

The stack trace will always show the sequence of method invocations that led up to the error. For example, here is another version of the preceding program that introduces the same error but in a method separate from **main()**:

```
class Exc1 {
    static void subroutine() {
        int d = 0;
        int a = 10 / d;
    }
    public static void main(String args[]) {
        Exc1.subroutine();
    }
}
```

The resulting stack trace from the default exception handler shows how the entire call stack is displayed:

```
java.lang.ArithmeticException: / by zero
    at Exc1.subroutine(Exc1.java:4)
    at Exc1.main(Exc1.java:7)
```

As you can see, the bottom of the stack is **main**'s line 7, which is the call to **subroutine()**, which caused the exception at line 4. The call stack is quite useful for debugging, because it pinpoints the precise sequence of steps that led to the error.

## Using try and catch

Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself. Doing so provides two benefits. First, it allows you to fix the error. Second, it prevents the program from automatically terminating. Most users would be confused (to say the least) if your program stopped running and printed a stack trace whenever an error occurred! Fortunately, it is quite easy to prevent this.

To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a **try** block. Immediately following the **try** block, include a **catch** clause that specifies the exception type that you wish to catch. To illustrate how easily this can be done, the following program includes a **try** block and a **catch** clause that processes the **ArithmeticException** generated by the division-by-zero error:

```
class Exc2 {
    public static void main(String args[]) {
        int d, a;

        try { // monitor a block of code.
            d = 0;
            a = 42 / d;
            System.out.println("This will not be printed.");
        } catch (ArithmeticException e) { // catch divide-by-zero error
            System.out.println("Division by zero.");
        }
    }
}
```

```

        System.out.println("After catch statement.");
    }
}

```

This program generates the following output:

```

Division by zero.
After catch statement.

```

Notice that the call to `println()` inside the **try** block is never executed. Once an exception is thrown, program control transfers out of the **try** block into the **catch** block. Put differently, **catch** is not “called,” so execution never “returns” to the **try** block from a **catch**. Thus, the line “This will not be printed.” is not displayed. Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try** / **catch** mechanism.

A **try** and its **catch** statement form a unit. The scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement. A **catch** statement cannot catch an exception thrown by another **try** statement (except in the case of nested **try** statements, described shortly). The statements that are protected by **try** must be surrounded by curly braces. (That is, they must be within a block.) You cannot use **try** on a single statement.

The goal of most well-constructed **catch** clauses should be to resolve the exceptional condition and then continue on as if the error had never happened. For example, in the next program each iteration of the **for** loop obtains two random integers. Those two integers are divided by each other, and the result is used to divide the value 12345. The final result is put into **a**. If either division operation causes a divide-by-zero error, it is caught, the value of **a** is set to zero, and the program continues.

```

// Handle an exception and move on.
import java.util.Random;

class HandleError {
    public static void main(String args[]) {
        int a=0, b=0, c=0;
        Random r = new Random();

        for(int i=0; i<32000; i++) {
            try {
                b = r.nextInt();
                c = r.nextInt();
                a = 12345 / (b/c);
            } catch (ArithmeticException e) {
                System.out.println("Division by zero.");
                a = 0; // set a to zero and continue
            }
            System.out.println("a: " + a);
        }
    }
}

```

## Displaying a Description of an Exception

**Throwable** overrides the **toString()** method (defined by **Object**) so that it returns a string containing a description of the exception. You can display this description in a **println()** statement by simply passing the exception as an argument. For example, the **catch** block in the preceding program can be rewritten like this:

```
catch (ArithmeticException e) {
    System.out.println("Exception: " + e);
    a = 0; // set a to zero and continue
}
```

When this version is substituted in the program, and the program is run, each divide-by-zero error displays the following message:

```
Exception: java.lang.ArithmeticException: / by zero
```

While it is of no particular value in this context, the ability to display a description of an exception is valuable in other circumstances—particularly when you are experimenting with exceptions or when you are debugging.

## Multiple catch Clauses

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more **catch** clauses, each catching a different type of exception. When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed. After one **catch** statement executes, the others are bypassed, and execution continues after the **try / catch** block. The following example traps two different exception types:

```
// Demonstrate multiple catch statements.
class MultipleCatches {
    public static void main(String args[]) {
        try {
            int a = args.length;
            System.out.println("a = " + a);
            int b = 42 / a;
            int c[] = { 1 };
            c[42] = 99;
        } catch (ArithmeticException e) {
            System.out.println("Divide by 0: " + e);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index oob: " + e);
        }
        System.out.println("After try/catch blocks.");
    }
}
```

This program will cause a division-by-zero exception if it is started with no command-line arguments, since **a** will equal zero. It will survive the division if you provide a command-

line argument, setting **a** to something larger than zero. But it will cause an **ArrayIndexOutOfBoundsException**, since the **int** array **c** has a length of 1, yet the program attempts to assign a value to **c[42]**.

Here is the output generated by running it both ways:

```
C:\>java MultipleCatches
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.

C:\>java MultipleCatches TestArg
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException:42
After try/catch blocks.
```

When you use multiple **catch** statements, it is important to remember that exception subclasses must come before any of their superclasses. This is because a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass. Further, in Java, unreachable code is an error. For example, consider the following program:

```
/* This program contains an error.

A subclass must come before its superclass in
a series of catch statements. If not,
unreachable code will be created and a
compile-time error will result.
*/
class SuperSubCatch {
    public static void main(String args[]) {
        try {
            int a = 0;
            int b = 42 / a;
        } catch (Exception e) {
            System.out.println("Generic Exception catch.");
        }
        /* This catch is never reached because
           ArithmeticException is a subclass of Exception. */
        catch (ArithmeticException e) { // ERROR - unreachable
            System.out.println("This is never reached.");
        }
    }
}
```

If you try to compile this program, you will receive an error message stating that the second **catch** statement is unreachable because the exception has already been caught. Since **ArithmeticException** is a subclass of **Exception**, the first **catch** statement will handle all **Exception**-based errors, including **ArithmeticException**. This means that the second **catch** statement will never execute. To fix the problem, reverse the order of the **catch** statements.

## Nested try Statements

The **try** statement can be nested. That is, a **try** statement can be inside the block of another **try**. Each time a **try** statement is entered, the context of that exception is pushed on the stack. If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match. This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted. If no **catch** statement matches, then the Java run-time system will handle the exception. Here is an example that uses nested **try** statements:

```
// An example of nested try statements.
class NestTry {
    public static void main(String args[]) {
        try {
            int a = args.length;

            /* If no command-line args are present,
             the following statement will generate
             a divide-by-zero exception. */
            int b = 42 / a;

            System.out.println("a = " + a);

            try { // nested try block
                /* If one command-line arg is used,
                 then a divide-by-zero exception
                 will be generated by the following code. */
                if(a==1) a = a/(a-a); // division by zero

                /* If two command-line args are used,
                 then generate an out-of-bounds exception. */
                if(a==2) {
                    int c[] = { 1 };
                    c[42] = 99; // generate an out-of-bounds exception
                }
            } catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("Array index out-of-bounds: " + e);
            }

            } catch (ArithmeticException e) {
                System.out.println("Divide by 0: " + e);
            }
        }
    }
}
```

As you can see, this program nests one **try** block within another. The program works as follows. When you execute the program with no command-line arguments, a divide-by-zero exception is generated by the outer **try** block. Execution of the program with one command-line argument generates a divide-by-zero exception from within the nested **try** block. Since the inner block does not catch this exception, it is passed on to the outer **try** block, where it is handled. If you execute the program with two command-line arguments,

an array boundary exception is generated from within the inner **try** block. Here are sample runs that illustrate each case:

```
C:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One Two
a = 2
Array index out-of-bounds:
java.lang.ArrayIndexOutOfBoundsException:42
```

Nesting of **try** statements can occur in less obvious ways when method calls are involved. For example, you can enclose a call to a method within a **try** block. Inside that method is another **try** statement. In this case, the **try** within the method is still nested inside the outer **try** block, which calls the method. Here is the previous program recoded so that the nested **try** block is moved inside the method **nesttry()**:

```
/* Try statements can be implicitly nested via
calls to methods. */
class MethNestTry {
    static void nesttry(int a) {
        try { // nested try block
            /* If one command-line arg is used,
            then a divide-by-zero exception
            will be generated by the following code. */
            if(a==1) a = a/(a-a); // division by zero

            /* If two command-line args are used,
            then generate an out-of-bounds exception. */
            if(a==2) {
                int c[] = { 1 };
                c[42] = 99; // generate an out-of-bounds exception
            }
        } catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index out-of-bounds: " + e);
        }
    }
}

public static void main(String args[]) {
    try {
        int a = args.length;

        /* If no command-line args are present,
        the following statement will generate
        a divide-by-zero exception. */
        int b = 42 / a;
        System.out.println("a = " + a);
    }
}
```

```

        nesttry(a);
    } catch(ArithmeticException e) {
        System.out.println("Divide by 0: " + e);
    }
}
}

```

The output of this program is identical to that of the preceding example.

## throw

So far, you have only been catching exceptions that are thrown by the Java run-time system. However, it is possible for your program to throw an exception explicitly, using the **throw** statement. The general form of **throw** is shown here:

```
throw ThrowableInstance;
```

Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**. Primitive types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions. There are two ways you can obtain a **Throwable** object: using a parameter in a **catch** clause or creating one with the **new** operator.

The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed. The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing **try** statement is inspected, and so on. If no matching **catch** is found, then the default exception handler halts the program and prints the stack trace.

Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

```

// Demonstrate throw.
class ThrowDemo {
    static void demoproc() {
        try {
            throw new NullPointerException("demo");
        } catch(NullPointerException e) {
            System.out.println("Caught inside demoproc.");
            throw e; // rethrow the exception
        }
    }

    public static void main(String args[]) {
        try {
            demoproc();
        } catch(NullPointerException e) {
            System.out.println("Recaught: " + e);
        }
    }
}

```

This program gets two chances to deal with the same error. First, **main( )** sets up an exception context and then calls **demoproc( )**. The **demoproc( )** method then sets up

another exception-handling context and immediately throws a new instance of **NullPointerException**, which is caught on the next line. The exception is then rethrown. Here is the resulting output:

```
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

The program also illustrates how to create one of Java's standard exception objects. Pay close attention to this line:

```
throw new NullPointerException("demo");
```

Here, **new** is used to construct an instance of **NullPointerException**. Many of Java's built-in run-time exceptions have at least two constructors: one with no parameter and one that takes a string parameter. When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to **print()** or **println()**. It can also be obtained by a call to **getMessage()**, which is defined by **Throwable**.

## throws

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a **throws** clause in the method's declaration. A **throws** clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses. All other exceptions that a method can throw must be declared in the **throws** clause. If they are not, a compile-time error will result.

This is the general form of a method declaration that includes a **throws** clause:

```
type method-name(parameter-list) throws exception-list
{
    // body of method
}
```

Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.

Following is an example of an incorrect program that tries to throw an exception that it does not catch. Because the program does not specify a **throws** clause to declare this fact, the program will not compile.

```
// This program contains an error and will not compile.
class ThrowsDemo {
    static void throwOne() {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[]) {
        throwOne();
    }
}
```



To make this example compile, you need to make two changes. First, you need to declare that `throwOne()` throws `IllegalAccessException`. Second, `main()` must define a `try / catch` statement that catches this exception.

The corrected example is shown here:

```
// This is now correct.
class ThrowsDemo {
    static void throwOne() throws IllegalAccessException {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[]) {
        try {
            throwOne();
        } catch (IllegalAccessException e) {
            System.out.println("Caught " + e);
        }
    }
}
```

Here is the output generated by running this example program:

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

## finally

When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The **finally** keyword is designed to address this contingency.

**finally** creates a block of code that will be executed after a `try / catch` block has completed and before the code following the `try / catch` block. The **finally** block will execute whether or not an exception is thrown. If an exception is thrown, the **finally** block will execute even if no `catch` 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
ArithmeticException	Arithmetic 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.
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.
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.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

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

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

**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.

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 <b>try</b> -with-resources statement.
Throwable fillInStackTrace( )	Returns a <b>Throwable</b> 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, <b>null</b> 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 <b>StackTraceElement</b> . 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 <b>StackTraceElement</b> 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 <b>try</b> -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 <b>String</b> object containing a description of the exception. This method is called by <b>println( )</b> when outputting a <b>Throwable</b> object.

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

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

Beginning with JDK 1.4, 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. Thus, 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.ArithmeticException: cause

```

In this example, the top-level exception is **NullPointerException**. To it is added a cause exception, **ArithmeticException**. 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 Recently Added Exception Features

Beginning with JDK 7, three interesting and useful features have been added to 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 **ArithmeticException** and **ArrayIndexOutOfBoundsException**:

```
catch(ArithmeticException | 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 ArithmeticException

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

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

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

The program will generate an **ArithmeticException** 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. Unlike some other languages in which error return codes are used to indicate failure, Java uses exceptions. 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 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, multi-core 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 multi-core 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** Recently, the Fork/Join Framework was added to Java. It provides a powerful means of creating multithreaded applications that automatically scale to make best use of multi-core 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 28. 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 is done by explicitly yielding, sleeping, or blocking on pending I/O. 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 operating systems such as Windows, 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
<code>getName</code>	Obtain a thread's name.
<code>getPriority</code>	Obtain a thread's priority.
<code>isAlive</code>	Determine if a thread is still running.
<code>join</code>	Wait for a thread to terminate.
<code>run</code>	Entry point for the thread.
<code>sleep</code>	Suspend a thread for a period of time.
<code>start</code>	Start a thread by calling its <code>run</code> 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()** executes 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);
        t.start(); // Start the thread
    }

    // 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[ ] ) {
            new NewThread(); // create a new 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. Next, **start()** is called, which starts the thread of execution beginning at the **run()** method. This causes the child thread's **for** loop to begin. After calling **start()**, **NewThread**'s constructor returns to **main()**. When the main thread resumes, it 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, often the main thread must be the last thread to finish running. In fact, for some older JVMs, if the main thread finishes before a child thread has completed, then the Java run-time system may “hang.” 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. It must also call **start()** to begin 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);
        start(); // Start the thread
    }

    // 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[]) {
        new NewThread(); // create a new 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 modified 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);
        t.start(); // Start the thread
    }

    // 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[]) {
        new NewThread("One"); // start threads
        new NewThread("Two");
        new NewThread("Three");

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

    // 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 ob1 = new NewThread("One");
        NewThread ob2 = new NewThread("Two");
        NewThread ob3 = new NewThread("Three");
```

```

System.out.println("Thread One is alive: "
    + ob1.t.isAlive());
System.out.println("Thread Two is alive: "
    + ob2.t.isAlive());
System.out.println("Thread Three is alive: "
    + ob3.t.isAlive());
// wait for threads to finish
try {
    System.out.println("Waiting for threads to finish.");
    ob1.t.join();
    ob2.t.join();
    ob3.t.join();
} catch (InterruptedException e) {
    System.out.println("Main thread Interrupted");
}

System.out.println("Thread One is alive: "
    + ob1.t.isAlive());
System.out.println("Thread Two is alive: "
    + ob2.t.isAlive());
System.out.println("Thread Three is alive: "
    + ob3.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 thread is started immediately. 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);
            t.start();
        }
        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");

            // 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) {
        ...
    }
}
```

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);
        t.start();
    }

    // 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");

        // 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, Oracle 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;

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

    public void run() {
        int i = 0;

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

class Consumer implements Runnable {
    Q q;

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

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

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

```

        Q q = new Q();
        new Producer(q);
        new Consumer(q);

        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;

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

    public void run() {
        int i = 0;

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

class Consumer implements Runnable {
    Q q;

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

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

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

        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 starts a second thread to set 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 A.last");
    }
}

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

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

```

When you run this program, you will see the output shown here:

```
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 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;
        t.start(); // Start the thread
    }

    // 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");

        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 isn't as "clean" 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.

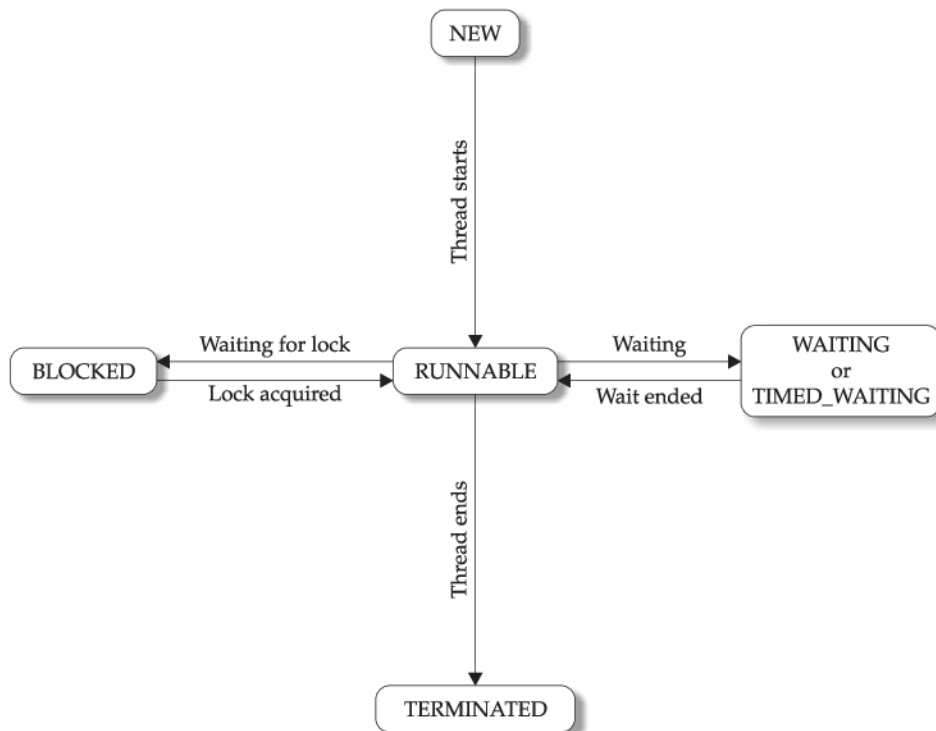


Figure 11-1 Thread states

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 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 multi-core system, consider using the new Fork/Join Framework, which is described in Chapter 28.

This page has been intentionally left blank

## CHAPTER

# 12

## Enumerations, Autoboxing, and Annotations (Metadata)

This chapter examines three relatively recent additions to the Java language: enumerations, autoboxing, and annotations (also referred to as metadata). Each expands the power of the language by offering a streamlined approach to handling common programming tasks. This chapter also discusses Java's type wrappers and introduces reflection.

### Enumerations

Versions of Java prior to JDK 5 lacked one feature that many programmers felt was needed: enumerations. In its simplest form, an *enumeration* is a list of named constants. Although Java offered other features that provide somewhat similar functionality, such as **final** variables, many programmers still missed the conceptual purity of enumerations—especially because enumerations are supported by many other commonly used languages. Beginning with JDK 5, enumerations were added to the Java language, and they are now an integral and widely used part of Java.

In their simplest form, Java enumerations appear similar to enumerations in other languages. However, 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. Therefore, although enumerations were several years in the making, Java's rich implementation made them well worth the wait.

### 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:

```
Value of ap: RedDel

ap contains GoldenDel.

Golden Delicious is yellow.
```