible answers. While there is nothing technically wrong with that approach, the enumeration is a better choice. Here is an improved version of that program that uses an **enum** called **Answers** to define the answers. You should compare this version to the original in [Chapter 9](#).

```
// An improved version of the "Decision Maker"
// program from Chapter 9. This version uses an
// enum, rather than interface variables, to
// represent the answers.

import java.util.Random;

// An enumeration of the possible answers.
enum Answers {
    NO, YES, MAYBE, LATER, SOON, NEVER
}

class Question {
    Random rand = new Random();
    Answers ask() {
        int prob = (int) (100 * rand.nextDouble());

        if (prob < 15)
            return Answers.MAYBE; // 15%
        else if (prob < 30)
            return Answers.NO;    // 15%
        else if (prob < 60)
            return Answers.YES;   // 30%
```

```
        else if (prob < 75)
            return Answers.LATER; // 15%
        else if (prob < 98)
            return Answers.SOON; // 13%
        else
            return Answers.NEVER; // 2%
    }
}

class AskMe {
    static void answer(Answers result) {
        switch(result) {
            case NO:
                System.out.println("No");
                break;
            case YES:
                System.out.println("Yes");
                break;
            case MAYBE:
                System.out.println("Maybe");
                break;
            case LATER:
                System.out.println("Later");
                break;
            case SOON:
                System.out.println("Soon");
                break;
            case NEVER:
                System.out.println("Never");
                break;
        }
    }
}

public static void main(String[] args) {
    Question q = new Question();
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
}
}
```

Type Wrappers

As you know, Java uses primitive types (also called simple types), such as **int** or **double**, to hold the basic data types supported by the language. Primitive types, rather than objects, are used for these quantities for the sake of performance. Using objects for these values would add an unacceptable overhead to even the simplest of calculations. Thus, the primitive types are not part of the object hierarchy, and they do not inherit **Object**.

Despite the performance benefit offered by the primitive types, there are times when you will need an object representation. For example, you can't pass a primitive type by reference to a method. Also, many of the standard data structures implemented by Java operate on objects, which means that you can't use these data structures to store primitive types. To handle these (and other) situations, Java provides *type wrappers*, which are classes that encapsulate a primitive type within an object. The type wrapper classes are described in detail in [Part II](#), but they are introduced here because they relate directly to Java's autoboxing feature.

The type wrappers are **Double**, **Float**, **Long**, **Integer**, **Short**, **Byte**, **Character**, and **Boolean**. These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy. Each is briefly examined next.

Character

Character is a wrapper around a **char**. The constructor for **Character** is

`Character(char ch)`

Here, *ch* specifies the character that will be wrapped by the **Character** object being created.

However, beginning with JDK 9, the **Character** constructor was deprecated, and beginning with JDK 16, it has been deprecated for removal. Today, it is strongly recommended that you use the static method **valueOf()** to obtain a **Character** object. It is shown here:

`static Character valueOf(char ch)`

It returns a **Character** object that wraps *ch*.

To obtain the **char** value contained in a **Character** object, call **charValue()**, shown here:

```
char charValue()
```

It returns the encapsulated character.

Boolean

Boolean is a wrapper around **boolean** values. It defines these constructors:

```
Boolean(boolean boolValue)  
Boolean(String boolString)
```

In the first version, *boolValue* must be either **true** or **false**. In the second version, if *boolString* contains the string "true" (in uppercase or lowercase), then the new **Boolean** object will be true. Otherwise, it will be false.

However, beginning with JDK 9, the **Boolean** constructors were deprecated, and beginning with JDK 16, they have been deprecated for removal. Today, it is strongly recommended that you use the static method **valueOf()** to obtain a **Boolean** object. It has the two versions shown here:

```
static Boolean valueOf(boolean boolValue)  
static Boolean valueOf(String boolString)
```

Each returns a **Boolean** object that wraps the indicated value.

To obtain a **boolean** value from a **Boolean** object, use **booleanValue()**, shown here:

```
boolean booleanValue()
```

It returns the **boolean** equivalent of the invoking object.

The Numeric Type Wrappers

By far, the most commonly used type wrappers are those that represent numeric values. These are **Byte**, **Short**, **Integer**, **Long**, **Float**, and **Double**.

All of the numeric type wrappers inherit the abstract class **Number**. **Number** declares methods that return the value of an object in each of the different number formats. These methods are shown here:

```
byte byteValue()
double doubleValue()
float floatValue()
int intValue()
long longValue()
short shortValue()
```

For example, **doubleValue()** returns the value of an object as a **double**, **floatValue()** returns the value as a **float**, and so on. These methods are implemented by each of the numeric type wrappers.

All of the numeric type wrappers define constructors that allow an object to be constructed from a given value, or a string representation of that value. For example, here are the constructors defined for **Integer**:

```
Integer(int num)
Integer(String str)
```

If *str* does not contain a valid numeric value, then a **NumberFormatException** is thrown.

However, beginning with JDK 9, the numeric type-wrapper constructors were deprecated, and beginning with JDK 16, they have been deprecated for removal. Today, it is strongly recommended that you use one of the **valueOf()** methods to obtain a wrapper object. The **valueOf()** method is a static member of all of the numeric wrapper classes and all numeric classes support forms that convert a numeric value or a string into an object. For example, here are two of the forms supported by **Integer**:

```
static Integer valueOf(int val)
static Integer valueOf(String valStr) throws NumberFormatException
```

Here, *val* specifies an integer value and *valStr* specifies a string that represents a properly formatted numeric value in string form. Each returns an **Integer** object that wraps the specified value. Here is an example:

```
Integer iOb = Integer.valueOf(100);
```

After this statement executes, the value 100 is represented by an **Integer** instance. Thus, **iOb** wraps the value 100 within an object. In addition to the forms **valueOf()** just shown, the integer wrappers, **Byte**, **Short**, **Integer**, and **Long**, also supply a form that lets you specify a radix.

All of the type wrappers override **toString()**. It returns the human-readable form of the value contained within the wrapper. This allows you to output the value by passing a type wrapper object to **println()**, for example, without having to convert it into its primitive type.

The following program demonstrates how to use a numeric type wrapper to encapsulate a value and then extract that value.

```
// Demonstrate a type wrapper.
class Wrap {
    public static void main(String[] args) {

        Integer iOb = Integer.valueOf(100);

        int i = iOb.intValue();

        System.out.println(i + " " + iOb); // displays 100 100
    }
}
```

This program wraps the integer value 100 inside an **Integer** object called **iOb**. The program then obtains this value by calling **intValue()** and stores the result in **i**.

The process of encapsulating a value within an object is called *boxing*. Thus, in the program, this line boxes the value 100 into an **Integer**:

```
Integer iOb = Integer.valueOf(100);
```

The process of extracting a value from a type wrapper is called *unboxing*. For example, the program unboxes the value in **iOb** with this statement:

```
int i = iOb.intValue();
```

The same general procedure used by the preceding program to box and unbox values has been available for use since the original version of Java.

However, today, Java provides a more streamlined approach, which is described next.

Autoboxing

Modern versions of Java have included two important features: *autoboxing* and *auto-unboxing*. Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed. There is no need to explicitly construct an object. Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed. There is no need to call a method such as **intValue()** or **doubleValue()**.

Autoboxing and auto-unboxing greatly streamline the coding of several algorithms, removing the tedium of manually boxing and unboxing values. They also help prevent errors. Moreover, they are very important to generics, which operate only on objects. Finally, autoboxing makes working with the Collections Framework (described in [Part II](#)) much easier.

With autoboxing, it is not necessary to manually construct an object in order to wrap a primitive type. You need only assign that value to a type-wrapper reference. Java automatically constructs the object for you. For example, here is the modern way to construct an **Integer** object that has the value 100:

```
Integer iOb = 100; // autobox an int
```

Notice that the object is not explicitly boxed. Java handles this for you, automatically.

To unbox an object, simply assign that object reference to a primitive-type variable. For example, to unbox **iOb**, you can use this line:

```
int i = iOb; // auto-unbox
```

Java handles the details for you.

Here is the preceding program rewritten to use autoboxing/unboxing:

```
// Demonstrate autoboxing/unboxing.
class AutoBox {
    public static void main(String[] args) {

        Integer iOb = 100; // autobox an int

        int i = iOb; // auto-unbox

        System.out.println(i + " " + iOb); // displays 100 100
    }
}
```

Autoboxing and Methods

In addition to the simple case of assignments, autoboxing automatically occurs whenever a primitive type must be converted into an object; auto-unboxing takes place whenever an object must be converted into a primitive type. Thus, autoboxing/unboxing might occur when an argument is passed to a method, or when a value is returned by a method. For example, consider this:

```

// Autoboxing/unboxing takes place with
// method parameters and return values.

class AutoBox2 {
    // Take an Integer parameter and return
    // an int value;
    static int m(Integer v) {
        return v ; // auto-unbox to int
    }

    public static void main(String[] args) {
        // Pass an int to m() and assign the return value
        // to an Integer. Here, the argument 100 is autoboxed
        // into an Integer. The return value is also autoboxed
        // into an Integer.
        Integer iOb = m(100);

        System.out.println(iOb);
    }
}

```

This program displays the following result:

100

In the program, notice that **m()** specifies an **Integer** parameter and returns an **int** result. Inside **main()**, **m()** is passed the value 100. Because **m()** is expecting an **Integer**, this value is automatically boxed. Then, **m()** returns the **int** equivalent of its argument. This causes **v** to be auto-unboxed. Next, this **int** value is assigned to **iOb** in **main()**, which causes the **int** return value to be autoboxed.

Autoboxing/Unboxing Occurs in Expressions

In general, autoboxing and unboxing take place whenever a conversion into an object or from an object is required. This applies to expressions. Within an expression, a numeric object is automatically unboxed. The outcome of the expression is reboxed, if necessary. For example, consider the following program:

```
// Autoboxing/unboxing occurs inside expressions.

class AutoBox3 {
    public static void main(String[] args) {

        Integer iOb, iOb2;
        int i;

        iOb = 100;
        System.out.println("Original value of iOb: " + iOb);

        // The following automatically unboxes iOb,
        // performs the increment, and then reboxes
        // the result back into iOb.
        ++iOb;
        System.out.println("After ++iOb: " + iOb);

        // Here, iOb is unboxed, the expression is
        // evaluated, and the result is reboxed and
        // assigned to iOb2.
        iOb2 = iOb + (iOb / 3);
        System.out.println("iOb2 after expression: " + iOb2);

        // The same expression is evaluated, but the
        // result is not reboxed.
        i = iOb + (iOb / 3);
        System.out.println("i after expression: " + i);

    }
}
```

The output is shown here:

```
Original value of iOb: 100
After ++iOb: 101
iOb2 after expression: 134
i after expression: 134
```

In the program, pay special attention to this line:

```
++iOb;
```

This causes the value in **iOb** to be incremented. It works like this: **iOb** is unboxed, the value is incremented, and the result is reboxed.

Auto-unboxing also allows you to mix different types of numeric objects in an expression. Once the values are unboxed, the standard type promotions and conversions are applied. For example, the following program is perfectly valid:

```
class AutoBox4 {  
    public static void main(String[] args) {  
  
        Integer iOb = 100;  
        Double dOb = 98.6;  
  
        dOb = dOb + iOb;  
        System.out.println("dOb after expression: " + dOb);  
    }  
}
```

The output is shown here:

```
dOb after expression: 198.6
```

As you can see, both the **Double** object **dOb** and the **Integer** object **iOb** participated in the addition, and the result was reboxed and stored in **dOb**.

Because of auto-unboxing, you can use **Integer** numeric objects to control a **switch** statement. For example, consider this fragment:

```
Integer iOb = 2;  
  
switch(iOb) {  
    case 1: System.out.println("one");  
    break;  
    case 2: System.out.println("two");  
    break;  
    default: System.out.println("error");  
}
```

When the **switch** expression is evaluated, **iOb** is unboxed and its **int** value is obtained.

As the examples in the programs show, because of autoboxing/unboxing, using numeric objects in an expression is both intuitive and easy. In the early days of Java, such code would have involved casts and calls to methods such as `intValue()`.

Autoboxing/Unboxing Boolean and Character Values

As described earlier, Java also supplies wrappers for `boolean` and `char`. These are **Boolean** and **Character**. Autoboxing/unboxing applies to these wrappers, too. For example, consider the following program:

```
// Autoboxing/unboxing a Boolean and Character.

class AutoBox5 {
    public static void main(String[] args) {
        // Autobox/unbox a boolean.
        Boolean b = true;

        // Below, b is auto-unboxed when used in
        // a conditional expression, such as an if.
        if(b) System.out.println("b is true");

        // Autobox/unbox a char.
        Character ch = 'x'; // box a char
        char ch2 = ch; // unbox a char

        System.out.println("ch2 is " + ch2);
    }
}
```

The output is shown here:

```
b is true
ch2 is x
```

The most important thing to notice about this program is the auto-unboxing of **b** inside the **if** conditional expression. As you should recall, the conditional expression that controls an **if** must evaluate to type

boolean. Because of auto-unboxing, the **boolean** value contained within **b** is automatically unboxed when the conditional expression is evaluated. Thus, with autoboxing/unboxing, a **Boolean** object can be used to control an **if** statement.

Because of auto-unboxing, a **Boolean** object can now also be used to control any of Java's loop statements. When a **Boolean** is used as the conditional expression of a **while**, **for**, or **do/while**, it is automatically unboxed into its **boolean** equivalent. For example, this is perfectly valid code:

```
Boolean b;  
// ...  
while(b) { // ...
```

Autoboxing/Unboxing Helps Prevent Errors

In addition to the convenience that it offers, autoboxing/unboxing can also help prevent errors. For example, consider the following program:

```
// An error produced by manual unboxing.  
class UnboxingError {  
    public static void main(String[] args) {  
  
        Integer iOb = 1000; // autobox the value 1000  
  
        int i = iOb.byteValue(); // manually unbox as byte !!!  
  
        System.out.println(i); // does not display 1000 !  
    }  
}
```

This program displays not the expected value of 1000, but –24! The reason is that the value inside **iOb** is manually unboxed by calling **byteValue()**, which causes the truncation of the value stored in **iOb**, which is 1,000. This results in the garbage value of –24 being assigned to **i**. Auto-unboxing prevents this type of error because the value in **iOb** will always auto-unbox into a value compatible with **int**.

In general, because autoboxing always creates the proper object, and auto-unboxing always produces the proper value, there is no way for the process to produce the wrong type of object or value. In the rare instances where you want a type different than that produced by the automated process, you can still manually box and unbox values. Of course, the benefits of autoboxing/unboxing are lost. In general, you should employ autoboxing/unboxing. It is the way that modern Java code is written.

A Word of Warning

Because of autoboxing and auto-unboxing, some might be tempted to use objects such as **Integer** or **Double** exclusively, abandoning primitives altogether. For example, with autoboxing/unboxing it is possible to write code like this:

```
// A bad use of autoboxing/unboxing!
Double a, b, c;

a = 10.0;
b = 4.0;

c = Math.sqrt(a*a + b*b);

System.out.println("Hypotenuse is " + c);
```

In this example, objects of type **Double** hold values that are used to calculate the hypotenuse of a right triangle. Although this code is technically correct and does, in fact, work properly, it is a very bad use of autoboxing/unboxing. It is far less efficient than the equivalent code written using the primitive type **double**. The reason is that each autobox and auto-unbox adds overhead that is not present if the primitive type is used.

In general, you should restrict your use of the type wrappers to only those cases in which an object representation of a primitive type is required. Autoboxing/unboxing was not added to Java as a “back door” way of eliminating the primitive types.

Annotations

Java provides a feature that enables you to embed supplemental information into a source file. This information, called an *annotation*, does not change the actions of a program. Thus, an annotation leaves the semantics of a program unchanged. However, this information can be used by various tools during both development and deployment. For example, an annotation might be processed by a source-code generator. The term *metadata* is also used to refer to this feature, but the term *annotation* is the most descriptive and more commonly used.

Annotation Basics

An annotation is created through a mechanism based on the **interface**. Let's begin with an example. Here is the declaration for an annotation called **MyAnno**:

```
// A simple annotation type.  
@interface MyAnno {  
    String str();  
    int val();  
}
```

First, notice the @ that precedes the keyword **interface**. This tells the compiler that an annotation type is being declared. Next, notice the two members **str()** and **val()**. All annotations consist solely of method declarations. However, you don't provide bodies for these methods. Instead, Java implements these methods. Moreover, the methods act much like fields, as you will see.

An annotation cannot include an **extends** clause. However, all annotation types automatically extend the **Annotation** interface. Thus, **Annotation** is a super-interface of all annotations. It is declared within the **java.lang.annotation** package. It overrides **hashCode()**, **equals()**, and **toString()**, which are defined by **Object**. It also specifies **annotationType()**, which returns a **Class** object that represents the invoking annotation.

Once you have declared an annotation, you can use it to annotate something. Initially, annotations could be used only on declarations, and that is where we will begin. (JDK 8 added the ability to annotate type use, and this is described later in this chapter. However, the same basic techniques apply to both kinds of annotations.) Any type of declaration can have an annotation associated with it. For example, classes, methods, fields, parameters, and **enum** constants can be annotated. Even an annotation can be annotated. In all cases, the annotation precedes the rest of the declaration.

When you apply an annotation, you give values to its members. For example, here is an example of **MyAnno** being applied to a method declaration:

```
// Annotate a method.  
@MyAnno(str = "Annotation Example", val = 100)  
public static void myMeth() { // ...}
```

This annotation is linked with the method **myMeth()**. Look closely at the annotation syntax. The name of the annotation, preceded by an **@**, is followed by a parenthesized list of member initializations. To give a member a value, that member's name is assigned a value. Therefore, in the example, the string "Annotation Example" is assigned to the **str** member of **MyAnno**. Notice that no parentheses follow **str** in this assignment. When an annotation member is given a value, only its name is used. Thus, annotation members look like fields in this context.

Specifying a Retention Policy

Before exploring annotations further, it is necessary to discuss *annotation retention policies*. A retention policy determines at what point an annotation is discarded. Java defines three such policies, which are encapsulated within the **java.lang.annotation.RetentionPolicy** enumeration. They are **SOURCE**, **CLASS**, and **RUNTIME**.

An annotation with a retention policy of **SOURCE** is retained only in the source file and is discarded during compilation.

An annotation with a retention policy of **CLASS** is stored in the **.class** file during compilation. However, it is not available through the JVM

during run time.

An annotation with a retention policy of **RUNTIME** is stored in the **.class** file during compilation and is available through the JVM during run time. Thus, **RUNTIME** retention offers the greatest annotation persistence.

Note An annotation on a local variable declaration is not retained in the **.class** file.

A retention policy is specified for an annotation by using one of Java's built-in annotations: **@Retention**. Its general form is shown here:

@Retention(*retention-policy*)

Here, *retention-policy* must be one of the previously discussed enumeration constants. If no retention policy is specified for an annotation, then the default policy of **CLASS** is used.

The following version of **MyAnno** uses **@Retention** to specify the **RUNTIME** retention policy. Thus, **MyAnno** will be available to the JVM during program execution.

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str();
    int val();
}
```

Obtaining Annotations at Run Time by Use of Reflection

Although annotations are designed mostly for use by other development or deployment tools, if they specify a retention policy of **RUNTIME**, then they can be queried at run time by any Java program through the use of *reflection*. Reflection is the feature that enables information about a class to be obtained at run time. The reflection API is contained in the **java.lang.reflect** package. There are a number of ways to use reflection,

and we won't examine them all here. We will, however, walk through a few examples that apply to annotations.

The first step to using reflection is to obtain a **Class** object that represents the class whose annotations you want to obtain. **Class** is one of Java's built-in classes and is defined in **java.lang**. It is described in detail in [Part II](#). There are various ways to obtain a **Class** object. One of the easiest is to call **getClass()**, which is a method defined by **Object**. Its general form is shown here:

```
final Class<?> getClass()
```

It returns the **Class** object that represents the invoking object.

Note Notice the `<?>` that follows **Class** in the declaration of **getClass()** just shown. This is related to Java's generics feature. **getClass()** and several other reflection-related methods discussed in this chapter make use of generics. Generics are described in [Chapter 14](#). However, an understanding of generics is not needed to grasp the fundamental principles of reflection.

After you have obtained a **Class** object, you can use its methods to obtain information about the various items declared by the class, including its annotations. If you want to obtain the annotations associated with a specific item declared within a class, you must first obtain an object that represents that item. For example, **Class** supplies (among others) the **getMethod()**, **getField()**, and **getConstructor()** methods, which obtain information about a method, field, and constructor, respectively. These methods return objects of type **Method**, **Field**, and **Constructor**.

To understand the process, let's work through an example that obtains the annotations associated with a method. To do this, you first obtain a **Class** object that represents the class, and then call **getMethod()** on that **Class** object, specifying the name of the method. **getMethod()** has this general form:

```
Method getMethod(String methName, Class<?> ... paramTypes)
```

The name of the method is passed in *methName*. If the method has arguments, then **Class** objects representing those types must also be

specified by *paramTypes*. Notice that *paramTypes* is a varargs parameter. This means that you can specify as many parameter types as needed, including zero. **getMethod()** returns a **Method** object that represents the method. If the method can't be found, **NoSuchMethodException** is thrown.

From a **Class**, **Method**, **Field**, or **Constructor** object, you can obtain a specific annotation associated with that object by calling **getAnnotation()**. Its general form is shown here:

```
<A extends Annotation> getAnnotation(Class<A> annoType)
```

Here, *annoType* is a **Class** object that represents the annotation in which you are interested. The method returns a reference to the annotation. Using this reference, you can obtain the values associated with the annotation's members. The method returns **null** if the annotation is not found, which will be the case if the annotation does not have **RUNTIME** retention.

Here is a program that assembles all of the pieces shown earlier and uses reflection to display the annotation associated with a method:

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// An annotation type declaration.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str();
    int val();
}

class Meta {

    // Annotate a method.
    @MyAnno(str = "Annotation Example", val = 100)
    public static void myMeth() {
        Meta ob = new Meta();

        // Obtain the annotation for this method
        // and display the values of the members.
    }
}
```

```

try {
    // First, get a Class object that represents
    // this class.
    Class<?> c = ob.getClass();

    // Now, get a Method object that represents
    // this method.
    Method m = c.getMethod("myMeth");

    // Next, get the annotation for this class.
    MyAnno anno = m.getAnnotation(MyAnno.class);

    // Finally, display the values.
    System.out.println(anno.str() + " " + anno.val());
} catch (NoSuchMethodException exc) {
    System.out.println("Method Not Found.");
}
}

public static void main(String[] args) {
    myMeth();
}
}

```

The output from the program is shown here:

Annotation Example 100

This program uses reflection as described to obtain and display the values of **str** and **val** in the **MyAnno** annotation associated with **myMeth()** in the **Meta** class. There are two things to pay special attention to. First, in this line

`MyAnno anno = m.getAnnotation(MyAnno.class);`

notice the expression **MyAnno.class**. This expression evaluates to a **Class** object of type **MyAnno**, the annotation. This construct is called a *class literal*. You can use this type of expression whenever a **Class** object of a known class is needed. For example, this statement could have been used to obtain the **Class** object for **Meta**:

```
Class<?> c = Meta.class;
```

Of course, this approach only works when you know the class name of an object in advance, which might not always be the case. In general, you can obtain a class literal for classes, interfaces, primitive types, and arrays. (Remember, the <?> syntax relates to Java's generics feature. It is described in [Chapter 14](#).)

The second point of interest is the way the values associated with **str** and **val** are obtained when they are output by the following line:

```
System.out.println(anno.str() + " " + anno.val());
```

Notice that they are invoked using the method-call syntax. This same approach is used whenever the value of an annotation member is required.

A Second Reflection Example

In the preceding example, **myMeth()** has no parameters. Thus, when **getMethod()** was called, only the name **myMeth** was passed. However, to obtain a method that has parameters, you must specify class objects representing the types of those parameters as arguments to **getMethod()**. For example, here is a slightly different version of the preceding program:

```

import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str();
    int val();
}

class Meta {

    // myMeth now has two arguments.
    @MyAnno(str = "Two Parameters", val = 19)
    public static void myMeth(String str, int i)
    {
        Meta ob = new Meta();

        try {
            Class<?> c = ob.getClass();

            // Here, the parameter types are specified.
            Method m = c.getMethod("myMeth", String.class, int.class);

            MyAnno anno = m.getAnnotation(MyAnno.class);

            System.out.println(anno.str() + " " + anno.val());
        } catch (NoSuchMethodException exc) {
            System.out.println("Method Not Found.");
        }
    }

    public static void main(String[] args) {
        myMeth("test", 10);
    }
}

```

The output from this version is shown here:

```
Two Parameters 19
```

In this version, **myMeth()** takes a **String** and an **int** parameter. To obtain information about this method, **getMethod()** must be called as shown here:

```
Method m = c.getMethod("myMeth", String.class, int.class);
```

Here, the **Class** objects representing **String** and **int** are passed as additional arguments.

Obtaining All Annotations

You can obtain all annotations that have **RUNTIME** retention that are associated with an item by calling **getAnnotations()** on that item. It has this general form:

```
Annotation[ ] getAnnotations()
```

It returns an array of the annotations. **getAnnotations()** can be called on objects of type **Class**, **Method**, **Constructor**, and **Field**, among others.

Here is another reflection example that shows how to obtain all annotations associated with a class and with a method. It declares two annotations. It then uses those annotations to annotate a class and a method.

```
// Show all annotations for a class and a method.
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str();
    int val();
}

@Retention(RetentionPolicy.RUNTIME)
@interface What {
    String description();
}

@What(description = "An annotation test class")
@MyAnno(str = "Meta2", val = 99)
class Meta2 {

    @What(description = "An annotation test method")
    @MyAnno(str = "Testing", val = 100)
    public static void myMeth() {
        Meta2 ob = new Meta2();

        try {
            Annotation[] annos = ob.getClass().getAnnotations();

            // Display all annotations for Meta2.
            System.out.println("All annotations for Meta2:");
            for(Annotation a : annos)
                System.out.println(a);

            System.out.println();

            // Display all annotations for myMeth.
            Method m = ob.getClass( ).getMethod("myMeth");
            annos = m.getAnnotations();
        }
    }
}
```

```

        System.out.println("All annotations for myMeth:");
        for(Annotation a : annos)
            System.out.println(a);

    } catch (NoSuchMethodException exc) {
        System.out.println("Method Not Found.");
    }
}

public static void main(String[] args) {
    myMeth();
}
}

```

The output is shown here:

```
All annotations for Meta2:
@What(description=An annotation test class)
@MyAnno(str=Meta2, val=99)
```

```
All annotations for myMeth:
@What(description=An annotation test method)
@MyAnno(str=Testing, val=100)
```

The program uses **getAnnotations()** to obtain an array of all annotations associated with the **Meta2** class and with the **myMeth()** method. As explained, **getAnnotations()** returns an array of **Annotation** objects. Recall that **Annotation** is a super-interface of all annotation interfaces and that it overrides **toString()** in **Object**. Thus, when a reference to an **Annotation** is output, its **toString()** method is called to generate a string that describes the annotation, as the preceding output shows.

The AnnotatedElement Interface

The methods **getAnnotation()** and **getAnnotations()** used by the preceding examples are defined by the **AnnotatedElement** interface, which is defined in **java.lang.reflect**. This interface supports reflection for

annotations and is implemented by the classes **Method**, **Field**, **Constructor**, **Class**, and **Package**, among others.

In addition to **getAnnotation()** and **getAnnotations()**, **AnnotatedElement** defines several other methods. Two have been available since annotations were initially added to Java. The first is **getDeclaredAnnotations()**, which has this general form:

```
Annotation[ ] getDeclaredAnnotations( )
```

It returns all non-inherited annotations present in the invoking object. The second is **isAnnotationPresent()**, which has this general form:

```
default boolean isAnnotationPresent(Class<? extends Annotation>  
 annoType)
```

It returns **true** if the annotation specified by *annoType* is associated with the invoking object. It returns **false** otherwise. To these, JDK 8 added **getDeclaredAnnotation()**, **getAnnotationsByType()**, and **getDeclaredAnnotationsByType()**. Of these, the last two automatically work with a repeated annotation.(Repeated annotations are discussed at the end of this chapter.)

Using Default Values

You can give annotation members default values that will be used if no value is specified when the annotation is applied. A default value is specified by adding a **default** clause to a member's declaration. It has this general form:

```
type member( ) default value ;
```

Here, *value* must be of a type compatible with *type*.

Here is **@MyAnno** rewritten to include default values:

```
// An annotation type declaration that includes defaults.  
@Retention(RetentionPolicy.RUNTIME)  
@interface MyAnno {  
    String str() default "Testing";  
    int val() default 9000;  
}
```

This declaration gives a default value of "Testing" to **str** and 9000 to **val**. This means that neither value needs to be specified when **@MyAnno** is used. However, either or both can be given values if desired. Therefore, following are the four ways that **@MyAnno** can be used:

```
@MyAnno() // both str and val default  
@MyAnno(str = "some string") // val defaults  
@MyAnno(val = 100) // str defaults  
@MyAnno(str = "Testing", val = 100) // no defaults
```

The following program demonstrates the use of default values in an annotation.