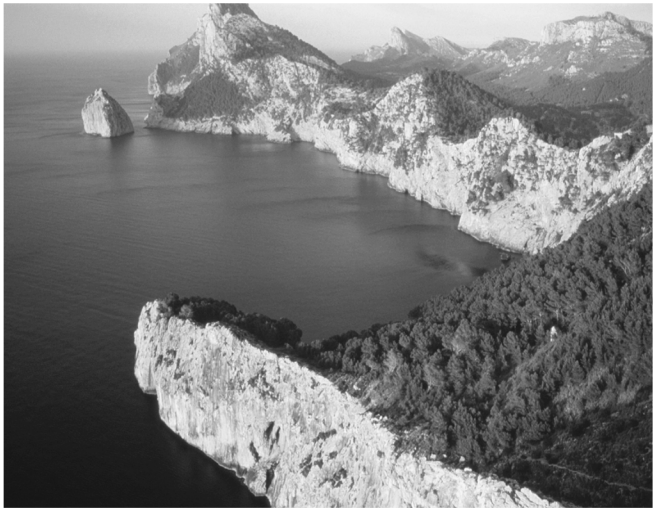
Complete Java 2 Certification

Study Guide

4th Edition



PART I

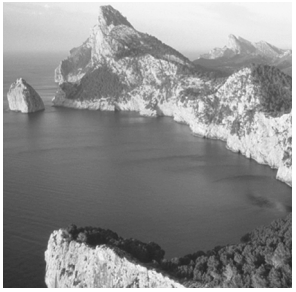
The Programmer's Exam

1

Language Fundamentals

JAVA CERTIFICATION EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- ✓ **1.1 Write code that declares, constructs, and initializes arrays of any base type using any of the permitted forms both for declaration and for initialization.**
- ✓ **1.3 For a given class, determine if a default constructor will be created and if so, state the prototype of that constructor.**
- ✓ **3.1 State the behavior that is guaranteed by the garbage collection system.**
- ✓ **3.2 Write code that explicitly makes objects eligible for garbage collection.**
- ✓ **3.3 Recognize the point in a piece of source code at which an object becomes eligible for garbage collection.**
- ✓ **4.1 Identify correctly constructed source files, package declarations, import statements, class declarations (of all forms including inner classes), interface declarations and implementations (for `java.lang.Runnable` or other interfaces described in the test), method declarations (including the main method that is used to start execution of a class), variable declarations, and identifiers.**
- ✓ **4.3 State the correspondence between index values in the argument array passed to a main method and command-line arguments.**
- ✓ **4.4 Identify all Java programming language keywords and correctly constructed identifiers.**
- ✓ **4.5 State the effect of using a variable or array element of any kind when no explicit assignment has been made to it.**
- ✓ **4.6 State the range of all primitive formats, data types, and declare literal values for String and all primitive types using all permitted formats bases and representations.**
- ✓ **5.4 Determine the effect upon objects and primitive values of passing variables into methods and performing assignments or other modifying operations in that method.**



This book is not an introduction to Java. Because you are preparing for certification, you are obviously already familiar with the fundamentals. The purpose of this chapter is to review those fundamentals covered by the Certification Exam objectives.

Source Files

All Java source files must end with the `.java` extension. A source file should generally contain, at most, one top-level public class definition; if a public class is present, the class name should match the unextended filename. For example, if a source file contains a public class called `RayTraceApplet`, then the file must be called `RayTraceApplet.java`. A source file may contain an unlimited number of non-public class definitions.

This is not actually a language requirement, but it is an implementation requirement of many compilers, including the reference compilers from Sun. It is therefore unwise to ignore this convention, because doing so limits the portability of your source files (but not, of course, your compiled files).

Three top-level elements known as *compilation units* may appear in a file. None of these elements is required. If they are present, then they must appear in the following order:

1. Package declaration
2. Import statements
3. Class definitions

The format of the package declaration is quite simple. The keyword `package` occurs first and is followed by the package name. The package name is a series of elements separated by periods. When class files are created, they must be placed in a directory hierarchy that reflects their package names. You must be careful that each component of your package name hierarchy is a legitimate directory name on all platforms. Therefore, you must not use characters such as the space, forward slash, backslash, or other symbols. Use only alphanumeric characters in package names.

Import statements have a similar form, but you may import either an individual class from a package or the entire package. To import an individual class, simply place the fully qualified class name after the `import` keyword and finish the statement with a semicolon (;); to import an entire package, simply add an asterisk (*) to the end of the package name.

White space and comments may appear before or after any of these elements. For example, a file called `Test.java` might look like this:

```
1. // Package declaration
2. package exam.prepguide;
3.
4. // Imports
5. import java.awt.Button; // imports a specific class
6. import java.util.*;     // imports an entire package
7.
8. // Class definition
9. public class Test {...}
```

Sometimes you might use classes with the same name in two different packages, such as the `Date` classes in the packages `java.util` and `java.sql`. If you use the asterisk form of import to import both entire packages and then attempt to use a class simply called `Date`, you will get a compiler error reporting that this usage is ambiguous. You must either make an additional import, naming one or the other `Date` class explicitly, or you must refer to the class using its fully qualified name.

Keywords and Identifiers

The Java language specifies 52 keywords and other reserved words, which are listed in Table 1.1.

TABLE 1.1 Java Keywords and Reserved Words

abstract	class	false	import	package	super	try
assert	const	final	instanceof	private	switch	void
boolean	continue	finally	int	protected	synchronized	volatile
break	default	float	interface	public	this	while
byte	do	for	long	return	throw	
case	double	goto	native	short	throws	
catch	else	if	new	static	transient	
char	extends	implements	null	strictfp	true	

The words `goto` and `const` are reserved words. However, they have no meaning in Java and programmers may not use them as identifiers.

An *identifier* is a word used by a programmer to name a variable, method, class, or label. Keywords and reserved words may not be used as identifiers. An identifier must begin with a letter, a dollar sign ($), or an underscore (_); subsequent characters may be letters, dollar signs, underscores, or digits. Some examples are

```
1. foobar                // legal
2. BIGinterface          // legal: embedded keywords
3.                       // are OK.
4. $incomeAfterExpenses  // legal
5. 3_node5               // illegal: starts with a digit
6. !theCase              // illegal: must start with
7.                       // letter, $, or _
```

Identifiers are case sensitive—for example, `radius` and `Radius` are distinct identifiers.

The exam is careful to avoid potentially ambiguous questions that require you to make purely academic distinctions between reserved words and keywords.

Primitive Data Types

Java's primitive data types are

- `boolean`
- `char`
- `byte`
- `short`
- `int`
- `long`
- `float`
- `double`

The apparent bit patterns of these types are defined in the Java language specification, and their effective sizes are listed in Table 1.2.

TABLE 1.2 Primitive Data Types and Their Effective Sizes

Type	Effective Representation Size (bits)	Type	Effective Representation Size (bits)
`boolean`	1	`char`	16
`byte`	8	`short`	16
`int`	32	`long`	64
`float`	32	`double`	64

Variables of type `boolean` may take only the values `true` or `false`.

The actual storage size and memory layout for these data items are not, in fact, required by the language specification. The specification does dictate the *apparent* behavior; so, for example, the effect of bit mask operations, shifts, and so on are entirely predictable at the Java level. If you write native code, you might find things are different from these tables. This means that you cannot reliably calculate the amount of memory consumed by adding up data sizes.

The exam is careful to avoid potentially ambiguous questions and asks about variables only from the Java language perspective, not the underlying implementation.

The four signed integral data types are

- `byte`
- `short`
- `int`
- `long`

Variables of these types are two's-complement numbers; their ranges are given in Table 1.3. Notice that for each type, the exponent of 2 in the minimum and maximum is one less than the size of the type.

Two's-complement is a way of representing signed integers that was originally developed for microprocessors in such a way as to have a single binary representation for the number 0. The most significant bit is used as the sign bit, where 0 is positive and 1 is negative.

TABLE 1.3 Ranges of the Integral Primitive Types

Type	Size	Minimum	Maximum
`byte`	8 bits	-2^7	$2^7 - 1$
`short`	16 bits	-2^{15}	$2^{15} - 1$
`int`	32 bits	-2^{31}	$2^{31} - 1$
`long`	64 bits	-2^{63}	$2^{63} - 1$

The `char` type is integral but unsigned. The range of a variable of type `char` is from 0 through $2^{16} - 1$. Java characters are in Unicode, which is a 16-bit encoding capable of representing a wide range of international characters. If the most significant 9 bits of a `char` are all 0, then the encoding is the same as 7-bit ASCII.

The two floating-point types are

- `float`
- `double`

The ranges of the floating-point primitive types are given in Table 1.4.

TABLE 1.4 Ranges of the Floating-Point Primitive Types

Type	Size	Minimum	Maximum
`float`	32 bits	$+/-1.40239846^{-45}$	$+/-3.40282347^{+38}$
`double`	16 bits	$+/-4.94065645841246544^{-324}$	$+/-1.79769313486231570^{+308}$

These types conform to the IEEE 754 specification. Many mathematical operations can yield results that have no expression in numbers (infinity, for example). To describe such non-numeric situations, both `double`s and `float`s can take on values that are bit patterns that do not represent numbers. Rather, these patterns represent non-numeric values. The patterns are defined in the `Float` and `Double` classes and may be referenced as follows (NaN stands for Not a Number):

- `Float.NaN`
- `Float.NEGATIVE_INFINITY`
- `Float.POSITIVE_INFINITY`
- `Double.NaN`
- `Double.NEGATIVE_INFINITY`
- `Double.POSITIVE_INFINITY`

The following code fragment shows the use of these constants:

```
1. double d = -10.0 / 0.0;
2. if (d == Double.NEGATIVE_INFINITY) {
3.   System.out.println("d just exploded: " + d);
4. }
```

In this code fragment, the test on line 2 passes, so line 3 is executed.

All numeric primitive types are signed.

Literals

A *literal* is a value specified in the program source, as opposed to one determined at runtime. Literals can represent primitive or string variables, and may appear on the right side of assignments or in method calls. You cannot assign values into literals, so they cannot appear on the left side of assignments.

boolean Literals

The only valid literals of `boolean` type are `true` and `false`. For example:

```
1. boolean isBig = true;
2. boolean isLittle = false;
```

char Literals

A `char` literal can be expressed by enclosing the desired character in single quotes, as shown here:

```
char c = 'w';
```

Of course, this technique only works if the desired character is available on the keyboard at hand. Another way to express a character literal is as a Unicode value specified using four hexadecimal digits, preceded by `\u`, with the entire expression in single quotes. For example:

```
char c1 = '\u4567';
```

Java supports a few escape sequences for denoting special characters:

- `'\n'` for new line
- `'\r'` for return

- `'\t'` for tab
- `'\b'` for backspace
- `'\f'` for formfeed
- `'\''` for single quote
- `'\" '` for double quote
- `'\\'` for backslash

Integral Literals

Integral literals may be expressed in decimal, octal, or hexadecimal. The default is decimal. To indicate octal, prefix the literal with `0` (zero). To indicate hexadecimal, prefix the literal with `0x` or `0X`; the hex digits may be upper- or lowercase. The value 28 may thus be expressed six ways:

- `28`
- `034`
- `0x1c`
- `0x1C`
- `0X1c`
- `0X1C`

By default, an integral literal is a 32-bit value. To indicate a `long` (64-bit) literal, append the suffix `L` to the literal expression. (The suffix can be lowercase, but then it looks so much like a one that your readers are bound to be confused.)

Floating-Point Literals

A *floating-point* literal expresses a floating-point number. In order to be interpreted as a floating-point literal, a numerical expression must contain one of the following:

- A decimal point: `1.414`
- The letter `E` or `e`, indicating scientific notation: `4.23E+21`
- The suffix `F` or `f`, indicating a `float` literal: `1.828f`
- The suffix `D` or `d`, indicating a `double` literal: `1234d`

A floating-point literal with no `F` or `D` suffix defaults to `double` type.

String Literals

A *string literal* is a sequence of characters enclosed in double quotes. For example:

```
String s = "Characters in strings are 16-bit Unicode.";
```

Java provides many advanced facilities for specifying non-literal string values, including a concatenation operator and some sophisticated constructors for the `String` class. These facilities are discussed in detail in Chapter 8, "The `java.lang` and `java.util` Packages."

Arrays

A Java *array* is an ordered collection of primitives, object references, or other arrays. Java arrays are homogeneous: except as allowed by polymorphism, all elements of an array must be of the same type. That is, when you create an array, you specify the element type, and the resulting array can contain only elements that are instances of that class or subclasses of that class.

To create and use an array, you must follow three steps:

1. Declaration
2. Construction
3. Initialization

Declaration tells the compiler the array's name and what type its elements will be. For example:

```
1. int[] ints;
2. Dimension[] dims;
3. float[][] twoDee;
```

Line 1 declares an array of a primitive type. Line 2 declares an array of object references (`Dimension` is a class in the `java.awt` package). Line 3 declares a two-dimensional array—that is, an array of arrays of `float`s.

The square brackets can come before or after the array variable name. This is also true, and perhaps most useful, in method declarations. A method that takes an array of `double`s could be declared as `myMethod(double dubs[])` or as `myMethod(double[] dubs)`; a method that returns an array of doubles may be declared as either `double[] anotherMethod()` or as `double anotherMethod()[]`. In this last case, the first form is probably more readable.

Generally, placing the square brackets adjacent to the type, rather than following the variable or method, allows the type declaration part to be read as a single unit: "int array" or "float array", which might make more sense. However, C/C++ programmers will be more familiar with the form where the brackets are placed to the right of the variable or method declaration. Given the number of magazine articles that have been dedicated to ways to correctly interpret complex C/C++ declarations (perhaps you recall the "spiral rule"), it's probably not a bad thing that Java has modified the syntax for these declarations. Either way, you need to recognize both forms.

Notice that the declaration does not specify the size of an array. Size is specified at runtime, when the array is allocated via the `new` keyword. For example:

```
1. int[] ints;           // Declaration to the compiler
2. ints = new int[25];   // Runtime construction
```

Since array size is not used until runtime, it is legal to specify size with a variable rather than a literal:

```
1. int size = 1152 * 900;
2. int[] raster;
3. raster = new int[size];
```

Declaration and construction may be performed in a single line:

```
1. int[] ints = new int[25];
```

When an array is constructed, its elements are automatically initialized to their default values. These defaults are the same as for object member variables. Numerical elements are initialized to 0; non-numeric elements are initialized to 0-like values, as shown in Table 1.5.

TABLE 1.5 Array Element Initialization Values

Element Type	Initial Value	Element Type	Initial Value
`byte`	0	`short`	0
`int`	0	`long`	0L
`float`	0.0f	`double`	0.0d
`char`	'\u0000'	`boolean`	false
`object reference`	`null`		

Arrays are actually objects, even to the extent that you can execute methods on them (mostly the methods of the `Object` class), although you cannot subclass the array class. Therefore, this initialization is exactly the same as for other objects, and as a consequence you will see this table again in the next section.

If you want to initialize an array to values other than those shown in Table 1.5, you can combine declaration, construction, and initialization into a single step. The following line of code creates a custom-initialized array of five floats:

```
1. float[] diameters = {1.1f, 2.2f, 3.3f, 4.4f, 5.5f};
```

The array size is inferred from the number of elements within the curly braces.

Of course, an array can also be initialized by explicitly assigning a value to each element, starting at array index 0:

```
1. long[] squares;
2. squares = new long[6000];
3. for (int i = 0; i < 6000; i++) {
4.   squares[i] = i * i;
5. }
```

When the array is created at line 2, it is full of default values (`0L`) which are replaced in lines 3–4. The code in the example works but can be improved. If you later need to change the array size (in line 2), the loop counter will have to change (in line 3), and the program could be damaged if line 3 is not taken care of. The safest way to refer to the size of an array is to append the `.length` member variable to the array name. Thus, our example becomes:

```
1. long[] squares;
2. squares = new long[6000];
3. for (int i = 0; i < squares.length; i++) {
4.   squares[i] = i * i;
5. }
```

Java allows you to create non-rectangular arrays. Because multi-dimensional arrays are simply arrays of arrays, each subarray is a separate object, and there is no requirement that the dimension of each subarray be the same. Of course, this type of array requires more care in handling because you cannot simply iterate each subarray using the same limits.

Class Fundamentals

Java is all about classes, and a review of the Certification Exam objectives will show that you need to be intimately familiar with them. Classes are discussed in detail in Chapter 6, "Objects and Classes"; for now, let's examine a few fundamentals.

The *main()* Method

The `main()` method is the entry point for standalone Java applications. To create an application, you write a class definition that includes a `main()` method. To execute an application, type **java** at the command line, followed by the name of the class containing the `main()` method to be executed.

The signature for `main()` is

```
public static void main(String[] args)
```

The `main()` method is declared public by convention. However, it is a requirement that it be static so that it can be executed without the necessity of constructing an instance of the corresponding class.

The `args` array contains any arguments that the user might have entered on the command line. For example, consider the following command line:

```
% java Mapper France Belgium
```

With this command line, the `args[]` array has two elements: `France` in `args[0]`, and `Belgium` in `args[1]`. Note that neither the class name (`Mapper`) nor the command name (`java`) appears in the array. Of course, the name `args` is purely arbitrary: any legal identifier may be used, provided the array is a single-dimensional array of `String` objects.

Variables and Initialization

Java supports variables of three different lifetimes:

Member variable A *member variable* of a class is created when an instance is created, and it is destroyed when the object is destroyed. Subject to accessibility rules and the need for a reference to the object, member variables are accessible as long as the enclosing object exists.

Automatic variable An *automatic variable* of a method (also known as a *method local*) is created on entry to the method and exists only during execution of the method, and therefore is accessible only during the execution of that method. (You'll see an exception to this rule when you look at inner classes, but don't worry about that for now.)

Class variable A *class variable* (also known as a *static variable*) is created when the class is loaded and is destroyed when the class is unloaded. There is only one copy of a class variable, and it exists regardless of the number of instances of the class, even if the class is never instantiated.

All member variables that are not explicitly assigned a value upon declaration are automatically assigned an initial value. The initialization value for member variables depends on the member variable's type. Values are listed in Table 1.6.

TABLE 1.6 Initialization Values for Member Variables

Element Type	Initial Value	Element Type	Initial Value
byte	0	short	0
int	0	long	0L
float	0.0f	double	0.0d
char	'\u0000'	boolean	false
object reference	null		

The values in Table 1.6 are the same as those in Table 1.5; member variable initialization values are the same as array element initialization values.

A member value may be initialized in its own declaration line:

```
1. class HasVariables {
2.   int x = 20;
3.   static int y = 30;
```

When this technique is used, nonstatic instance variables are initialized just before the class constructor is executed; here `x` would be set to 20 just before invocation of any `HasVariables` constructor. Static variables are initialized at class load time; here `y` would be set to 30 when the `HasVariables` class is loaded.

Automatic variables (also known as *local variables*) are not initialized by the system; every automatic variable must be explicitly initialized before being used. For example, this method will not compile:

```
1. public int wrong() {
2.   int i;
3.   return i+5;
4. }
```

The compiler error at line 3 is, "Variable `i` may not have been initialized." This error often appears when initialization of an automatic variable occurs at a lower level of curly braces than the use of that variable. For example, the following method below returns the fourth root of a positive number:

```
1. public double fourthRoot(double d) {
2.   double result;
3.     if (d >= 0) {
4.       result = Math.sqrt(Math.sqrt(d));
5.     }
```

```
6.     return result;
7. }
```

Here the result is initialized on line 4, but the initialization takes place within the curly braces of lines 3 and 5. The compiler will flag line 6, complaining that "Variable `result` may not have been initialized." A common solution is to initialize `result` to some reasonable default as soon as it is declared:

```
1. public double fourthRoot(double d) {
2.   double result = 0.0;  // Initialize
3.     if (d >= 0) {
4.       result = Math.sqrt(Math.sqrt(d));
5.     }
6.     return result;
7. }
```

Now `result` is satisfactorily initialized. Line 2 demonstrates that an automatic variable may be initialized in its declaration line. Initialization on a separate line is also possible.

Class variables are initialized in the same manner as for member variables.

Argument Passing: By Reference or By Value

When Java passes an argument into a method call, a *copy* of the argument is actually passed. Consider the following code fragment:

```
1. double radians = 1.2345;
2. System.out.println("Sine of " + radians +
3.                    " = " + Math.sin(radians));
```

The variable `radians` contains a pattern of bits that represents the number 1.2345. On line 2, a copy of this bit pattern is passed into the method-calling apparatus of the Java Virtual Machine (JVM).

When an argument is passed into a method, changes to the argument value by the method do not affect the original data. Consider the following method:

```
1. public void bumper(int bumpMe) {
2.   bumpMe += 15;
3. }
```

Line 2 modifies a copy of the parameter passed by the caller. For example:

```
1. int xx = 12345;
2. bumper(xx);
3. System.out.println("Now xx is " + xx);
```

On line 2, the caller's xx variable is copied; the copy is passed into the bumper() method and incremented by 15. Because the original xx is untouched, line 3 will report that xx is still 12345.

This is also true when the argument to be passed is an object rather than a primitive. However, it is crucial for you to understand that the effect is very different. In order to understand the process, you have to understand the concept of the *object reference*.

Java programs do not deal directly with objects. When an object is constructed, the constructor returns a value—a bit pattern—that uniquely identifies the object. This value is known as a *reference* to the object. For example, consider the following code:

```
1. Button btn;
2. btn = new Button("Ok");
```

In line 2, the Button constructor returns a reference to the just-constructed button—not the actual button object or a copy of the button object. This reference is stored in the variable btn. In some implementations of the JVM, a reference is simply the address of the object; however, the JVM specification gives wide latitude as to how references can be implemented. You can think of a reference as simply a pattern of bits that uniquely identifies an individual object.

In most JVMs, the reference value is actually the address of an address. This second address refers to the real data. This approach, called *double indirection*, allows the garbage collector to relocate objects to reduce memory fragmentation.

When Java code appears to store objects in variables or pass objects into method calls, the object references are stored or passed.

Consider this code fragment:

```
1. Button btn;
2. btn = new Button("Pink");
3. replacer(btn);
4. System.out.println(btn.getLabel());
5.
6. public void replacer(Button replaceMe) {
7.   replaceMe = new Button("Blue");
8. }
```

Line 2 constructs a button and stores a reference to that button in btn. In line 3, a copy of the reference is passed into the replacer() method. Before execution of line 7, the value in replaceMe is a reference to the Pink button. Then line 7 constructs a second button and stores a reference to the second button in replaceMe, thus overwriting the reference to the Pink button. However, the caller's copy of the reference is not affected, so on line 4 the call to btn.getLabel() calls the original button; the string printed out is "Pink".

You have seen that called methods cannot affect the original value of their arguments—that is, the values stored by the caller. However, when the called method operates on an object via

the reference value that is passed to it, there are important consequences. If the method modifies the object via the reference, as distinguished from modifying the method argument—the reference—then the changes will be visible to the caller. For example:

```
1. Button btn;
2. btn = new Button("Pink");
3. changer(btn);
4. System.out.println(btn.getLabel());
5.
6. public void changer(Button changeMe) {
7.   changeMe.setLabel("Blue");
8. }
```

In this example, the variable `changeMe` is a copy of the reference `btn`, just as before. However, this time the code uses the copy of the reference to change the actual original object rather than trying to change the reference. Because the caller's object is changed rather than the callee's reference, the change is visible and the value printed out by line 4 is "Blue".

Arrays are objects, meaning that programs deal with references to arrays, not with arrays themselves. What gets passed into a method is a copy of a reference to an array. It is therefore possible for a called method to modify the contents of a caller's array.

How to Create a Reference to a Primitive

This is a useful technique if you need to create the effect of passing primitive values by reference. Simply pass an array of one primitive element over the method call, and the called method can now change the value seen by the caller. To do so, use code like this:

```
 1. public class PrimitiveReference {
 2.   public static void main(String args[]) {
 3.     int [] myValue = { 1 };
 4.     modifyIt(myValue);
 5.     System.out.println("myValue contains " +
 6.                        myValue[0]);
 7.   }
 8.   public static void modifyIt(int [] value) {
 9.     value[0]++;
10.   }
11. }
```

Garbage Collection

Most modern languages permit you to allocate data storage during a program run. In Java, this is done directly when you create an object with the `new` operation and indirectly when you call a method that has local variables or arguments. Method locals and arguments are allocated space on the stack and are discarded when the method exits, but objects are allocated space on the heap and have a longer lifetime.

Each process has its own stack and heap, and they are located on opposite sides of the process address space. The sizes of the stack and heap are limited by the amount of memory that is available on the host running the program. They may be further limited by the operating system or user-specific limits.

It is important to recognize that objects are always allocated on the heap. Even if they are created in a method using code like

```
public void aMethod() {
   MyClass mc = new MyClass();
}
```

the local variable `mc` is a reference, allocated on the stack, whereas the object to which that variable refers, an instance of `MyClass`, is allocated on the heap.

This section is concerned with recovery of space allocated on the heap. The increased lifetime raises the question of when storage allocation on the heap can be released. Some languages require that you, the programmer, explicitly release the storage when you have finished with it. This approach has proven seriously error-prone, because you might release the storage too soon (causing corrupted data if any other reference to the data is still in use) or forget to release it altogether (causing a memory shortage). Java's garbage collection solves the first of these problems and greatly simplifies the second.

In Java, you never explicitly free memory that you have allocated; instead, Java provides automatic garbage collection. The runtime system keeps track of the memory that is allocated and is able to determine whether that memory is still useable. This work is usually done in the background by a low-priority thread that is referred to as the *garbage collector*. When the garbage collector finds memory that is no longer accessible from any live thread (the object is out of scope), it takes steps to release it back into the heap for re-use. Specifically, the garbage collector calls the class destructor method called `finalize()` (if it is defined) and then frees the memory.

Garbage collection can be done in a number of different ways; each has advantages and disadvantages, depending on the type of program that is running. A real-time control system, for example, needs to know that nothing will prevent it from responding quickly to interrupts; this application requires a garbage collector that can work in small chunks or that can be interrupted easily. On the other hand, a memory-intensive program might work better with a garbage collector that stops the program from time to time but recovers memory more urgently as

a result. At present, garbage collection is hardwired into the Java runtime system; most garbage collection algorithms use an approach that gives a reasonable compromise between speed of memory recovery and responsiveness. In the future, you will probably be able to plug in different garbage-collection algorithms or buy different JVMs with appropriate collection algorithms, according to your particular needs.

This discussion leaves one crucial question unanswered: When is storage recovered? The best answer is that storage is not recovered unless it is definitely no longer in use. That's it. Even though you are not using an object any longer, you cannot say if it will be collected in 1 millisecond, in 100 milliseconds, or even if it will be collected at all. The methods `System.gc()` and `Runtime.gc()` look as if they run the garbage collector, but even these cannot be relied upon in general, because some other thread might prevent the garbage collection thread from running. In fact, the documentation for the `gc()` methods states:

> Calling this method suggests that the Java Virtual Machine expends effort toward recycling unused objects

How to Cause Leaks in a Garbage Collection System

The nature of automatic garbage collection has an important consequence: you can still get memory leaks. If you allow live, accessible references to unneeded objects to persist in your programs, then those objects cannot be garbage collected. Therefore, it may be a good idea to explicitly assign `null` into a variable when you have finished with it. This issue is particularly noticeable if you are implementing a collection of some kind.

In this example, assume the array `storage` is being used to maintain the storage of a stack. This `pop()` method is inappropriate:

```
1. public Object pop() {
2.   return storage[index--];
3. }
```

If the caller of this `pop()` method abandons the popped value, it will not be eligible for garbage collection until the array element containing a reference to it is overwritten. This might take a long time. You can speed up the process like this:

```
1. public Object pop() {
2.   Object returnValue = storage[index];
3.   storage[index--] = null;
4.   return returnValue;
5. }
```

Summary

This chapter has covered quite a bit of ground and a large variety of topics. You learned that a source file's elements must appear in this order:

1. Package declaration
2. Import statements
3. Class definitions

There should be, at most, one public class definition per source file; the filename must match the name of the public class.

You also learned that an identifier must begin with a letter, a dollar sign, or an underscore; subsequent characters may be letters, dollar signs, underscores, or digits. Java has four signed integral primitive data types: `byte`, `short`, `int`, and `long`; all four types display the behavior of two's-complement representation. Java's two floating-point primitive data types are `float` and `double`, the `char` type is unsigned and represents a Unicode character, and the `boolean` type may only take on the values `true` and `false`.

In addition, you learned that arrays must be (in order):

1. Declared
2. Constructed
3. Initialized

Default initialization is applied to member variables, class variables, and array elements, but not automatic variables. The default values are `0` for numeric types, the `null` value for object references, the `null` character for `char`, and `false` for `boolean`. The `length` member of an array gives the number of elements in the array. A class with a `main()` method can be invoked from the command line as a Java application. The signature for `main()` is `public static void main(String[] args)`. The `args[]` array contains all command-line arguments that appeared after the name of the application class.

You should also understand that method arguments are copies, not originals. For arguments of primitive data type, this means that modifications to an argument within a method are not visible to the caller of the method. For arguments of object type (including arrays), modifications to an argument value within a method are still not visible to the caller of the method; however, modifications in the object or array to which the argument refers *do* appear to the caller.

Finally, Java's garbage collection mechanism may only recover memory that is definitely unused. It is not possible to force garbage collection reliably. It is not possible to predict when a piece of unused memory will be collected, only to say when it becomes *eligible* for collection. Garbage collection does not prevent memory leaks; they can still occur if unused references are not cleared to `null` or destroyed.

Exam Essentials

Recognize and create correctly constructed source files. You should know the various kinds of compilation units and their required order of appearance.

Recognize and create correctly constructed declarations. You should be familiar with declarations of packages, classes, interfaces, methods, and variables.

Recognize Java keywords. You should recognize the keywords and reserved words listed in Table 1.1.

Distinguish between legal and illegal identifiers. You should know the rules that restrict the first character and the subsequent characters of an identifier.

Know all the primitive data types and the ranges of the integral data types. These are summarized in Tables 1.2 and 1.3.

Recognize correctly formatted literals. You should be familiar with all formats for literal characters, strings, and numbers.

Know how to declare and construct arrays. The declaration includes one empty pair of square brackets for each dimension of the array. The square brackets can appear before or after the array name. Arrays are constructed with the keyword `new`.

Know the default initialization values for all possible types of class variables and array elements. Know when data is initialized. Initialization takes place when a class or array is constructed. The initialization values are `0` for numeric type arrays, `false` for `boolean` arrays, and `null` for object reference type arrays.

Know the contents of the argument list of an application's `main()` method, given the command line that invoked the application. Be aware that the list is an array of `String`s containing everything on the command line except the `java` command, command-line options, and the name of the class.

Know that Java passes method arguments by value. Changes made to a method argument are not visible to the caller, because the method argument changes a copy of the argument. Objects are not passed to methods; only references to objects are passed.

Understand memory reclamation and the circumstances under which memory will be reclaimed. If an object is still accessible to any live thread, that object will certainly not be collected. This is true even if the program will never access the object again—the logic is simple and cannot make inferences about the semantics of the code. No guarantees are made about reclaiming available memory or the timing of reclamation if it does occur. A standard JVM has no entirely reliable, platform-independent way to force garbage collection. The `System` and `Runtime` classes each have a `gc()` method, and these methods make it more likely that garbage collection will run, but they provide no guarantee.

Key Terms

Before you take the exam, be certain you are familiar with the following terms:

array
automatic variable
class variable
compilation units
floating-point
garbage collector
identifier
literal
local variable
member variable
method local
object reference
reference
static variable
string literal

Review Questions

1. A signed data type has an equal number of non-zero positive and negative values available.
   - **A.** True
   - **B.** False

2. Choose the valid identifiers from those listed here. (Choose all that apply.)
   - **A.** `BigOlLongStringWithMeaninglessName`
   - **B.** `$int`
   - **C.** `bytes`
   - **D.** `$1`
   - **E.** `finalist`

3. Which of the following signatures are valid for the `main()` method entry point of an application? (Choose all that apply.)
   - **A.** `public static void main()`
   - **B.** `public static void main(String arg[])`
   - **C.** `public void main(String [] arg)`
   - **D.** `public static void main(String[] args)`
   - **E.** `public static int main(String [] arg)`

4. If all three top-level elements occur in a source file, they must appear in which order?
   - **A.** Imports, package declarations, classes
   - **B.** Classes, imports, package declarations
   - **C.** Package declarations must come first; order for imports and class definitions is not significant
   - **D.** Package declarations, imports, classes
   - **E.** Imports must come first; order for package declarations and class definitions is not significant

5. Consider the following line of code:

```
int[] x = new int[25];
```

   After execution, which statements are true? (Choose all that apply.)
   - **A.** `x[24]` is 0.
   - **B.** `x[24]` is undefined.
   - **C.** `x[25]` is 0.
   - **D.** `x[0]` is `null`.
   - **E.** `x.length` is 25.

6. Consider the following application:

```
1. class Q6 {
2.   public static void main(String args[]) {
3.     Holder h = new Holder();
4.     h.held = 100;
5.     h.bump(h);
6.     System.out.println(h.held);
7.   }
8. }
9.
10. class Holder {
11.   public int held;
12.   public void bump(Holder theHolder) {
13.     theHolder.held++; }
14.   }
15. }
```

What value is printed out at line 6?

A. 0

B. 1

C. 100

D. 101

7. Consider the following application:

```
1. class Q7 {
2.   public static void main(String args[]) {
3.     double d = 12.3;
4.     Decrementer dec = new Decrementer();
5.     dec.decrement(d);
6.     System.out.println(d);
7.   }
8. }
9.
10. class Decrementer {
11.   public void decrement(double decMe) {
12.     decMe = decMe - 1.0;
13.  }
14. }
```

What value is printed out at line 6?

A. 0.0

B. 1.0

C. 12.3

D. 11.3

8. How can you force garbage collection of an object?

A. Garbage collection cannot be forced.

B. Call `System.gc()`.

C. Call `System.gc()`, passing in a reference to the object to be garbage-collected.

D. Call `Runtime.gc()`.

E. Set all references to the object to new values (`null`, for example).

9. What is the range of values that can be assigned to a variable of type `short`?

A. Depends on the underlying hardware

B. 0 through $2^{16} - 1$

C. 0 through $2^{32} - 1$

D. -2^{15} through $2^{15} - 1$

E. -2^{31} through $2^{31} - 1$

10. What is the range of values that can be assigned to a variable of type `byte`?

A. Depends on the underlying hardware

B. 0 through $2^{8} - 1$

C. 0 through $2^{16} - 1$

D. -2^{7} through $2^{7} - 1$

E. -2^{15} through $2^{15} - 1$

Answers to Review Questions

1. B. The range of negative numbers is greater by one than the range of positive numbers.

2. A, B, C, D, E. All of the identifiers are valid.

3. B, D. All the choices are valid method signatures. However, in order to be the entry point of an application, a `main()` method must be public, static, and void; it must take a single argument of type `String[]`.

4. D. This order must be strictly observed.

5. A, E. The array has 25 elements, indexed from 0 through 24. All elements are initialized to 0.

6. D. A holder is constructed on line 3. A reference to that holder is passed into method `bump()` on line 5. Within the method call, the holder's `held` variable is bumped from 100 to 101.

7. C. The `decrement()` method is passed a copy of the argument `d`; the copy gets decremented, but the original is untouched.

8. A. Garbage collection cannot be forced. Calling `System.gc()` or `Runtime.gc()` is not 100 percent reliable, because the garbage-collection thread might defer to a thread of higher priority; thus B and D are incorrect. C is incorrect because the two `gc()` methods do not take arguments; in fact, if you still have a reference to pass into any method, the object is not yet eligible to be collected. E will make the object eligible for collection the next time the garbage collector runs.

9. D. The range for a 16-bit `short` is -2^{15} through $2^{15} - 1$. This range is part of the Java specification, regardless of the underlying hardware.

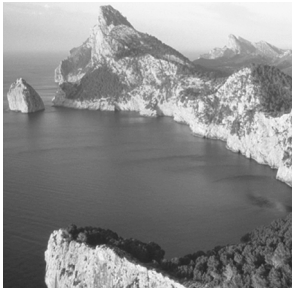
10. D. The range for an 8-bit byte is -2^7 through 2^7-1. Table 1.3 lists the ranges for Java's integral primitive data types.

Chapter 2

Operators and Assignments

JAVA CERTIFICATION EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- ✓ **5.1 Determine the result of applying any operator, including assignment operators, `instanceof`, and casts to operands of any type, class, scope, or accessibility, or any combination of these.**
- ✓ **5.2 Determine the result of applying the boolean `equals(object)` method to objects of any combination of the classes `java.lang.String`, `java.lang.Boolean`, and `java.lang.Object`.**
- ✓ **5.3 In an expression involving the operators &, |, &&, ||, and variables of known values, state which operands are evaluated and the value of the expression.**



Java provides a full set of operators, most of which are taken from C and C++. However, Java's operators differ in some important aspects from their counterparts in these other languages, and you need to understand clearly how Java's operators behave. This chapter describes all the operators. Some are described briefly, whereas operators that sometimes cause confusion are described in more detail. You will also learn about the behavior of expressions under conditions of arithmetic overflow.

Java's operators are shown in Table 2.1. They are listed in precedence order, with the highest precedence at the top of the table. Each group has been given a name for reference purposes; that name is shown in the left column of the table. Arithmetic and comparison operators are each split further into two subgroupings because they have different levels of precedence. We'll discuss these groupings later.

TABLE 2.1 Operators in Java, in Descending Order of Precedence

Category	Operators
Unary	`++ -- + - ! ~ (type)`
Arithmetic	`* / %`
	`+ -`
Shift	`<< >> >>>`
Comparison	`< <= > >= instanceof`
	`== !=`
Bitwise	`& ^ |`
Short-circuit	`&& ||`
Conditional	`?:`
Assignment	`= op=`

The rest of this chapter examines each of these operators. But before we start, let's consider the general issue of evaluation order.

Evaluation Order

In Java, the order of evaluation of operands in an expression is fixed. Consider this code fragment:

```
1. int [] a = { 4, 4 };
2. int b = 1;
3. a[b] = b = 0;
```

In this case, it might be unclear which element of the array is modified: which value of `b` is used to select the array element, 0 or 1? An evaluation from left to right requires that the leftmost expression, `a[b]`, be evaluated first, so it is a reference to the element `a[1]`. Next, `b` is evaluated, which is simply a reference to the variable called `b`. The constant expression `0` is evaluated next, which clearly does not involve any work. Now that the operands have been evaluated, the operations take place. This is done in the order specified by precedence and associativity. For assignments, associativity is right-to-left, so the value `0` is first assigned to the variable called `b`; then the value `0` is assigned into the last element of the array `a`.

The following sections examine each of these operators in turn.

Although Table 2.1 shows the precedence order, the degree of detail in this precedence ordering is rather high. It is generally better style to keep expressions simple and to use redundant bracketing to make it clear how any particular expression should be evaluated. This approach reduces the chance that less experienced programmers will have difficulty trying to read or maintain your code. Bear in mind that the code generated by the compiler will be the same despite redundant brackets.

The Unary Operators

The first group of operators in Table 2.1 consists of the *unary operators*. Most operators take two operands. When you multiply, for example, you work with two numbers. Unary operators, on the other hand, take only a single operand and work just on that. Java provides seven unary operators:

- The increment and decrement operators: `++` and `--`
- The unary plus and minus operators: `+` and `-`
- The bitwise inversion operator: `~`
- The `boolean` complement operator: `!`
- The cast: `()`

Strictly speaking, the cast is not an operator. However, we discuss it as if it were for simplicity, because it fits well with the rest of our discussion.

The Increment and Decrement Operators: *++* and *--*

These operators modify the value of an expression by adding or subtracting 1. So, for example, if an `int` variable `x` contains 10, then `++x` results in 11. Similarly, `--x`, again applied when `x` contains 10, gives a value of 9. In this case, the expression `--x` itself describes storage (the value of the variable `x`), so the resulting value is stored in `x`.

The preceding examples show the operators positioned before the expression (known as *pre-increment* or *pre-decrement*). They can, however, be placed after the expression instead (known as *post-increment* or *post-decrement*). To understand how the position of these operators affects their operation, you must appreciate the difference between the value stored by these operators and the result value they give. Both `x++` and `++x` cause the same result in `x`. However, the value of the expression itself is different. For example, if you say `y = x++;`, then the value assigned to `y` is the original value of `x`. If you say `y = ++x;`, then the value assigned to `y` is 1 more than the original value of `x`. In both cases, the value of `x` is incremented by 1.

Table 2.2 shows the values of `x` and `y` before and after particular assignments using these operators.

TABLE 2.2 Examples of Pre-and Post- Increment and Decrement Operations

Initial Value of *x*	Expression	Final Value of *y*	Final Value of *x*
5	`y = x++`	5	6
5	`y = ++x`	6	6
5	`y = x--`	5	4
5	`y = --x`	4	4

The Unary Plus and Minus Operators: *+* and *-*

The unary operators + and – are distinct from the more common binary + and – operators, which are usually just referred to as + and – (add and subtract). Both the programmer and the compiler are able to determine which meaning these symbols should have in a given context.

Unary + has no effect beyond emphasizing the positive nature of a numeric literal. Unary – negates an expression. So, you might make a block of assignments like this:

```
1. x = -3;
2. y = +3;
3. z = -(y + 6);
```

In such an example, the only reasons for using the unary + operator are to make it explicit that `y` is assigned a positive value and perhaps to keep the code aligned more pleasingly. At line 3, notice that these operators are not restricted to literal values but can be applied to expressions equally well, so the value of `z` is initialized to –9.

The Bitwise Inversion Operator: ~

The ~ operator performs *bitwise inversion* on integral types.

For each primitive type, Java uses a virtual machine representation that is platform independent. This means that the bit pattern used to represent a particular value in a particular variable type is always the same. This feature makes bit-manipulation operators even more useful, because they do not introduce platform dependencies. The ~ operator works by inverting all the 1 bits in a binary value to 0s and all the 0 bits to 1s.

For example, applying this operator to a byte containing 00001111 would result in the value 11110000. The same simple logic applies, no matter how many bits there are in the value being operated on. This operator is often used in conjunction with shift operators (<<, >>, and >>>) to perform bit manipulation, for example when driving I/O ports.

The *Boolean* Complement Operator: *!*

The ! operator inverts the value of a `boolean` expression. So `!true` gives false and `!false` gives true.

This operator is often used in the test part of an `if()` statement. The effect is to change the value of the `Boolean` expression. In this way, for example, the body of the `if()` and `else` parts can be swapped. Consider these two equivalent code fragments:

```
1. public Object myMethod(Object x) {
2.   if (x instanceof String) {
3.     // do nothing
4.   }
5.   else {
6.     x = x.toString();
7.   }
8.   return x;
9. }
```

and

```
1. public Object myMethod(Object x) {
2.   if (!(x instanceof String)) {
3.     x = x.toString();
4.   }
5.   return x;
6. }
```

In the first fragment, a test is made at line 2, but the conversion and assignment, at line 6, occurs only if the test failed. This is achieved by the somewhat cumbersome technique of using only the `else` part of an `if/else` construction. The second fragment uses the complement operator so that the overall test performed at line 2 is reversed; it may be read as, "If it is false that `x` is an instance of a string" or more likely, "If `x` is not a string." Because of this change to the test, the conversion can be performed at line 3 in the situation that the test has succeeded; no `else` part is required, and the resulting code is cleaner and shorter.

This is a simple example, but such usage is common. This level of understanding will leave you well-armed for the Certification Exam.

The Cast Operator: *(type)*

Casting is used for explicit conversion of the type of an expression. This is possible only for plausible target types. The compiler and the runtime system check for conformance with typing rules, which are described later.

Casting Primitives

Casts can be applied to change the type of primitive values—for example, forcing a `double` value into an `int` variable like this:

```
int circum = (int)(Math.PI * diameter);
```

If the cast, which is represented by the `(int)` part, were not present, the compiler would reject the assignment; a `double` value, such as is returned by the arithmetic here, cannot be represented accurately by an `int` variable. The cast is the programmer's way to say to the compiler, "I know you think this is risky, but trust me—I'm an engineer." Of course, if the result loses value or precision to the extent that the program does not work properly, then you are on your own.

Casting Object References

Casts can also be applied to object references. This often happens when you use containers, such as the `Vector` object. If you put, for example, `String` objects into a `Vector`, then when you

extract them, the return type of the `elementAt()` method is simply `Object`. To use the recovered value as a `String` reference, a cast is needed, like this:

```
1. Vector v = new Vector();
2. v.add ("Hello");
3. String s = (String)v.get(0);
```

The cast here occurs at line 3, in the form `(String)`. Although the compiler allows this cast, checks occur at runtime to determine if the object extracted from the `Vector` really is a `String`. Chapter 4, "Converting and Casting," covers casting, the rules governing which casts are legal and which are not, and the nature of the runtime checks that are performed on cast operations.

Now that we have considered the unary operators, which have the highest precedence, we will discuss the five arithmetic operators.

The Arithmetic Operators

Next highest in precedence, after the unary operators, are the *arithmetic operators*. This group includes, but is not limited to, the four most familiar operators, which perform addition, subtraction, multiplication, and division. Arithmetic operators are split into two further subgroupings, as shown in Table 2.1. In the first group are `*`, `/`, and `%`; in the second group, at a lower precedence, are `+` and `-`. The following sections discuss these operators and also what happens when arithmetic goes wrong.

The Multiplication and Division Operators: * and /

The operators * and / perform multiplication and division on all primitive numeric types and `char`. Integer division will generate an `ArithmeticException` when attempting to divide by zero.

You probably understand multiplication and division quite well from years of rote learning at school. In programming, of course, some limitations are imposed by the representation of numbers in a computer. These limitations apply to all number formats, from `byte` to `double`, but are perhaps most noticeable in integer arithmetic.

If you multiply or divide two integers, the result will be calculated using integer arithmetic in either `int` or `long` representation. If the numbers are large enough, the result will be bigger than the maximum number that can be represented, and the final value will be meaningless. This condition is referred to as *overflow*. For example, `byte` values can represent a range of –128 to +127, so if two particular bytes have the values 64 and 4, then multiplying them should, arithmetically, give a value of 256 (100000000 in binary—note that this value has nine digits). Actually, when you store the result in a `byte` variable, you will get a value of 0, because only the low-order eight bits of the result can be represented.

On the other hand, when you divide with integer arithmetic, the result is forced into an integer and, typically, a lot of information that would have formed a fractional part of the answer is lost. This condition is referred to as *underflow*. For example, 7 / 4 should give 1.75, but integer arithmetic will result in a value of 1. You therefore have a choice in many expressions: multiply first and then divide, which risks overflow, or divide first and then multiply, which risks underflow. Conventional wisdom says that you should multiply first and then divide, because this at least might work perfectly, whereas dividing first almost definitely loses precision. Consider this example:

```
 1. int a = 12345, b = 234567, c, d;
 2. long e, f;
 3.
 4. c = a * b / b; // this should equal a, that is, 12345
 5. d = a / b * b; // this should also equal a
 6. System.out.println("a is " + a +
 7.    "\nb is " + b +
 8.    "\nc is " + c +
 9.    "\nd is " + d);
10.
11. e = (long)a * b / b;
12. f = (long)a / b * b;
13. System.out.println(
14.    "\ne is " + e +
15.    "\nf is " + f);
```

The output from this code is

```
a is 12345
b is 234567
c is -5965
d is 0
e is 12345
f is 0
```

Do not worry about the exact numbers in this example. The important feature is that in the case where multiplication is performed first, the calculation overflows when performed with `int` values, resulting in a nonsense answer. However, the result is correct if the representation is wide enough—as when using the `long` variables. In both cases, dividing first has a catastrophic effect on the result, regardless of the width of the representation.

The Modulo Operator: %

Although multiplication and division are generally familiar operations, the *modulo operator* is perhaps less well known. The modulo operator gives a value that is related to the remainder of a division. It is

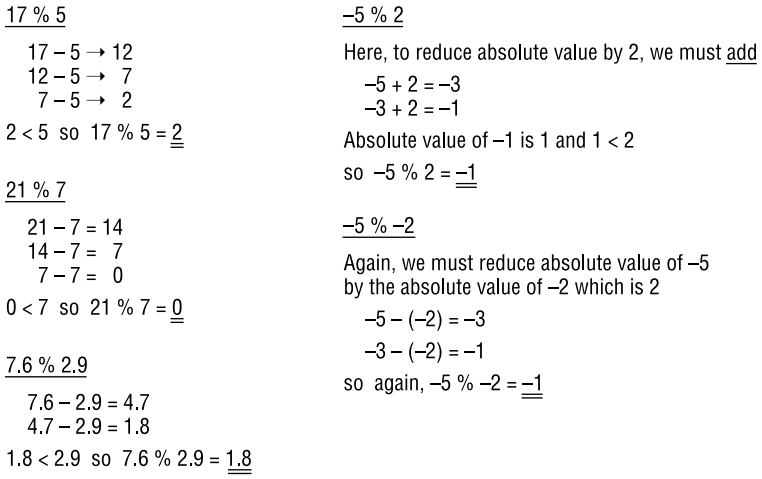
generally applied to two integers, although it can be applied to floating-point numbers, too. In school, we learned that 7 divided by 4 gives 1, remainder 3. In Java, we say `x = 7 % 4;` and expect that `x` will have the value 3.

The previous paragraph describes the essential behavior of the modulo operator, but additional concerns appear if you use negative or floating-point operands. In such cases, follow this procedure:

1. Reduce the magnitude of the left operand by the magnitude of the right one.
2. Repeat until the magnitude of the result is less than the magnitude of the right operand.

This result is the result of the modulo operator. Figure 2.1 shows some examples of this process.

FIGURE 2.1 Calculating the result of the modulo operator for a variety of conditions



Note that the sign of the result is entirely determined by the sign of the left operand. When the modulo operator is applied to floating-point types, the effect is to perform an integral number of subtractions, leaving a floating-point result that might well have a fractional part.

A useful rule of thumb for dealing with modulo calculations that involve negative numbers is this: Simply drop any negative signs from either operand and calculate the result. Then, if the original left operand was negative, negate the result. The sign of the right operand is irrelevant.

The modulo operation involves division during execution. As a result, it can throw an `ArithmeticException` if it's applied to integral types and the second operand is 0.

You might not have learned about the modulo operator in school, but you will certainly recognize the + and – operators. Although basically familiar, the + operator has some capabilities beyond simple addition.

The Addition and Subtraction Operators: + and -

The operators + and – perform addition and subtraction. They apply to operands of any numeric type but, uniquely, + is also permitted where either operand is a `String` object.

How the + Operator Is Overloaded

Java does not allow the programmer to perform operator overloading, but the + operator is overloaded by the language. This is not surprising, because in most languages that support multiple arithmetic types, the arithmetic operators (+, -, *, /, and so forth) are overloaded to handle these different types. Java, however, further overloads the + operator to support *concatenation*—that is, joining together—of `String` objects. The use of + with `String` arguments also performs conversions, and these can be succinct and expressive if you understand them. First, we will consider the use of the + operator in its conventional role of numeric addition.

Overloading is the term given when the same name is used for more than one piece of code, and the code that is to be used is selected by the argument or operand types provided. For example, the `println()` method can be given a `String` argument or an `int`. These two uses actually refer to entirely different methods; only the name is reused. Similarly, the + symbol is used to indicate addition of `int` values, but the exact same symbol is also used to indicate the addition of `float` values. These two forms of addition require entirely different code to execute; again, the operand types are used to decide which code is to be run. Where an operator can take different operand types, we refer to *operator overloading*. Some languages, but not Java, allow you to use operator overloading to define multiple uses of operators for your own types. Overloading is described in detail in Chapter 6, "Objects and Classes."

Where the + operator is applied to purely numeric operands, its meaning is simple and familiar. The operands are added together to produce a result. Of course, some promotions might take place, according to the normal rules, and the result might overflow. Generally, however, numeric addition behaves as you would expect. Promotions are discussed in a later section, "Arithmetic Promotion of Operands."

If overflow or underflow occurs during numeric addition or subtraction, then data is lost but no exception occurs. A more detailed description of behavior in arithmetic error conditions appears in a later section, "Arithmetic Error Conditions." Most of the new understanding to be gained about the + operator relates to its role in concatenating text.

Where either of the operands of a + expression is a `String` object, the meaning of the operator is changed from numeric addition to string concatenation. In order to achieve this result, both operands must be handled as text. If both operands are in fact `String` objects, this is simple. If, however, one of the operands is not a `String` object, then the non-`String` operand is converted to a `String` object before the concatenation takes place.

How Operands Are Converted to String Objects

Although a review of the certification objectives will show that the Certification Exam does not require it, it is useful in practice to know a little about how + converts operands to `String` objects. For object types, conversion to a `String` object is performed simply by invoking the `toString()` method of that object. The `toString()` method is defined in `java.lang.Object`, which is the root of the class hierarchy, and therefore all objects have a `toString()` method.

Sometimes, the effect of the `toString()` method is to produce rather cryptic text that is suitable only for debugging output, but it definitely exists and may legally be called.

Conversion of an operand of primitive type to a `String` is typically achieved by using, indirectly, the conversion utility methods in the wrapper classes. So, for example, an `int` value is converted by the static method `Integer.toString()`.

The `toString()` method in the `java.lang.Object` class produces a `String` that contains the name of the object's class and some identifying value—typically its reference value, separated by the at symbol (`@`). For example, this `String` might look like `java.lang.Object@1cc6dd`. This behavior is inherited by subclasses unless they deliberately override it.

It is a good idea to define a helpful `toString()` method in all your classes, even if you do not require it as part of the class behavior. Code the `toString()` method so that it represents the state of the object in a fashion that can assist in debugging; for example, output the names and values of the main instance variables.

To prepare for the Certification Exam questions, and to use the + operator effectively in your own programs, you should understand the following points:

- For a + expression with two operands of primitive numeric type, the result
  - Is of a primitive numeric type.
  - Is at least `int`, because of normal promotions.
  - Is of a type at least as wide as the wider of the two operands.
  - Has a value calculated by promoting the operands to the result type, and then performing the addition using that type. This might result in overflow or loss of precision.
- For a + expression with any operand that is not of primitive numeric type,
  - At least one operand must be a `String` object or literal; otherwise, the expression is illegal.
  - Any remaining non-`String` operands are converted to `String`, and the result of the expression is the concatenation of all operands.
- To convert an operand of some object type to a `String`, the conversion is performed by invoking the `toString()` method of that object.
- To convert an operand of a primitive type to a `String`, the conversion is performed by a static method in a container class, such as `Integer.toString()`.

If you want to control the formatting of the converted result, you should use the facilities in the `java.text` package.

Now that you understand arithmetic operators and the concatenation of text using the + operator, you should realize that sometimes arithmetic does not work as intended—it could result in an error of some kind. The next section discusses what happens under such error conditions.

Arithmetic Error Conditions

We expect arithmetic to produce "sensible" results that reflect the mathematical meaning of the expression being evaluated. However, because the computation is performed on a machine with specific limits on its ability to represent numbers, calculations can sometimes result in errors. You saw, in the section on the multiplication and division operators, that overflow and underflow can occur if the operands are too large or too small. In exceptional conditions, the following rules apply:

- Integer division by zero, including modulo (%) operation, results in an `ArithmeticException`.
- No other arithmetic causes any exception. Instead, the operation proceeds to a result, even though that result might be arithmetically incorrect.
- Floating-point calculations represent out-of-range values using the IEEE 754 infinity, minus infinity, and Not a Number (NaN) values. Named constants representing these are declared in both the `Float` and `Double` classes.
- Integer calculations, other than division by zero, that cause overflow or a similar error simply leave the final, typically truncated bit pattern in the result. This bit pattern is derived from the operation and the number representation and might even be of the wrong sign. Because the operations and number representations do not depend upon the platform, neither do the result values under error conditions.

These rules describe the effect of error conditions, but some additional significance is associated with the NaN values. NaN values are used to indicate that a calculation has no result in ordinary arithmetic, such as some calculations involving infinity or the square root of a negative number.

Comparisons with Not a Number

Some floating-point calculations can return a NaN. This occurs, for example, as a result of calculating the square root of a negative number. Two NaN values are defined in the `java.lang` package (`Float.NaN` and `Double.NaN`) and are considered non-ordinal for comparisons. This means that for *any* value of `x`, including NaN itself, all of the following comparisons will return false:

```
x < Float.NaN
x <= Float.NaN
x == Float.NaN
x > Float.NaN
x >= Float.NaN
```

In fact, the test

```
Float.NaN != Float.NaN
```

and the equivalent with `Double.NaN` return true, as you might deduce from the item indicating that `x == Float.NaN` gives false even if `x` contains `Float.NaN`.

The most appropriate way to test for a NaN result from a calculation is to use the `Float.isNaN(float)` or `Double.isNaN(double)` static method provided in the `java.lang` package.

The next section discusses a concept often used for manipulating bit patterns read from I/O ports: the shift operators <<, >>, and >>>.

The Shift Operators: <<, >>, and >>>

Java provides three *shift operators*. Two of these, << and >>, are taken directly from C/C++, but the third, >>>, is new in Java.

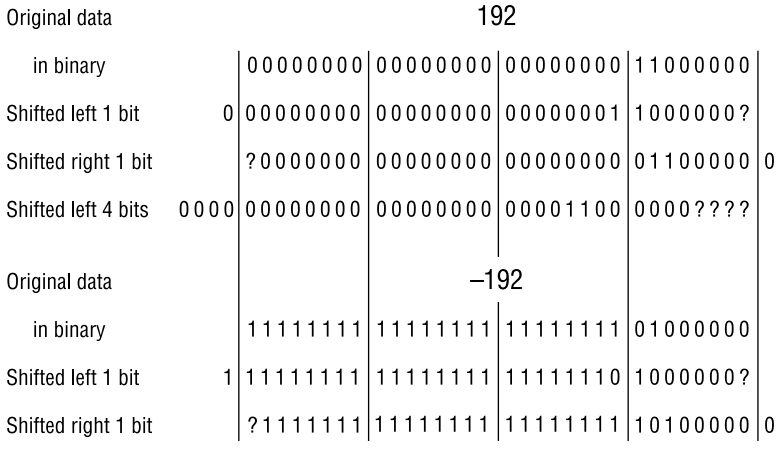
Shifting is common in control systems, where it can align bits that are read from or that are to be written to I/O ports. It can also provide efficient integer multiplication or division by powers of two. In Java, because the bit-level representation of all types is well defined and platform independent, you can use shifting with confidence.

Fundamentals of Shifting

Shifting is, on the face of it, a simple operation: it involves taking the binary representation of a number and moving the bit pattern left or right. However, the unsigned right-shift operator >>> is a common source of confusion.

The shift operators may be applied to arguments of integral types only. In fact, they should generally be applied only to operands of either `int` or `long` type. This is a consequence of the effects of promotion in expressions (see "Arithmetic Promotion of Operands" later in this chapter). Figure 2.2 illustrates the basic mechanism of shifting.

FIGURE 2.2 The basic mechanisms of shifting



The diagram in Figure 2.2 shows the fundamental idea of shifting, which involves moving the bits that represent a number to positions either to the left or right of their starting points. This raises two questions:

- What happens to the bits that "fall off" the end? The type of the result will have the same number of bits as the original value, but the result of a shift looks as if it might have more bits than that original.
- What defines the value of the bits that are shifted in? These bits are marked by question marks in Figure 2.2.

The first question has a simple answer. Bits that move off the end of a representation are discarded.

In some languages, mostly assembly languages, an additional operation called *rotate* uses these bits to define the value of the bits at the other end of the result. Java, like most high-level languages, does not provide a rotate operation.

Shifting Negative Numbers

The second question, regarding the value of the bits that are shifted in, requires more attention. In the case of the left-shift << and the unsigned right-shift >>> operators, the new bits are set to 0. However, in the case of the signed right-shift >> operator, the new bits take the value of the most significant bit before the shift. Figure 2.3 shows this. Notice that where a 1 bit is in the most significant position before the shift (indicating a negative number), 1 bits are introduced to fill the spaces introduced by shifting. Conversely, when a 0 bit is in the most significant position before the shift, 0 bits are introduced during the shift.

FIGURE 2.3 Signed right shift of positive and negative numbers

Original data	192			
in binary	00000000	00000000	00000000	11000000
Shifted right 1 bit	00000000	00000000	00000000	01100000
Shifted right 7 bits	00000000	00000000	00000000	00000001
Original data	–192			
in binary	11111111	11111111	11111111	01000000
Shifted right 1 bit	11111111	11111111	11111111	10100000
Shifted right 7 bits	11111111	11111111	11111111	11111110

This might seem like an arbitrary and unduly complex rule governing the bits that are shifted in during a signed right-shift operation, but there is a good reason for the rule. If a binary number is shifted left one position (and provided none of the bits that move off the ends of a left-shift operation are lost), the effect of the shift is to double the original number. Shifts by more than one bit effectively double and double again, so the result is as if the number had been multiplied by 2, 4, 8, 16, and so on.

Real World Scenario

A Closer Look at Shifting

Write an application that illustrates the functionality of Java's shift operators: <<, >>, and >>>. The application should take two command-line `int` arguments, `x` and `y`, and should output the results of the following shifts:

```
x << y
x >> y
x >>> y
```

The values of `x` and the shift result should be displayed in both decimal and 32-bit binary. Sample output from the solution looks like this:

```
>java ShowShift -64 5

-64 << 5 = -2048
11111111111111111111111111000000 << 5 =
11111111111111111111100000000000

-64 >> 5 = -2
11111111111111111111111111000000 >> 5 =
11111111111111111111111111111110

-64 >>> 5 = 134217726
11111111111111111111111111000000 >>> 5 =
00000111111111111111111111111110
```

A static method called `toBinaryString()` in the `Integer` wrapper class outputs a binary string. Unfortunately, it truncates leading 0s, so it doesn't always give a 32-bit result. Try creating your own integer-to-binary-string method, using a shift operator. Play with the command-line input values `x` and `y`. Can you find values such that `x` is positive and `x >>> y` is negative? Can you find values such that `x` is negative and `x >>> y` is positive?

If shifting the bit pattern of a number left by one position doubles that number, then you might reasonably expect that shifting the pattern right, which apparently puts the bits back where they came from, would halve the number, returning it to its original value. If the right shift results in 0 bits being added at the most significant bit positions, then for positive numbers, this division does result. However, if the original number was negative, then the assumption is false.

Notice that with the negative number in two's-complement representation, the most significant bit is 1. In order to preserve the significance of a right shift as a division by two when dealing with negative numbers, you must bring in bits set to 1, rather than 0. This is how the behavior of the arithmetic right shift is determined. If a number is positive, its most significant bit is 0 and when shifting right, more 0 bits are brought in. However, if the number is negative, its most significant bit is 1, and more 1 bits must be propagated in when the shift occurs. This is illustrated in the examples in Figure 2.4.

FIGURE 2.4 Shifting positive and negative numbers right

Original data	192			
in binary	00000000	00000000	00000000	11000000
Shifted right 1 bit = 96 = 192 / 2	00000000	00000000	00000000	01100000
Shifted right 4 bits = 12 = 192 / 16 = 192 / 2^4	00000000	00000000	00000000	00001100
Original data	–192			
in binary	11111111	11111111	11111111	01000000
Shifted right 1 bit = –96 = –192 / 2	11111111	11111111	11111111	10100000
Shifted right 4 bits = –12 = –192 / 16 = –192 / 2^4	11111111	11111111	11111111	11110100

There is a feature of the arithmetic right shift that differs from simple division by 2. If you divide -1 by 2, the result will be 0. However, the result of the arithmetic shift right of -1 right is -1. You can think of this as the shift operation rounding down, whereas the division rounds toward 0.

You now have two right-shift operators: one that treats the left integer argument as a bit pattern with no special arithmetic significance and another that attempts to ensure that the arithmetic equivalence of shifting right with division by powers of two is maintained.

Why does Java need a special operator for unsigned shift right, when neither C nor C++ required it? The answer is simple: both C and C++ provide for unsigned numeric types, but Java does not. If you shift an unsigned value right in either C or C++, you get the behavior associated with the >>> operator in Java. However, this does not work in Java simply because the numeric types (other than `char`) are signed.

Reduction of the Right Operand

The right argument of the shift operators is taken to be the number of bits by which the shift should move. However, for shifting to behave properly, this value should be smaller than the number of bits in the result. That is, if the shift is being done as an `int` type, then the right operand should be less than 32. If the shift is being done as `long`, then the right operand should be less than 64.

In fact, the shift operators do not reject values that exceed these limits. Instead, they calculate a new value by reducing the supplied value modulo the number of bits. This means that if you attempt to shift an `int` value by 33 bits, you will actually shift by `33 % 32`—that is, by only one bit. This shift produces an anomalous result. You would expect that shifting a 32-bit number by 33 bits would produce 0 as a result (or possibly –1 in the signed right-shift case). However, because of the reduction of the right operand, this is not the case.

Why Java Reduces the Right Operand of Shift Operators, or "The Sad Story of the Sleepy Processor"

The first reason for reducing the number of bits to shift modulo the number of bits in the left operand is that many CPUs implement the shift operations in this way. Why should CPUs do this?

Some years ago, a powerful and imaginatively designed CPU provided both shift and rotate operations and could shift by any number of bits specified by any of its registers. Because the registers were wide, this was a very large number, and as each bit position shifted took a finite time to complete, the effect was that you could code an instruction that would take minutes to complete.

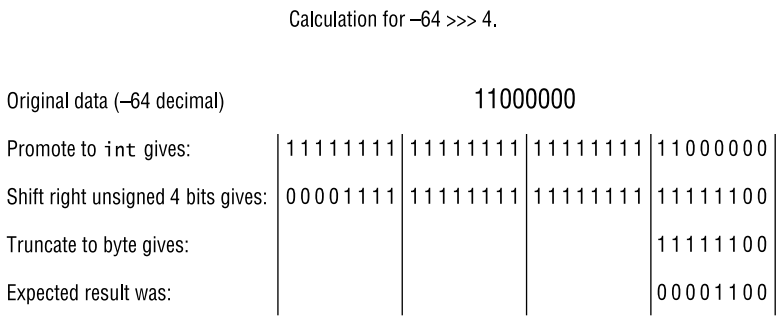
One of the intended target applications of this particular CPU was in control systems, and one of the most important features of real-time control systems is the worst-case time to respond to an external event, known as the *interrupt latency*. Unfortunately, because a single instruction on this CPU was indivisible—so that interrupts could not be serviced until it was complete—execution of a large shift instruction effectively crippled the CPU. The next version of that CPU changed the implementations of shift and rotate so that the number of bits by which to shift or rotate was treated as being limited to the size of the target data item. This change restored a sensible interrupt latency. Since then, many other CPUs have adopted reduction of the right operand.

Arithmetic Promotion of Operands

Arithmetic promotion of operands takes place before any binary operator is applied so that all numeric operands are at least `int` type. This promotion has an important consequence for the unsigned right-shift operator when applied to values that are narrower than `int`.

The diagram in Figure 2.5 shows the process by which a `byte` is shifted right. First the byte is promoted to an `int`, which is done treating the `byte` as a signed quantity. Next, the shift occurs, and 0 bits are indeed propagated into the top bits of the result—but these bits are not part of the original `byte`. When the result is cast down to a `byte` again, the high-order bits of that `byte` appear to have been created by a signed shift right, rather than an unsigned one. This is why you should generally not use the logical right-shift operator with operands smaller than an `int`: It is unlikely to produce the result you expected.

FIGURE 2.5 Unsigned right shift of a byte



The Comparison Operators

Comparison operators—<, <=, >, >=, ==, and !=—return a `boolean` result; the relation as written is either true or it is false. Additionally, the `instanceof` operator determines whether or not a given object is an instance of a particular class. These operators are commonly used to form conditions, such as in `if()` statements or in loop control. There are three types of comparison: *ordinal* comparisons test the relative value of numeric operands. *Object-type comparisons* determine whether the runtime type of an object is of a particular type or a subclass of that particular type. *Equality comparisons* test whether two values are the same and may be applied to values of non-numeric types.

Ordinal Comparisons with <, <=, >, and >=

The ordinal comparison operators are

- Less than: <
- Less than or equal to: <=
- Greater than: >
- Greater than or equal to: >=

These are applicable to all numeric types and to `char` and produce a `boolean` result.

So, for example, given the following declarations,

```
int p = 9;
int q = 65;
int r = -12;
float f = 9.0F;
char c = 'A';
```

the following tests all return true:

```
p < q
f < q
f <= c
c > r
c >= q
```

Notice that arithmetic promotions are applied when these operators are used. This is entirely according to the normal rules discussed in Chapter 4. For example, although it would be an error to attempt to assign, say, the `float` value 9.0F to the `char` variable `c`, it is perfectly acceptable to compare the two. To achieve the result, Java promotes the smaller type to the larger type; hence the `char` value 'A' (represented by the Unicode value 65) is promoted to a `float` 65.0F. The comparison is then performed on the resulting `float` values.

Although the ordinal comparisons operate satisfactorily on dissimilar numeric types, including `char`, they are not applicable to any non-numeric types. They cannot take `boolean` or any class-type operands.

The *instanceof* Operator

The `instanceof` operator tests the class of an object at runtime. The left argument can be any object reference expression, usually a variable or an array element, whereas the right operand must be a class, interface, or array type. You cannot use a `java.lang.Class` object reference or a `String` representing the name of the class as the right operand. A compiler error results if the left operand cannot be cast to the right operand. (Casting is discussed in Chapter 4.)

This code fragment shows an example of how `instanceof` may be used. Assume that a class hierarchy exists with `Person` as a base class and `Parent` as a subclass:

```
1. public class Classroom {
2.   private Hashtable inTheRoom = new Hashtable();
3.   public void enterRoom(Person p) {
```

```
 4.     inTheRoom.put(p.getName(), p);
 5.   }
 6.   public Person getParent(String name) {
 7.     Object p = inTheRoom.get(name);
 8.     if (p instanceof Parent) {
 9.       return (Parent)p;
10.     }
11.     else {
12.       return null;
13.     }
14.   }
15. }
```

The method `getParent()` at lines 6–14 checks to see if the `Hashtable` contains a parent with the specified name. This is done by first searching the `Hashtable` for an entry with the given name and then testing to see if the entry that is returned is actually a `Parent`. The `instanceof` operator returns true if the class of the left argument is the same as, or is some subclass of, the class specified by the right operand.

The right operand may equally well be an interface. In such a case, the test determines if the object at the left argument implements the specified interface.

You can also use the `instanceof` operator to test whether a reference refers to an array. Because arrays are themselves objects in Java, this is natural enough, but the test that is performed actually checks two things: First, it checks if the object is an array, and then it checks if the element type of that array is some subclass of the element type of the right argument. This is a logical extension of the behavior that is shown for simple types and reflects the idea that an array of, say, `Button` objects is an array of `Component` objects, because a `Button` is a `Component`. A test for an array type looks like this:

```
if (x instanceof Component[])
```

Note, however, that you cannot simply test for "any array of any element type," as the syntax. This line is not legal:

```
if (x instanceof [])
```

Neither is it sufficient to test for arrays of `Object` element type like this:

```
if (x instanceof Object [])
```

because the array might be of a primitive base type, in which case the test will fail.

Although it is not required by the Certification Exam, you might find it useful to know that you can determine if an object is in fact an array without regard to the base type. You can do this using the `isArray()` method of the `Class` class. For example, this test returns true if the variable `myObject` refers to an array: `myObject.getClass().isArray()`.

If the left argument of the `instanceof` operator is a `null` value, the `instanceof` test simply returns false; it does not cause an exception.

The Equality Comparison Operators: == and !=

The operators == and != test for equality and inequality, respectively, returning a `boolean` value. For primitive types, the concept of equality is quite straightforward and is subject to promotion rules so that, for example, a `float` value of 10.0 is considered equal to a `byte` value of 10. For variables of object type, the "value" is taken as the reference to the object—typically, the memory address. You should not use these operators to compare the contents of objects, such as strings, because they will return true if two references refer to the same object, rather than if the two objects have equivalent value. *Object comparisons* compare the data of two objects, whereas *reference comparisons* compare the memory locations of two objects.

To achieve a content or semantic comparison, for example, so that two different `Double` objects containing the value 1234 are considered equal, you must use the `equals()` method rather than the == or != operator.

To operate appropriately, the `equals()` method must have been defined for the class of the objects you are comparing. To determine whether it has, check the documentation supplied with the JDK or, for third-party classes, produced by javadoc. The documentation should report that an `equals()` method is defined for the class and overrides `equals()` in some superclass. If this is not indicated, then you should assume that the `equals()` method will not produce a useful content comparison. You also need to know that `equals()` is defined as accepting an `Object` argument, but the actual argument must be of the same type as the object upon which the method is invoked—that is, for `x.equals(y)`, the test `y instanceof the-type-of-x` must be true. If this is not the case, then `equals()` must return false.

Defining an *equals()* Method

The information in this warning is not required for the Certification Exam but is generally of value when writing real programs. If you define an `equals()` method in your own classes, you should be careful to observe three rules, or else your classes might behave incorrectly in some specific circumstances.

First, the argument to the `equals()` method is an `Object`; you must avoid the temptation to make the argument to `equals()` specific to the class you are defining. If you do this, you have overloaded the `equals()` method, not overridden it, and functionality in other parts of the Java APIs that depends on the `equals()` method will fail. Perhaps most significantly, lookup methods in containers, such as `containsKey()` and `get()` in the `HashMap`, will fail.

The second rule is that the `equals()` method should be commutative: the result of `x.equals(y)` should always be the same as the result of `y.equals(x)`.

The final rule is that if you define an `equals()` method, you should also define a `hashCode()` method. This method should return the same value for objects that compare equal using the `equals()` method. Again, this behavior is needed to support the containers and other classes. A minimal but acceptable behavior for the `hashCode()` method is simply to return 1. Doing so removes any efficiency gains that hashing would give, forcing a `HashMap` to behave like a linked list when storing such objects, but at least the behavior is correct.

The Bitwise Operators: &, ^, and |

The *bitwise operators* &, ^, and | provide bitwise AND, eXclusive-OR (XOR), and OR operations, respectively. They are applicable to integral types. Collections of bits are sometimes used to save storage space where several `boolean` values are needed or to represent the states of a collection of binary inputs from physical devices.

The bitwise operations calculate each bit of their results by comparing the corresponding bits of the two operands on the basis of these three rules:

- For AND operations, 1 AND 1 produces 1. Any other combination produces 0.
- For XOR operations, 1 XOR 0 produces 1, as does 0 XOR 1. (All these operations are commutative.) Any other combination produces 0.
- For OR operations, 0 OR 0 produces 0. Any other combination produces 1.

The names AND, XOR, and OR are intended to be mnemonic for these operations. You get a 1 result from an AND operation if both the first operand *and* the second operand are 1. An XOR gives a 1 result if one *or* the other operand, but not both (this is the *exclusiveness*), is 1. In the OR operation, you get a 1 result if either the first operand *or* the second operand (*or* both) is 1. These rules are represented in Tables 2.3 through 2.5.

TABLE 2.3 The AND Operation

Op1	Op2	Op1 AND Op2
0	0	0
0	1	0
1	0	0
1	1	1

TABLE 2.4 The XOR Operation

Op1	Op2	Op1 XOR Op2
0	0	0
0	1	1
1	0	1
1	1	0

TABLE 2.5 The OR Operation

Op1	Op2	Op1 OR Op2
0	0	0
0	1	1
1	0	1
1	1	1

Compare the rows of each table with the corresponding rule for the operations listed in the previous bullets. You will see that for the AND operation, the only situation that leads to a 1 bit as the result is when both operands are 1 bits. For XOR, a 1 bit results when one or the other (but not both) of the operands is a 1 bit. Finally, for the OR operation, the result is a 1 bit, except when both operands are 0 bits. Now let's see how this concept works when applied to whole binary numbers, rather than just single bits. The approach can be applied to any size of integer, but we will look at bytes because they serve to illustrate the idea without putting so many digits on the page as to cause confusion. Consider this example:

```
    00110011
    11110000
AND ---------
    00110000
```

Observe that each bit in the result is calculated solely on the basis of the two bits appearing directly above it in the calculation. The next calculation looks at the least significant bit:

```
    00110011
    11110000
AND ---------
    00110000
```

This result bit is calculated as 1 and 0, which gives 0.

For the fourth bit from the left, see the following calculation:

```
    00110011
    11110000
AND ---------
    00110000
```

This result bit is calculated as 1 AND 1, which gives 1. All the other bits in the result are calculated in the same fashion, using the two corresponding bits and the rules stated earlier.

Exclusive-OR operations are done by a comparable approach, using the appropriate rules for calculating the individual bits, as the following calculations show:

```
    00110011          00110011
    11110000          11110000
XOR ---------     XOR ---------
    11000011          11000011
```

All the highlighted bits are calculated as either 1 XOR 0 or as 0 XOR 1, producing 1 in either case.

```
    00110011
    11110000
XOR ---------
    11000011
```

In the previous calculation, the result bit is 0 because both operand bits were 1.

```
     00110011
     11110000
XOR  --------
     11000011
```

And here, the 0 operand bits also result in a 0 result bit.

The OR operation again takes a similar approach, but with its own rules for calculating the result bits. Consider this example:

```
     00110011
     11110000
OR   --------
     11110011
```

Here, the two operand bits are 1 and 0, so the result is 1.

```
     00110011
     11110000
OR   --------
     11110011
```

However, in this calculation, both operand bits are 0, which is the condition that produces a 0 result bit for the OR operation.

Although programmers usually apply these operators to the bits in integer variables, it is also permitted to apply them to `boolean` operands.

Boolean Operations

The `&`, `^`, and `|` operators behave in fundamentally the same way when applied to arguments of `boolean`, rather than integral, types. However, instead of calculating the result on a bit-by-bit basis, the `boolean` values are treated as single bits, with true corresponding to a 1 bit and false to a 0 bit. The general rules discussed in the previous section may be modified like this when applied to `boolean` values:

- For AND operations, true AND true produces true. Any other combination produces false.
- For XOR operations, true XOR false produces true, and false XOR true produces true. Other combinations produce false.
- For OR operations, false OR false produces false. Any other combination produces true.

These rules are represented in Tables 2.6 through 2.8.

TABLE 2.6 The AND Operation on *boolean* Values

Op1	Op2	Op1 AND Op2
false	false	false
false	true	false
true	false	false
true	true	true

TABLE 2.7 The XOR Operation on *boolean* Values

Op1	Op2	Op1 XOR Op2
false	false	false
false	true	true
true	false	true
true	true	false

TABLE 2.8 The OR Operation on *boolean* Values

Op1	Op2	Op1 OR Op2
false	false	false
false	true	true
true	false	true
true	true	true

Again, compare these tables with the rules stated in the bulleted list. Also compare them with Tables 2.3 through 2.5, which describe the same operations on bits. You will see that 1 bits are replaced by true, and 0 bits are replaced by false.

As with all operations, the two operands must be of compatible types. So, if either operand is of `boolean` type, both must be. Java does not permit you to cast any type to `boolean`; instead you must use comparisons or methods that return `boolean` values.

The next section covers the short-circuit logical operators. These operators perform logical AND and OR operations, but are slightly different in implementation from the operators just discussed.

The Short-Circuit Logical Operators

The short-circuit logical operators `&&` and `||` provide logical AND and OR operations on `boolean` types. Note that no XOR operation is provided. Superficially, these operators are similar to the `&` and `|` operators, with the limitation of being applicable only to `boolean` values and not integral types. However, the `&&` and `||` operations have a valuable additional feature: the ability to "short circuit" a calculation if the result is definitely known. This feature makes these operators central to a popular `null`-reference-handling idiom in Java programming. They can also improve efficiency.

The main difference between the `&` and `&&` and between the `|` and `||` operators is that the right operand might not be evaluated in the latter cases. We will look at how this happens in the rest of this section. This behavior is based on two mathematical rules that define conditions under which the result of a `boolean` AND or OR operation is entirely determined by one operand without regard for the value of the other:

- For an AND operation, if one operand is false, the result is false, without regard to the other operand.
- For an OR operation, if one operand is true, the result is true, without regard to the other operand.

To put it another way, for any `boolean` value X:

- false AND X = false
- true OR X = true

Given these rules, if the left operand of a `boolean` AND operation is false, then the result is definitely false, whatever the right operand. It is therefore unnecessary to evaluate the right operand. Similarly, if the left operand of a `boolean` OR operation is true, the result is definitely true and the right operand need not be evaluated.

Consider a fragment of code intended to print out a `String` if that `String` exists and is longer than 20 characters:

```
1. if (s != null) {
2.   if (s.length() > 20) {
3.     System.out.println(s);
4.   }
5. }
```

However, the same operation can be coded very succinctly like this:

```
1. if ((s != null) && (s.length() > 20)) {
2.   System.out.println(s);
3. }
```

If the `String` reference `s` is `null`, then calling the `s.length()` method would raise a `NullPointerException`. In both of these examples, however, the situation never arises. In the second example, avoiding execution of the `s.length()` method is a direct consequence of the short-circuit behavior of the && operator. If the test `(s != null)` returns false (if `s` is in fact `null`), then the whole test expression is guaranteed to be false. Where the first operand is false, the && operator does not evaluate the second operand; so, in this case, the expression `(s.length() > 20)` is not evaluated.

Although these shortcuts do not affect the result of the operation, side effects might well be changed. If the evaluation of the right operand involves a side effect, then omitting the evaluation will change the overall meaning of the expression in some way. This behavior distinguishes these operators from the bitwise operators applied to `boolean` types. Consider these fragments:

```
//first example:
1. int val = (int)(2 * Math.random());
2. boolean test = (val == 0) || (++val == 2);
3. System.out.println("test = " + test + "\nval = " + val);
//second example:
1. int val = (int)(2 * Math.random());
2. boolean test = (val == 0) | (++val == 2);
3. System.out.println("test = " + test + "\nval = " + val);
```

The first example will sometimes print:

```
test = true
val = 0
```

and sometimes it will print:

```
test = true
val = 2
```

The second example will sometimes print:

```
test = true
val = 1
```

and sometimes it will print:

```
test = true
val = 2
```

The point is that in the case of the short circuit operator, if `val` starts out at 0, then the second part of the expression (`++val`) is never executed, and `val` remains at 0. Alternatively, `val` starts at 1 and is incremented to 2. In the second case, the non-short-circuit version, the increment always occurs, and `val` ends up as either 1 or 2, depending on the original value returned by the `random()` method. In all cases, the value of `test` is true, because either `val` starts out at 0, or it starts at 1 and the test (`++val == 2`) is true.

So, the essential points about the && and `||` operators are as follows:

- They accept `boolean` operands.
- They evaluate the right operand only if the outcome is not certain based solely on the left operand. This is determined using these identities:
  - false AND X = false
  - true OR X = true

The next section discusses the ternary, or conditional, operator. Like the short-circuit logical operators, this operator may be less familiar than others, especially to programmers without a background in C or C++.

The Conditional Operator: *?:*

The *conditional operator* `?:` (also known as a *ternary operator*, because it takes three operands) provides a way to code simple conditions (`if/else`) into a single expression. The (`boolean`) expression to the left of the `?` is evaluated. If true, the result of the whole expression is the value of the expression to the left of the colon; otherwise it is the value of the expression to the right of the colon. The expressions on either side of the colon must be assignment-compatible with the result type.

For example, if `a`, `b`, and `c` are `int` variables, and `x` is a `boolean`, then the statement `a = x ? b : c;` is directly equivalent to the textually longer version:

```
1. if (x) {
2.   a = b;
3. }
```

```
4. else {
5.   a = c;
6. }
```

Of course x, a, b, and c can all be complex expressions if you desire.

Many people do not like the conditional operator, and in some companies its use is prohibited by the local style guide. This operator does keep source code more concise, but in many cases an optimizing compiler will generate equally compact and efficient code from the longer, and arguably more readable, `if/else` approach. One particularly effective way to abuse the conditional operator is to nest it, producing expressions of the form `a = b ? c ? d : e ? f : g : h ? i : j ? k : l;`. Whatever your feelings or corporate mandate, you should at least be able to read this operator, because you will find it used by other programmers.

Here are the points you should review for handling conditional operators in an exam question, or to use them properly in a program. In an expression of the form `a = x ? b : c;`:

- The types of the expressions b and c should be compatible and are made identical through conversion.
- The type of the expression x should be `boolean`.
- The types of the expressions b and c should be assignment-compatible with the type of a.
- The value assigned to a will be b if x is true or will be c if x is false.

Now that we have discussed the conditional (ternary) operator, only one group of operators remains: the assignment operators.

The Assignment Operators

Assignment operators set the value of a variable or expression to a new value. Assignments are supported by a battery of operators. Simple assignment uses =. Operators such as += and *= provide compound "calculate and assign" functions. These compound operators take a general form *op*=, where *op* can be any of the binary non-`boolean` operators already discussed. In general, for any compatible expressions *x* and *y*, the expression `x` *op*`= y` is a shorthand for `x = x` *op* `y`. However, there are two differences you must know. First, be aware that side effects in the expression *x* are evaluated exactly once, not twice, as the expanded view might suggest. The second issue is that the assignment operators include an implicit cast. Consider this situation:

```
1. byte x = 2;
2. x += 3;
```

If this had been written using the longhand approach

```
1. byte x = 2;
2. x = (byte)(x + 3);
```

the cast to `byte` would have been necessary because the result of an integer addition is at least an `int`. In the first case, using the assignment operator, this cast is implied. This is one of two situations where Java allows down-casting without explicit programmer intervention. (The other situation is in combined declaration and initialization.) Be sure to compare this with the general principles of assignment and casting laid out in Chapter 4.

The statement `x += 2;` involves typing two fewer characters, but is otherwise no more effective than the longer version `x = x + 2;` and is neither more nor less readable. However, if `x` is a complex expression, such as `target[temp.calculateOffset(1.9F) + depth++].item`, it is definitely more readable to express incrementing this value by 2 using the `+= 2` form. This is because these operators define that the exact same thing will be read on the right side as is written on the left side. So the maintainer does not have to struggle to decide whether the two complex expressions are actually the same, and the original programmer avoids some of the risk of mistyping a copy of the expression.

An Assignment Has Value

All the operators discussed up to this point have produced a value as a result of the operation. The expression 1 + 2, for example, results in a value 3, which can then be used in some further way—perhaps assignment to a variable. The assignment operators in Java are considered to be operators because they have a resulting value. So, given three `int` variables `a`, `b`, and `c`, the statement `a = b = c = 0;` is entirely legal. It is executed from right to left, so that first 0 is assigned into the variable `c`. After it has been executed, the expression `c = 0` takes the value that was assigned to the left side—that is, 0. Next, the assignment of `b` takes place, using the value of the expression to the right of the equals sign—again, 0. Similarly, that expression takes the value that was assigned, so finally the variable `a` is also set to 0.

Although *execution order* is determined by precedence and associativity, *evaluation order* of the arguments is not. Be sure you understand the points made in the section "Evaluation Order" at the start of this chapter.

As a general rule, avoid writing expressions that are complex enough for these issues to matter. A sequence of simply constructed expressions is easier to read and is less likely to cause confusion or other errors than complex ones. You are also likely to find that the compiler will optimize multiple simple expressions just as well as it would a single, very complex one.

Summary

We have covered a lot of material in this chapter, so let's recap some of the key points.

The unary operators were the first topics we covered. Recall that they take only a single operand. The seven unary operators are ++, --, +, -, !, ~, and (). Their key points are as follows:

- The ++ and -- operators increment and decrement expressions. The position of the operator (either prefix or suffix) is significant.
- The + operator has no effect on an expression other than to make it clear that a literal constant is positive. The - operator negates an expression's value.
- The ! operator inverts the value of a `boolean` expression.
- The ~ operator inverts the bit pattern of an integral expression.
- The (*type*) operator is used to persuade the compiler to permit certain assignments that the programmer believes are appropriate, but that break the normal, rigorous rules of the language. Its use is subject to extensive checks at compile time and runtime.

Next we covered arithmetic operators. We discussed in detail the four most familiar operators, which perform addition, subtraction, multiplication, and division. Recall that this group is further split into two subgroupings. There are five arithmetic operators:

- Multiplication: *
- Division: /
- Modulo: %
- Addition and `String` concatenation: +
- Subtraction: -

The arithmetic operators can be applied to any numeric type. Also, the + operator performs text concatenation if either of its operands is a `String` object. Under the conditions where one operand in a + expression is a `String` object, the other is forced to be a `String` object, too. Conversions are performed as necessary. They might result in cryptic text, but they are definitely legal.

Under conditions of arithmetic overflow or similar errors, accuracy is generally lost silently. Only integer division by zero can throw an exception. Floating-point calculations can produce NaN—indicating Not a Number (that is, the expression has no meaning in normal arithmetic)—or an infinity as their result under error conditions.

Java provides three shift operators. Two of them are derived directly from the C/C++ language, and a new one was added to Java.

These are the key points about the shift operators:

- The <<, >>, and >>> operators perform bit shifts of the binary representation of the left operand.
- The operands should be an integral type, generally either `int` or `long`.

- The right operand is reduced `modulo x`, where *x* depends on the type of the result of the operation. That type is either `int` or `long`, smaller operands being subjected to promotion. If the left operand is assignment-compatible with `int`, then *x* is 32. If the left operand is a `long`, then *x* is 64.
- The << operator shifts left. Zero bits are introduced at the least significant bit position.
- The >> operator performs a signed, or arithmetic, right shift. The result has 0 bits at the most significant positions if the original left operand was positive, and 1 bit at the most significant positions if the original left operand was negative. The result approximates dividing the left operand by two raised to the power of the right operand.
- The >>> operator performs an unsigned, or logical, right shift. The result has 0 bits at the most significant positions and might not represent a division of the original left operand.

We also discussed bitwise operators, which are sometimes used to save storage space, for instance. There are three bitwise operators: &, ^, and |. They are usually named AND, eXclusive-OR (XOR), and OR, respectively. For these operators, the following points apply:

- In bitwise operations, each result bit is calculated on the basis of the two bits from the same, corresponding position in the operands.
- For the AND operation, a 1 bit results if the first operand bit and the second operand bit are both 1.
- For the XOR operation, a 1 bit results only if exactly one operand bit is 1.
- For the OR operation, a 1 bit results if either the first operand bit `or` the second operand bit is 1.

For `boolean` operations, the arguments and results are treated as single-bit values with true represented by 1 and false by 0.

We described assignment operators, which set the value of a variable or expression to a new value. The key points about the assignment operators are as follows:

- Simple assignment, using =, assigns the value of the right operand to the left operand.
- The value of an object is its reference, not its contents.
- The right operand must be a type that is assignment-compatible with the left operand. Assignment compatibility and conversions are discussed in detail in Chapter 4.
- The assignment operators all return a value so that they can be used within larger expressions. The value returned is the value that was assigned to the left operand.
- The compound assignment operators, of the form *op*=, when applied in an expression like `a` *op*`= b;`, appear to behave like `a = a` *op* `b;`, except that the expression `a` and any of its side effects are evaluated only once.

Compound assignment operators exist for all binary, non-`boolean` operators: `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `>>>=`, `&=`, `^=`, and `|=`. We have now discussed all the operators provided by Java.

The ternary operator `?:` (also referred to as the conditional operator) requires three operands and provides the programmer with a more compact way to write an `if/else` statement.

The short circuit Boolean operators `&&` and `||` are binary operators that allow the programmer to circumvent evaluating one or more expressions, thereby making the code more efficient at runtime.

The remainder of the operators discussed in this chapter are the comparison operators `<`, `<=`, `>`, `>=`, and `instanceof`. These binary operators compare the left operand with the right operand and return a Boolean result of either `true` or `false`.

Exam Essentials

Understand the functionality of all the operators discussed in this section. These operators are

Unary operators: `++ -- + - ! ~`

The cast operator: `()`

Binary arithmetic operators: `* / % + -`

Shift operators: `<< >> >>>`

Comparison operators: `< <= > >= == != instanceof`

Bitwise operators: `& ^ |`

Short-circuit operators: `&& ||`

The ternary operator: `?:`

Assignment operators: `= op=`

Understand when arithmetic promotion takes place. You should know the type of the result of unary and binary arithmetic operations performed on operands of any type.

Understand the difference between object equality and reference equality; know the functionality of the `equals()` method of the `Object`, `Boolean`, and `String` classes. Object equality checks the data of two possibly distinct objects. Reference equality checks whether two references point to the same object. The `Object` version uses to a reference equality check; the `Boolean` and `String` versions compare encapsulated data.

Key Terms

Before you take the exam, be certain you are familiar with the following terms:

arithmetic operators
assignment operators
bitwise inversion
bitwise operators
comparison operators
concatenation
conditional operators
equality comparisons
modulo operators
object comparisons
object-type comparisons
ordinal
post-decrement
post-increment
pre-decrement
pre-increment
reference comparisons
shift operators
ternary operators
unary operators

Review Questions

1. After execution of the following code fragment, what are the values of the variables x, a, and b?

```
1. int x, a = 6, b = 7;
2. x = a++ + b++;
```

   A. x = 15, a = 7, b = 8

   B. x = 15, a = 6, b = 7

   C. x = 13, a = 7, b = 8

   D. x = 13, a = 6, b = 7

2. Which of the following expressions are legal? (Choose all that apply.)

   A. `int x = 6; x = !x;`

   B. `int x = 6; if (!(x > 3)) {}`

   C. `int x = 6; x = ~x;`

3. Which of the following expressions results in a positive value in x?

   A. `int x = -1; x = x >>> 5;`

   B. `int x = -1; x = x >>> 32;`

   C. `byte x = -1; x = x >>> 5;`

   D. `int x = -1; x = x >> 5;`

4. Which of the following expressions are legal? (Choose all that apply.)

   A. `String x = "Hello"; int y = 9; x += y;`

   B. `String x = "Hello"; int y = 9; if (x == y) {}`

   C. `String x = "Hello"; int y = 9; x = x + y;`

   D. `String x = "Hello"; int y = 9; y = y + x;`

   E. `String x = null;int y = (x != null) && (x.length() > 0) ? x.length() : 0;`

5. Which of the following code fragments would compile successfully and print "Equal" when run? (Choose all that apply.)

   A. `int x = 100; float y = 100.0F;if (x == y){System.out.println("Equal");}`

   B. `int x = 100; Integer y = new Integer(100);if (x == y) {`
      `System.out.println("Equal");}`

   C. `Integer x = new Integer(100);Integer y = new Integer(100);if (x == y) {`
      `System.out.println("Equal");}`

   D. `String x = new String("100");String y = new String("100");if (x == y) {`
      `System.out.println("Equal");}`

   E. `String x = "100";String y = "100";if (x == y) {`
      `System.out.println("Equal");}`

6. What results from running the following code?

```
1. public class Short {
2.   public static void main(String args[]) {
3.     StringBuffer s = new StringBuffer("Hello");
4.     if ((s.length() > 5) &&
5.       (s.append(" there").equals("False")))
6.       ; // do nothing
7.     System.out.println("value is " + s);
8.   }
9. }
```

A. The output: `value is Hello`

B. The output: `value is Hello there`

C. A compiler error at line 4 or 5

D. No output

E. A `NullPointerException`

7. What results from running the following code?

```
1. public class Xor {
2.   public static void main(String args[]) {
3.     byte b = 10; // 00001010 binary
4.     byte c = 15; // 00001111 binary
5.     b = (byte)(b ^ c);
6.     System.out.println("b contains " + b);
7.   }
8. }
```

A. The output: `b contains 10`

B. The output: `b contains 5`

C. The output: `b contains 250`

D. The output: `b contains 245`

8. What results from attempting to compile and run the following code?

```
1. public class Conditional {
2.   public static void main(String args[]) {
3.     int x = 4;
```

```
4.     System.out.println("value is " +
5.       ((x > 4) ? 99.99 : 9));
6.   }
7. }
```

A. The output: `value is 99.99`

B. The output: `value is 9`

C. The output: `value is 9.0`

D. A compiler error at line *5*

9. What is the output of this code fragment?

```
1. int x = 3; int y = 10;
2. System.out.println(y % x);
```

A. `0`

B. `1`

C. `2`

D. `3`

10. What results from the following fragment of code?

```
1. int x = 1;
2. String [] names = { "Fred", "Jim", "Sheila" };
3. names[--x] += ".";
4. for (int i = 0; i < names.length; i++) {
5.   System.out.println(names[i]);
6. }
```

A. The output includes `Fred.` with a trailing period.

B. The output includes `Jim.` with a trailing period.

C. The output includes `Sheila.` with a trailing period.

D. None of the outputs show a trailing period.

E. An `ArrayIndexOutOfBoundsException` is thrown.

Answers to Review Questions

1. C. The assignment statement is evaluated as if it were

```
x = a + b; a = a + 1; b = b + 1;
```

Therefore, the assignment to `x` is made using the sum of 6 + 7, giving 13. After the addition, the values of `a` and `b` are incremented; the new values, 7 and 8, are stored in the variables.

2. B, C. In A, the use of `!` is inappropriate, because `x` is of `int` type, not `boolean`. This is a common mistake among C and C++ programmers, because the expression would be valid in those languages. In B, the comparison is inelegant (being a cumbersome equivalent of `if (x <= 3))` but valid, because the expression `(x > 3)` is a `boolean` type and the `!` operator can properly be applied to it. In C, the bitwise inversion operator is applied to an integral type. The bit pattern of 6 looks like 0 0110 where the ellipsis represents 27 0 bits. The resulting bit pattern looks like 1 1001, where the ellipsis represents 27 1 bits.

3. A. In every case, the bit pattern for –1 is "all ones." In A, this pattern is shifted five places to the right with the introduction of 0 bits at the most significant positions. The result is 27 1 bits in the less significant positions of the `int` value. Because the most significant bit is 0, this result represents a positive value (134217727). In B, the shift value is 32 bits. This shift will result in no change at all to `x`, because the shift is actually performed by (32 mod 32) bits, which is 0. So in B, the value of `x` is unchanged at –1. C is illegal, because the result of `x >>> 5` is of type `int` and cannot be assigned into the `byte` variable `x` without explicit casting. Even if the cast were added, giving

```
byte x = -1; x = (byte)(x >>> 5);
```

the result of the expression `x >>> 5` would be calculated like this:

1. Promote `x` to an `int`. Doing so gives a sign-extended result—that is, an `int` –1 with 32 1 bits.
2. Perform the shift; it behaves the same as in A above, giving 134217727, which is the value of 27 1 bits in the less significant positions.
3. Casting the result of the expression simply retains the less significant 8 bits; because these are all 1s, the resulting `byte` represents –1.

Finally, D performs a signed shift, which propagates 1 bits into the most significant position. So, in this case, the resulting value of `x` is unchanged at –1.

4. A, C, E. In A, the use of `+=` is treated as a shorthand for the expression in C. This attempts to "add" an `int` to a `String`, which results in conversion of the `int` to a `String`—"9" in this case—and the concatenation of the two `String` objects. So in this case, the value of `x` after the code is executed is "Hello9".

In B, the comparison `(x == y)` is not legal, because variable `y` is an `int` type and cannot be compared with a reference value. Don't forget that comparison using `==` tests the values and that for objects, the "value" is the reference value and not the contents.

C is identical to A without the use of the shorthand assignment operator.

D calculates `y + x`, which is legal in itself, because it produces a `String` in the same way as did `x + y`. It then attempts to assign the result, which is "9Hello", into an `int` variable. Because the result of `y + x` is a `String`, this assignment is not permitted.

E is rather different from the others. The important points are the use of the short-circuit operator && and the conditional operator `?:`. The left operand of the && operator is always evaluated, and in this case the condition `(x != null)` is false. Because this is false, the right part of the expression `(x.length() > 0)` need not be evaluated, as the result of the && operator is known to be false. This short-circuit effect neatly avoids executing the method call `x.length()`, which would fail with a `NullPointerException` at runtime. This false result is then used in the evaluation of the conditional expression. Because the `boolean` value is false, the result of the overall expression is the value to the right of the colon, which is 0.

5. A, E. Although `int` and `float` are not assignment-compatible, they can generally be mixed on either side of an operator. Because == is not an assignment but is a comparison operator, it simply causes normal promotion, so that the `int` value 100 is promoted to a `float` value 100.0 and compared successfully with the other `float` value 100.0F. For this reason, A is true.

The code in B fails to compile, because of the mismatch between the `int` and the `Integer` object. The value of an object is its reference, and no conversions are ever possible between references and numeric types. As a result, the arguments cannot be promoted to the same type, and they cannot be compared.

In C, the code compiles successfully because the comparison is between two object references. However, the test for equality compares the value of the references (the memory address typically) and, because the variables `x` and `y` refer to two different objects, the test returns false. The code in D behaves exactly the same way.

Comparing E with D might persuade you that E should probably not print "Equal". In fact, it does so because of a required optimization. Because `String` objects are immutable, literal strings are inevitably constant strings, so the compiler re-uses the same `String` object if it sees the same literal value occur more than once in the source. This means that the variables `x` and `y` actually do refer to the same object; so the test `(x == y)` is true and the "Equal" message is printed. It is particularly important that you do not allow this special behavior to persuade you that the == operator can be used to compare the contents of objects in any general way.

6. A. The effect of the && operator is first to evaluate the left operand. That is the expression `(s.length() > 5)`. Because the length of the `StringBuffer` object `s` is 5, this test returns false. Using the logical identity false AND X = false, the value of the overall conditional is fully determined, and the && operator therefore skips evaluation of the right operand. As a result, the value in the `StringBuffer` object is still simply "Hello" when it is printed out.

If the test on the left side of && had returned true, as would have occurred had the `StringBuffer` contained a longer text segment, then the right side would have been evaluated. Although it might look a little strange, that expression, `(s.append("there").equals("False"))`, is valid and returns a `boolean`. In fact, the value of the expression is guaranteed to be false, because it is clearly impossible for any `StringBuffer` to contain exactly "False" when it has

just had the `String` "there" appended to it. This is irrelevant, however; the essence of this expression is that, if it is evaluated, it has the side effect of changing the original `StringBuffer` by appending the text "there".

7. B. The eXclusive-OR operator ^ works on the pairs of bits in equivalent positions in the two operands. In this example, this produces:

```
    00001010
    00001111
XOR --------
    00000101
```

Notice that the only 1 bits in the answer are in those columns where exactly one of the operands has a 1 bit. If neither or both of the operands has a 1, then a 0 bit results.

The value 00000101 binary corresponds to *5* decimal.

It is worth remembering that, although this example has been shown as a `byte` calculation, the actual work is done using `int` (32-bit) values. This is why the explicit cast is required before the result is assigned back into the variable `b` in line *5*.

8. C. In this code, the optional result values for the conditional operator, 99.99 (a `double`) and 9 (an `int`), are of different types. The result type of a conditional operator must be fully determined at compile time, and in this case the type chosen, using the rules of promotion for binary operands, is `double`. Because the result is a `double`, the output value is printed in a floating-point format.

The choice of which of the two values to output is made on the basis of the `boolean` value that precedes the `?`. Because `x` is 4, the test `(x > 4)` is false. This causes the overall expression to take the second of the possible values, which is 9 rather than 99.99. Because the result type is promoted to a `double`, the output value is written as 9.0, rather than the more obvious 9.

If the two possible argument types had been entirely incompatible—for example, `(x > 4) ? "Hello" : 9`—then the compiler would have issued an error at that line.

9. B. In this case, the calculation is relatively straightforward, because only positive integers are involved. Dividing 10 by 3 gives 3 remainder 1, and this 1 forms the result of the modulo expression. Another way to think of this calculation is 10 – 3 = 7, 7 – 3 = 4, 4 – 3 = 1, 1 is less than 3, therefore the result is 1. The second approach is more general, because it handles floating-point calculations, too. Don't forget that for negative numbers, you should ignore the signs during the calculation part, and simply attach the sign of the left operand to the result.

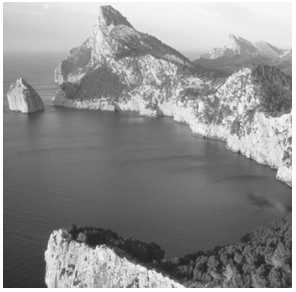
10. A. The assignment operators of the form *op*= evaluate the left expression only once. So, the effect of decrementing `x`, in `--x`, occurs only once, resulting in a value of 0 and not –1. Therefore, no out-of-bounds array accesses are attempted. The array element that is affected by this operation is "Fred", because the decrement occurs before the += operation is performed. Although `String` objects themselves are immutable, the references that are the array elements are not. It is entirely possible to cause the value `name[0]` to be modified to refer to a newly constructed `String`, which happens to be "Fred".

Chapter 3

Modifiers

JAVA CERTIFICATION EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- ✓ **1.2 Declare classes, nested classes, methods, instance variables, static variables, and automatic (method local) variables making appropriate use of all permitted modifiers (such as public, final, static, abstract, etc.). State the significance of each of these modifiers both singly and in combination and state the effect of package relationships on declared items qualified by these modifiers.**
- ✓ **4.1 Identify correctly constructed package declarations, import statements, class declarations (of all forms including inner classes) interface declarations, method declarations (including the main method that is used to start execution of a class), variable declarations, and identifiers.**



Modifiers are Java keywords that give the compiler information about the nature of code, data, or classes. Modifiers specify, for example, that a particular feature is static, final, or transient. (A *feature* is a class, a method, or a variable.) A group of modifiers, called *access modifiers*, dictates which classes are allowed to use a feature. Other modifiers can be used in combination to describe the attributes of a feature.

In this chapter you will learn about all of Java's modifiers as they apply to top-level classes. Inner classes are not discussed here but are covered in Chapter 6, "Objects and Classes".

Modifier Overview

The most common modifiers are the access modifiers: `public`, `protected`, and `private`. Access modifiers are covered in the next section. The remaining modifiers do not fall into any clear categorization. They are

- `final`
- `abstract`
- `static`
- `native`
- `transient`
- `synchronized`
- `volatile`

Each of these modifiers is discussed in its own section.

The Access Modifiers

Access modifiers control which classes may use a feature. A class's features are

- The class itself
- Its member variables
- Its methods and constructors

Note that, with rare exceptions, the only variables that may be controlled by access modifiers are class-level variables. The variables that you declare and use within a class's methods may not have access modifiers. This makes sense; a method variable can only be used within its method.

The access modifiers are

- `public`
- `protected`
- `private`

The only access modifier permitted to noninner classes is `public`; there is no such thing as a protected or private top-level class.

A feature may have at most one access modifier. If a feature has no access modifier, its access defaults to a mode that, unfortunately, has no standardized name. The default access mode is known variously as *friendly*, *package*, or *default*. In this book, we use the term *default*. Be aware that Sun is encouraging us to avoid the use of *friendly*, due to confusion with a somewhat similar concept in C++.

The following declarations are all legal (provided they appear in an appropriate context):

```
class Parser { ... }
public class EightDimensionalComplex { ... }
private int i;
Graphics offScreenGC;
protected double getChiSquared() { ... }
private class Horse { ... }
```

The following declarations are illegal:

```
public protected int x;       // At most 1 access modifier
default Button getBtn() {...} // ìdefaultî isn't a keyword
```

public

The most generous access modifier is `public`. A *public* class, variable, or method may be used in any Java program without restriction. An applet (a subclass of class `java.applet.Applet`) is declared as a public class so that it may be instantiated by browsers. An application declares its `main()` method to be public so that `main()` can be invoked from any Java runtime environment.

private

The least generous access modifier is `private`. Top-level (that is, not inner) classes may not be declared private. A *private* variable or method may only be used by an instance of the class

that declares the variable or method. For an example of the private access modifier, consider the following code:

```
 1. class Complex {
 2.   private double real, imaginary;
 3.
 4.   public Complex(double r, double i) {
 5.     real = r; imaginary = i;
 6.   }
 7.   public Complex add(Complex c) {
 8.     return new Complex(real + c.real,
 9.     imaginary + c.imaginary);
10.   }
11. }
12.
13. class Client {
14.   void useThem() {
15.     Complex c1 = new Complex(1, 2);
16.     Complex c2 = new Complex(3, 4);
17.     Complex c3 = c1.add(c2);
18.     double d = c3.real;   // Illegal!
19.   }
20. }
```

On line 17, a call is made to `c1.add(c2)`. Object `c1` will execute the method using object `c2` as a parameter. In line 8, `c1` accesses its own private variables as well as those of `c2`. There is nothing wrong with this. Declaring `real` and `imaginary` to be private means that they can only be accessed by instances of the `Complex` class, but they can be accessed by any instance of `Complex`. Thus `c1` may access its own `real` and `imaginary` variables, as well as the `real` and `imaginary` of any other instance of `Complex`. Access modifiers dictate which *classes*, not which *instances*, may access features.

Line 18 is illegal and will cause a compiler error. The error message says, "Variable real in class Complex not accessible from class Client". The private variable `real` may be accessed only by an instance of `Complex`.

Private data can be hidden from the very object that owns the data. If class `Complex` has a subclass called `SubComplex`, then every instance of `SubComplex` will inherit its own `real` and `imaginary` variables. Nevertheless, no instance of `SubComplex` can ever access those variables. Once again, the private features of `Complex` can only be accessed within the `Complex` class; an instance of a subclass is denied access. Thus, for example, the following code will not compile:

```
1. class Complex {
2.   private double real, imaginary;
3. }
```

```
 4.
 5.
 6. class SubComplex extends Complex {
 7.   SubComplex(double r, double i) {
 8.     real = r;            // Trouble!
 9.   }
10. }
```

In the constructor for class `SubComplex` (on line 8), the variable `real` is accessed. This line causes a compiler error, with a message that is very similar to the message of the previous example: "Undefined variable: real". The private nature of variable `real` prevents an instance of `SubComplex` from accessing one of its own variables!

Default

Default is the name of the access of classes, variables, and methods if you don't specify an access modifier. A class's data and methods may be default, as well as the class itself. A class's default features are accessible to any class in the same package as the class in question.

Default is not a Java keyword; it is simply a name that is given to the access level that results from not specifying an access modifier.

It would seem that default access is of interest only to people who are in the business of making packages. This is technically true, but actually everybody is always making packages, even if they aren't aware of it. The result of this behind-the-scenes package making is a degree of convenience for programmers that deserves investigation.

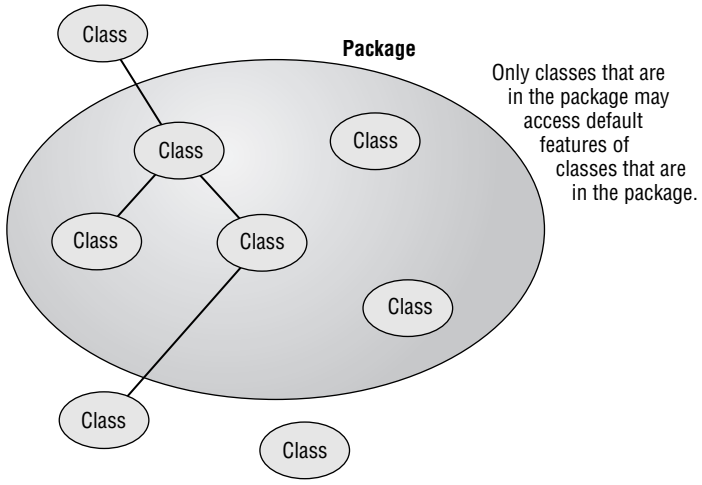
When you write an application that involves developing several different classes, you probably keep all your `.java` sources and all your `.class` class files in a single working directory. When you execute your code, you do so from that directory. The Java runtime environment considers that all class files in its current working directory constitute a package.

Imagine what happens when you develop several classes in this way and don't bother to provide access modifiers for your classes, data, or methods. These features are neither public nor private nor protected. They result in default access, which means they are accessible to any other classes in the package. Because Java considers that all the classes in the directory actually make up a package, all your classes get to access one another's features. This makes it easy to develop code quickly without worrying too much about access.

Now imagine what happens if you are deliberately developing your own package. A little extra work is required: You have to put a package statement in your source code, and you have to compile with the `-d` option. Any features of the package's classes that you do not explicitly mark with an access modifier will be accessible to all the members of the package, which is probably what you want. Fellow package members have a special relationship, and it stands to reason that they should get access not granted to classes outside the package. Classes outside the package may not access the default features, because the features are default, not public. Classes outside the package may subclass the classes in the package (you do something like this, for example, when you write an applet); however, even the subclasses may not access the default

features, because the features are default, not protected or public. Figure 3.1 illustrates default access both within and outside a package.

FIGURE 3.1 Default access



protected

The name *protected* is a bit misleading. From the sound of it, you might guess that protected access is extremely restrictive—perhaps the next closest thing to private access. In fact, protected features are even more accessible than default features.

Only variables and methods may be declared protected. A protected feature of a class is available to all classes in the same package, just like a default feature. Moreover, a protected feature of a class is available to all subclasses of the class that owns the protected feature. This access is provided even to subclasses that reside in a different package from the class that owns the protected feature.

As an example of the protected access modifier, consider the following code:

```
1. package sportinggoods;
2. class Ski {
3.   void applyWax() { . . . }
4. }
```

The `applyWax()` method has default access. Now consider the following subclass:

```
1. package sportinggoods;
2. class DownhillSki extends Ski {
3.   void tuneup() {
```

```
4.     applyWax();
5.     // other tuneup functionality here
6.   }
7. }
```

The subclass calls the inherited method `applyWax()`. This is not a problem as long as both the `Ski` and `DownhillSki` classes reside in the same package. However, if either class were to be moved to a different package, `DownhillSki` would no longer have access to the inherited `applyWax()` method, and compilation would fail. The problem would be fixed by making `applyWax()` protected on line 3:

```
1. package adifferentpackage;   // Class Ski now in
                                // a different package
2. class Ski {
3.   protected void applyWax() { . . . }
4. }
```

Real World Scenario

Protected Access in Depth

In this exercise, you will look at how protected data can be accessed from a subclass that belongs to a different package. Because access is enforced at compile time, you will not be writing any code that is intended to be executed. Rather, you will write several very simple classes and see which ones compile.

Begin by creating a public superclass called `Bird`, in a package called `birdpack`. This superclass should have a single data member: a protected `int` called `nFeathers`. Then, create four subclasses of `Bird`, all of which reside in a package called `duckpack`. Thus you will have subclasses whose package is different from their superclass's package; this is exactly the situation for which protected access is designed.

The first subclass, called `Duck1`, should have a method that accesses the `nFeathers` variable of the current instance of `Duck1`. Before compiling `Duck1`, ask yourself if the code should compile.

The second subclass, called `Duck2`, should have a method that constructs another instance of `Duck2` and accesses the `nFeathers` variable of the other instance. Before compiling `Duck2`, ask yourself if the code should compile.

The third subclass, called `Duck3`, should have a method that constructs an instance of `Bird` (the superclass) and accesses the `nFeathers` variable of the `Bird` instance. Before compiling `Duck3`, ask yourself if the code should compile.

The fourth subclass, called `Swan`, should have a method that constructs an instance of `Duck1` and accesses the `nFeathers` variable of that object. Before compiling `Swan`, ask yourself if the code should compile.

A note on compilation: When a source file contains a `package` declaration, it is generally most convenient to compile with the `-d` option. Doing so will ensure creation of an appropriate package directory hierarchy, with class files installed correctly. Thus, for example, the easiest way to compile `Bird.java` is with the following command line:

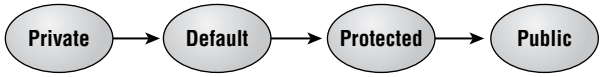
```
javac -d . Bird.java
```

Subclasses and Method Privacy

Java specifies that methods may not be overridden to be more private. For example, most applets provide an `init()` method, which overrides the do-nothing version inherited from the `java.applet.Applet` superclass. The inherited version is declared public, so declaring the subclass version to be private, protected, or default would result in a compiler error. The error message says, "Methods can't be overridden to be more private."

Figure 3.2 shows the legal access types for subclasses. A method with some particular access type may be overridden by a method with a different access type, provided there is a path in the figure from the original type to the new type.

FIGURE 3.2 Legal overridden method access

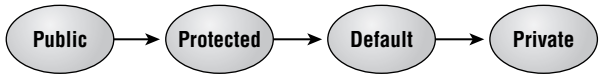


The rules for overriding can be summarized as follows:

- A private method may be overridden by a private, default, protected, or public method.
- A default method may be overridden by a default, protected, or public method.
- A protected method may be overridden by a protected or public method.
- A public method may only be overridden by a public method.

Figure 3.3 shows the illegal access types for subclasses. A method with some particular access type may not be shadowed by a method with a different access type, if there is a path in the figure from the original type to the new type.

FIGURE 3.3 Illegal overridden method access



The illegal overriding combinations can be summarized as follows:

- A default method may not be overridden by a private method.
- A protected method may not be overridden by a default or private method.
- A public method may not be overridden by a protected, default, or private method.

Summary of Access Modes

To summarize, Java's access modes are

public A public feature may be accessed by any class.

protected A protected feature may only be accessed by a subclass of the class that owns the feature or by a member of the same package as the class that owns the feature.

default A default feature may only be accessed by a class from the same package as the class that owns the feature.

private A private feature may only be accessed by the class that owns the feature.

Other Modifiers

The rest of this chapter covers Java's other modifiers: `final`, `abstract`, `static`, `native`, `transient`, `synchronized`, and `volatile`. (Transient and volatile are not mentioned in the Certification Exam objectives, so they are just touched on briefly in this chapter.)

Java does not care about order of appearance of modifiers. Declaring a class to be `public final` is no different from declaring it `final public`. Declaring a method to be `protected static` has the same effect as declaring it `static protected`.

Not every modifier can be applied to every kind of feature. Table 3.1, at the end of this chapter, summarizes which modifiers apply to which features.

final

The *final* modifier applies to classes, methods, and variables. The meaning of `final` varies from context to context, but the essential idea is the same: final features may not be changed.

A final class cannot be subclassed. For example, the following code will not compile, because the `java.lang.Math` class is final:

```
class SubMath extends java.lang.Math { }
```

The compiler error says, "Can't subclass final classes".

A final variable cannot be modified once it has been assigned a value. In Java, final variables play the same role as `consts` in C++ and `#define` constants in C. For example, the `java.lang.Math` class has a final variable, of type `double`, called `PI`. Obviously, pi is not the sort of value that should be changed during the execution of a program.

If a final variable is a reference to an object, it is the reference that must stay the same, not the object. This is shown in the following code:

```
 1.  class Walrus {
 2.    int weight;
 3.    Walrus(int w) { weight = w; }
 4.  }
 5.
 6.  class Tester {
 7.    final Walrus w1 = new Walrus(1500);
 8.    void test() {
 9.      w1 = new Walrus(1400);    // Illegal
10.      w1.weight = 1800;         // Legal
11.    }
12. }
```

Here the final variable is `w1`, declared on line 7. Because it is final, `w1` cannot receive a new value; line 9 is illegal. However, the data inside `w1` is not final, and line 10 is perfectly legal. In other words,

- You may *not* change a final object reference variable.
- You *may* change data owned by an object that is referred to by a final object reference variable.

A final method may not be overridden. For example, the following code will not compile:

```
1. class Mammal {
2.   final void getAround() { }
3. }
4.
5. class Dolphin extends Mammal {
6.   void getAround() { }
7. }
```

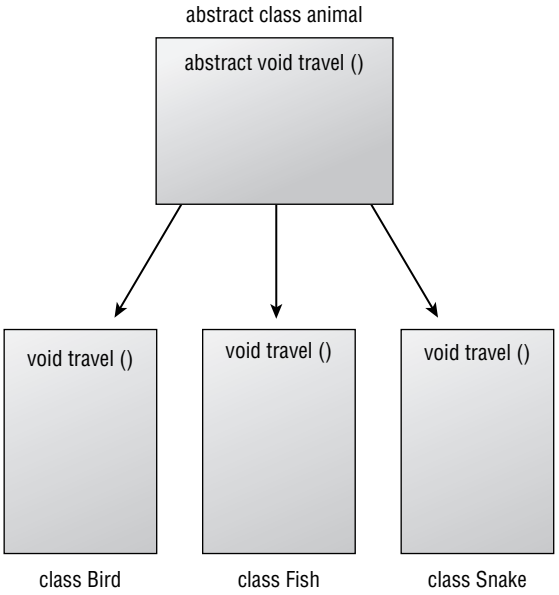
Dolphins get around in a very different way from most mammals, so it makes sense to try to override the inherited version of `getAround()`. However, `getAround()` is final, so the only result is a compiler error at line 6 that says, "Final methods can't be overridden."

abstract

The *abstract* modifier can be applied to classes and methods. A class that is abstract may not be instantiated (that is, you may not call its constructor).

Abstract classes provide a way to defer implementation to subclasses. Consider the class hierarchy shown in Figure 3.4.

FIGURE 3.4 A class hierarchy with abstraction



The designer of class `Animal` has decided that every subclass should have a `travel()` method. Each subclass has its own unique way of traveling, so it is not possible to provide `travel()` in the superclass and have each subclass inherit the same parental version. Instead, the `Animal` superclass declares `travel()` to be `abstract`. The declaration looks like this:

```
abstract void travel();
```

At the end of the line is a semicolon where you would expect to find curly braces containing the body of the method. The method body—its implementation—is deferred to the subclasses. The superclass provides only the method name and signature. Any subclass of `Animal` must provide an implementation of `travel()` or declare itself to be abstract. In the latter case, implementation of `travel()` is deferred yet again, to a subclass of the subclass.

If a class contains one or more abstract methods, the compiler insists that the class must be declared abstract. This is a great convenience to people who will be using the class: they need to look in only one place (the class declaration) to find out if they are allowed to instantiate the class directly or if they have to build a subclass.

In fact, the compiler insists that a class must be declared abstract if any of the following conditions is true:

- The class has one or more abstract methods.
- The class inherits one or more abstract method (from an abstract parent) for which it does not provide implementations.
- The class declares that it implements an interface but does not provide implementations for every method of that interface.

These three conditions are very similar to one another. In each case, there is an incomplete class. Some part of the class's functionality is missing and must be provided by a subclass.

In a way, `abstract` is the opposite of `final`. A final class, for example, may not be subclassed; an abstract class *must* be subclassed.

static

The *static* modifier can be applied to variables, methods, and even a strange kind of code that is not part of a method. You can think of static features as belonging to a class, rather than being associated with an individual instance of the class.

The following example shows a simple class with a single static variable:

```
1. class Ecstatic{
2.   static int x = 0;
3.   Ecstatic() { x++; }
4. }
```

Variable `x` is static; this means that there is only one `x`, no matter how many instances of class `Ecstatic` might exist at any moment. There might be one `Ecstatic` instance, or many, or even none; yet there is always precisely one `x`. The 4 bytes of memory occupied by `x` are allocated when class `Ecstatic` is loaded. The initialization to 0 (line 2) also happens at class-load time. The static variable is incremented every time the constructor is called, so it is possible to know how many instances have been created.

You can reference a static variable two ways:

- Via a reference to any instance of the class
- Via the class name

The first method works, but it can result in confusing code and is considered bad form. The following example shows why:

```
1. Ecstatic e1 = new Ecstatic();
2. Ecstatic e2 = new Ecstatic();
3. e1.x = 100;
4. e2.x = 200;
5. reallyImportantVariable = e1.x;
```

If you didn't know that `x` is static, you might think that `reallyImportantVariable` gets set to 100 in line 5. In fact, it gets set to 200, because `e1.x` and `e2.x` refer to the same (static) variable.

A better way to refer to a static variable is via the class name. The following code is identical to the previous code:

```
1. Ecstatic e1 = new Ecstatic();
2. Ecstatic e2 = new Ecstatic();
```

```
3. Ecstatic.x = 100;    // Why did I do this?
4. Ecstatic.x = 200;
5. reallyImportantVariable = Ecstatic.x;
```

Now it is clear that line 3 is useless, and the value of `reallyImportantVariable` gets set to 200 in line 5. Referring to static features via the class name rather than an instance results in source code that more clearly describes what will happen at runtime.

Methods, as well as data, can be declared static. Static methods are not allowed to use the nonstatic features of their class (although they are free to access the class's static data and call its other static methods). Thus static methods are not concerned with individual instances of a class. They may be invoked before even a single instance of the class is constructed. Every Java application is an example, because every application has a `main()` method that is static:

```
1. class SomeClass {
2.   static int i = 48;
3.   int j = 1;
4.
5.   public static void main(String args[]) {
6.     i += 100;
7.     // j *= 5; Lucky for us this is commented out!
8.   }
9. }
```

When this application is started (that is, when somebody types **`java SomeClass`** on a command line), no instance of class `SomeClass` exists. At line 6, the `i` that gets incremented is static, so it exists even though there are no instances. Line 7 would result in a compiler error if it were not commented out, because `j` is nonstatic.

Instance methods have an implicit variable named `this`, which is a reference to the object executing the method. In nonstatic code, you can refer to a variable or method without specifying which object's variable or method you mean. The compiler assumes you mean `this`. For example, consider the following code:

```
1. class Xyzzy {
2.   int w;
3.
4. void bumpW() {
5.     w++;
6.   }
7. }
```

On line 5, the programmer has not specified which object's `w` is to be incremented. The compiler assumes that line 5 is an abbreviation for `this.w++;`.

With static methods, there is no `this`. If you try to access an instance variable or call an instance method within a static method, you will get an error message that says, "Undefined variable: this." The concept of "the instance that is executing the current method" does not mean anything, because there is no such instance. Like static variables, static methods are not associated with any individual instance of their class.

If a static method needs to access a nonstatic variable or call a nonstatic method, it must specify which instance of its class owns the variable or executes the method. This situation is familiar to anyone who has ever written an application with a GUI:

```
 1.  import java.awt.*;
 2.
 3.  public class MyFrame extends Frame {
 4.    MyFrame() {
 5.      setSize(300, 300);
 6.    }
 7.
 8.    public static void main(String args[]) {
 9.      MyFrame theFrame = new MyFrame();
10.      theFrame.setVisible(true);
11.    }
12.  }
```

In line 9, the static method `main()` constructs an instance of class `MyFrame`. In the next line, that instance is told to execute the (nonstatic) method `setVisible()`. This technique bridges the gap from static to nonstatic, and it is frequently seen in applications.

The following code, for example, will not compile:

```
1. class Cattle {
2.   static void foo() {}
3. }
4.
5. class Sheep extends Cattle {
6.   void foo() {}
7. }
```

The compiler flags line 6 with the message, "Static methods can't be overridden." If line 6 were changed to `static void foo() { }`, then compilation would succeed. Static methods can appear to be overridden—a superclass and a subclass can have static methods with identical names, argument lists, and return types—but technically this is not considered overriding because the methods are static.

To summarize static methods:

- A static method may only access the static data of its class; it may not access nonstatic data.
- A static method may only call the static methods of its class; it may not call nonstatic methods.
- A static method has no `this`.
- A static method may not be overridden to be nonstatic.

Static Initializers

It is legal for a class to contain static code that does not exist within a method body. A class may have a block of initializer code that is simply surrounded by curly braces and labeled `static`. For example:

```
 1.  public class StaticExample {
 2.    static double d=1.23;
 3.
 4.    static {
 5.      System.out.println("Static code: d=" + d++);
 6.    }
 7.
 8.    public static void main(String args[]) {
 9.      System.out.println("main: d = " + d++);
10.    }
11.  }
```

Something seems to be missing from line 4. You might expect to see a complete method declaration there: `static void printAndBump()`, for example, instead of just `static`. In fact, line 4 is perfectly valid; it is known as *static initializer* code. The code inside the curlies is executed exactly once, at the time the class is loaded. At class-load time, all static initialization (such as line 2) and all free-floating static code (such as lines 4–6) are executed in order of appearance within the class definition.

Free-floating initializer code should be used with caution because it can easily be confusing and unclear. The compiler supports multiple initializer blocks within a class, but there is never a good reason for having more than one such block.

native

The *native* modifier can refer only to methods. Like the `abstract` keyword, `native` indicates that the body of a method is to be found elsewhere. In the case of abstract methods, the body

is in a subclass; with native methods, the body lies entirely outside the Java Virtual Machine (JVM), in a library.

Native code is written in a non-Java language, typically C or C++, and compiled for a single target machine type. (Thus Java's platform independence is violated.) People who port Java to new platforms implement extensive native code to support GUI components, network communication, and a broad range of other platform-specific functionality. However, it is rare for application and applet programmers to need to write native code.

One technique, however, is of interest in light of the last section's discussion of static code. When a native method is invoked, the library that contains the native code ought to be loaded and available to the JVM; if it is not loaded, there will be a delay. The library is loaded by calling `System.loadLibrary ("library_name")` and, to avoid a delay, it is desirable to make this call as early as possible. Often programmers will use the technique shown in the following code sample, which assumes the library name is `MyNativeLib`:

```
1. class NativeExample {
2.   native void doSomethingLocal(int i);
3.
4.   static {
5.     System.loadLibrary("MyNativeLib");
6.   }
7. }
```

Notice the `native` declaration on line 2, which declares that the code that implements `doSomethingLocal()` resides in a local library. Lines 4–6 are static initializer code, so they are executed at the time that class `NativeExample` is loaded; this ensures that the library will be available by the time somebody needs it.

Callers of native methods do not have to know that the method is native. The call is made in exactly the same way as if it were nonnative:

```
1. NativeExample natex;
2. natex = new NativeExample();
3. natex.doSomethingLocal(5);
```

Many common methods are native, including the `clone()` and `notify()` methods of the `Object` class.

transient

The *transient* modifier applies only to variables. A transient variable is not stored as part of its object's persistent state.

Many objects (specifically, those that implement the `Serializable` or `Externalizable` interfaces) can have their state serialized and written to some destination outside the JVM. This is done by passing the object to the `writeObject()` method of the `ObjectOutputStream` class.

If the stream is chained to a `FileOutputStream`, then the object's state is written to a file. If the stream is chained to a socket's `OutputStream`, then the object's state is written to the network. In both cases, the object can be reconstituted by reading it from an `ObjectInputStream`.

Sometimes an object contains extremely sensitive information. Consider the following class:

```
1. class WealthyCustomer
2. extends Customer implements Serializable {
3.   private float $wealth;
4.   private String accessCode;
5. }
```

Once an object is written to a destination outside the JVM, none of Java's elaborate security mechanisms is in effect. If an instance of this class were to be written to a file or to the Internet, somebody could snoop the access code. Line 4 should be marked with the `transient` keyword:

```
1. class WealthyCustomer
2. extends Customer implements Serializable {
3.   private float $wealth;
4.   private transient String accessCode;
5. }
```

Now the value of `accessCode` will not be written out during serialization.

synchronized

The *synchronized* modifier is used to control access to critical code in multithreaded programs. Multithreading is an extensive topic in its own right and is covered in Chapter 7, "Threads."

volatile

The last modifier is *volatile*. It is mentioned here only to make our list complete; it is not mentioned in the exam objectives and is not yet in common use. Only variables may be volatile; declaring them so indicates that such variables might be modified asynchronously, so the compiler takes special precautions. Volatile variables are of interest in multiprocessor environments.

Modifiers and Features

Not all modifiers can be applied to all features. Top-level classes may not be protected. Methods may not be transient. Static is so general that you can apply it to free-floating blocks of code.

Table 3.1 shows all the possible combinations of features and modifiers. Note that classes here are strictly top-level (that is, not inner) classes. (Inner classes are covered in Chapter 6.)

TABLE 3.1 All Possible Combinations of Features and Modifiers

Modifier	Class	Variable	Method	Constructor	FreeFloating Block
`public`	yes	yes	yes	yes	no
`protected`	no	yes	yes	yes	no
(default)*	yes	yes	yes	yes	yes
`private`	no	yes	yes	yes	no
`final`	yes	yes	yes	no	no
`abstract`	yes	no	yes	no	no
`static`	no	yes	yes	no	yes
`native`	no	no	yes	no	no
`transient`	no	yes	no	no	no
`volatile`	no	yes	no	no	no
`synchronized`	no	no	yes	no	yes

* *Default* is not a modifier; it is just the name of the access if no modifier is specified.

Summary

The focus of this chapter was to understand how all of the modifiers work and how they *can* or *cannot* work together. Some modifiers can be used in combination. Java's access modifiers are

- `public`
- `protected`
- `private`

If a feature does not have an access modifier, its access is "default."

Java's other modifiers are

- `final`
- `abstract`
- `static`
- `native`

- `transient`
- `synchronized`
- `volatile`

Exam Essentials

Understand the four access modes and the corresponding keywords. You should know the significance of public, default, protected, and private access when applied to data and methods.

Know the effect of declaring a final class, variable, or method. A final class cannot be subclassed; a final variable cannot be modified after initialization; a final method cannot be overridden.

Know the effect of declaring an abstract class or method. An abstract class cannot be instantiated; an abstract method's definition is deferred to a subclass.

Understand the effect of declaring a static variable or method. Static variables belong to the class; static methods have no `this` pointer and may not access nonstatic variables and methods of their class.

Know how to reference a static variable or method. A static feature may be referenced through the class name—the preferred method—or through a reference to any instance of the class.

Be able to recognize static initializer code. Static initializer code appears in curly brackets with no method declaration. Such code is executed once, when the class is loaded.

Key Terms

Before you take the exam, be certain you are familiar with the following terms:

abstract
access modifiers
default
feature
final
friendly
instance method
modifiers
native
package
private
protected
public
static
static initializer
synchronized
transient
volatile

Review Questions

1. Which of the following declarations are illegal? (Choose all that apply.)
   A. `default String s;`
   B. `transient int i = 41;`
   C. `public final static native int w();`
   D. `abstract double d;`
   E. `abstract final double hyperbolicCosine();`

2. Which of the following statements is true?
   A. An abstract class may not have any final methods.
   B. A final class may not have any abstract methods.

3. What is the minimal modification that will make this code compile correctly?

```
1. final class Aaa
2. {
3.     int xxx;
4.     void yyy() { xxx = 1; }
5. }
6.
7.
8. class Bbb extends Aaa
9. {
10.    final Aaa finalref = new Aaa();
11.
12.    final void yyy()
13.    {
14.        System.out.println("In method yyy()");
15.        finalref.xxx = 12345;
16.    }
17. }
```

   A. On line 1, remove the final modifier.
   B. On line 10, remove the `final` modifier.
   C. Remove line 15.
   D. On lines 1 and 10, remove the `final` modifier.
   E. The code will compile as is. No modification is needed.

4. Which of the following statements is true?
   A. Transient methods may not be overridden.
   B. Transient methods must be overridden.
   C. Transient classes may not be serialized.
   D. Transient variables must be static.
   E. Transient variables are not serialized.

5. Which statement is true about this application?

```
1. class StaticStuff
2  {
3.     static int x = 10;
4.
5.     static { x += 5; }
6.
7.     public static void main(String args[])
8.     {
9.         System.out.println("x = " + x);
10.    }
11.
12.    static {x /= 5; }
13. }
```

   A. Lines 5 and 12 will not compile because the method names and return types are missing.
   B. Line 12 will not compile because you can only have one static initializer.
   C. The code compiles and execution produces the output `x = 10`.
   D. The code compiles and execution produces the output `x = 15`.
   E. The code compiles and execution produces the output `x = 3`.

6. Which statement is true about this code?

```
1. class HasStatic
2. {
3.     private static int x = 100;
4.
5.     public static void main(String args[])
6.     {
7.         HasStatic hs1 = new HasStatic();
8.         hs1.x++;
9.         HasStatic hs2 = new HasStatic();
10.        hs2.x++;
```

```
11.         hs1 = new HasStatic();
12.         hs1.x++;
13.         HasStatic.x++;
14.         System.out.println("x = " + x);
15.     }
16. }
```

A. Line 8 will not compile because it is a static reference to a private variable.

B. Line 13 will not compile because it is a static reference to a private variable.

C. The program compiles and the output is `x = 102`.

D. The program compiles and the output is `x = 103`.

E. The program compiles and the output is `x = 104`.

7. Given the following code, and making no other changes, which combination of access modifiers (`public`, `protected`, or `private`) can legally be placed before `aMethod()` on line 3 and be placed before `aMethod()` on line 8?

```
1. class SuperDuper
2. {
3.     void aMethod() { }
4. }
5.
6. class Sub extends SuperDuper
7. {
8.     void aMethod() { }
9. }
```

A. line 3: public; line 8: private

B. line 3: protected; line 8: private

C. line 3: default; line 8: private

D. line 3: private; line 8: protected

E. line 3: public; line 8: protected

8. Which modifier or modifiers should be used to denote a variable that should not be written out as part of its class's persistent state? (Choose the shortest possible answer.)

A. `private`

B. `protected`

C. `private protected`

D. `transient`

E. `private transient`

9. This question concerns the following class definition:

```
1. package abcde;
2.
3. public class Bird {
4.   protected static int referenceCount = 0;
5.   public Bird() { referenceCount++; }
6.   protected void fly() { /* Flap wings, etc. */ }
7.   static int getRefCount() { return referenceCount; }
8. }
```

Which statement is true about class `Bird` and the following class `Parrot`?

```
1. package abcde;
2.
3. class Parrot extends abcde.Bird {
4.   public void fly() {
5.     /* Parrot-specific flight code. */
6.   }
7.   public int getRefCount() {
8.     return referenceCount;
9.   }
10. }
```

A. Compilation of `Parrot.java` fails at line 4 because method `fly()` is protected in the superclass, and classes `Bird` and `Parrot` are in the same package.

B. Compilation of `Parrot.java` fails at line 4 because method `fly()` is protected in the superclass and public in the subclass, and methods may not be overridden to be more public.

C. Compilation of `Parrot.java` fails at line 7 because method `getRefCount()` is static in the superclass, and static methods may not be overridden to be nonstatic.

D. Compilation of `Parrot.java` succeeds, but a runtime exception is thrown if method `fly()` is ever called on an instance of class `Parrot`.

E. Compilation of `Parrot.java` succeeds, but a runtime exception is thrown if method `getRefCount()` is ever called on an instance of class `Parrot`.

10. This question concerns the following class definition:

```
1. package abcde;
2.
3. public class Bird {
4.   protected static int referenceCount = 0;
5.   public Bird() { referenceCount++; }
6.   protected void fly() { /* Flap wings, etc. */ }
```

```
7.   static int getRefCount() { return referenceCount; }
8. }
```

Which statement is true about class `Bird` and the following class `Nightingale`?

```
1. package singers;
2.
3. class Nightingale extends abcde.Bird {
4.   Nightingale() { referenceCount++; }
5.
6.   public static void main(String args[]) {
7.     System.out.print("Before: " + referenceCount);
8.     Nightingale florence = new Nightingale();
9.     System.out.println("  After: " + referenceCount);
10.    florence.fly();
11.  }
12. }
```

A. The program will compile and execute. The output will be `Before: 0  After: 2`.

B. The program will compile and execute. The output will be `Before: 0  After: 1`.

C. Compilation of `Nightingale` will fail at line 4 because static members cannot be overridden.

D. Compilation of `Nightingale` will fail at line 10 because method `fly()` is protected in the superclass.

E. Compilation of `Nightingale` will succeed, but an exception will be thrown at line 10, because method `fly()` is protected in the superclass.

Answers to Review Questions

1. A, D, E. A is illegal because "default" is not a keyword. B is a legal transient declaration. C is strange but legal. D is illegal because only methods and classes may be abstract. E is illegal because `abstract` and `final` are contradictory.

2. B. Any class with abstract methods must itself be abstract, and a class may not be both abstract and final. Statement A says that an abstract class may not have final methods, but there is nothing wrong with this. The abstract class will eventually be subclassed, and the subclass must avoid overriding the parent's final methods. Any other methods can be freely overridden.

3. A. The code will not compile because on line 1, class `Aaa` is declared final and may not be subclassed. Lines 10 and 15 are fine. The instance variable `finalref` is final, so it may not be modified; it can only reference the object created on line 10. However, the data within that object is not final, so nothing is wrong with line 15.

4. E. A, B, and C don't mean anything because only variables may be transient, not methods or classes. D is false because transient variables may never be static. E is a good one-sentence definition of transient.

5. E. Multiple static initializers (lines 5 and 12) are legal. All static initializer code is executed at class-load time, so before `main()` is ever run, the value of `x` is initialized to 10 (line 3), then bumped to 15 (line 5), and then divided by 5 (line 12).

6. E. The program compiles fine; the "static reference to a private variable" stuff in answers A and B is nonsense. The static variable `x` gets incremented four times, on lines 8, 10, 12, and 13.

7. D. The basic principle is that a method may not be overridden to be more private. (See Figure 3.2 in this chapter.) All choices except D make the access of the overriding method more private.

8. D. The other modifiers control access from other objects within the Java Virtual Machine. Answer E also works but is not minimal.

9. C. Static methods may not be overridden to be nonstatic. B is incorrect because it states the case backward: methods can be overridden to be more public, not more private. Answers A, D, and E make no sense.

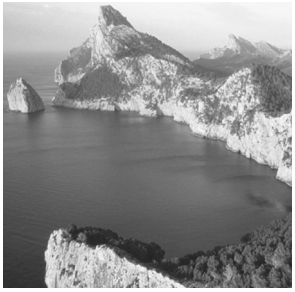
10. A. There is nothing wrong with `Nightingale`. The static `referenceCount` is bumped twice: once on line 4 of `Nightingale` and once on line 5 of `Bird`. (The no-argument constructor of the superclass is always implicitly called at the beginning of a class's constructor, unless a different superclass constructor is requested. This has nothing to do with modifiers; see Chapter 6.) Because `referenceCount` is bumped twice and not just once, answer B is wrong. C says that statics cannot be overridden, but no static method is being overridden on line 4; all that is happening is an increment of an inherited static variable. D is wrong because `protected` is precisely the access modifier you want `Bird.fly()` to have: you are calling `Bird.fly()` from a subclass in a different package. Answer E is ridiculous, but it uses credible terminology.

Chapter 4

Converting and Casting

JAVA CERTIFICATION EXAM OBJECTIVE COVERED IN THIS CHAPTER:

- ✓ **5.1 Determine the result of applying any operator including assignment operators, `instanceof`, and casts to operands of any type, class, scope, or accessibility, or any combination of these.**



Every Java variable has a type. Primitive data types include `int`, `long`, and `double`. Object reference data types may be classes (such as `Vector` or `Graphics`) or interfaces (such as `LayoutManager` or `Runnable`). There can also be arrays of primitives, objects, or arrays.

This chapter discusses the ways that a data value can change its type. Values can change type either explicitly or implicitly; that is, they change either at your request or at the system's initiative. Java places a lot of importance on type, and successful Java programming requires that you be aware of type changes.

Explicit and Implicit Type Changes

You can explicitly change the type of a value by *casting*. To cast an expression to a new type, just prefix the expression with the new type name in parentheses. For example, the following line of code retrieves an element from a vector, casts that element to type `Button`, and assigns the result to a variable called `btn`:

```
Button btn = (Button) (myVector.elementAt(5));
```

Of course, the sixth element of the vector must be capable of being treated as a `Button`. Compile-time rules and runtime rules must be observed. This chapter will familiarize you with those rules.

In some situations, the system implicitly changes the type of an expression without your explicitly performing a cast. For example, suppose you have a variable called `myColor` that refers to an instance of `Color`, and you want to store `myColor` in a vector. You would probably do the following:

```
myVector.add(myColor);
```

There is more to this code than meets the eye. The `add()` method of class `Vector` is declared with a parameter of type `Object`, not of type `Color`. As the argument is passed to the method, it undergoes an implicit type change. Such automatic, nonexplicit type changing is known as *conversion*. Conversion, like casting, is governed by rules. Unlike the casting rules, all conversion rules are enforced at compile time.

The number of casting and conversion rules is rather large, due to the large number of cases to be considered. (For example, can you cast a `char` to a `double`? Can you convert an interface to a final class? Yes to the first, no to the second.) The good news is that most of the rules accord with common sense, and most of the combinations can be generalized into rules of

thumb. By the end of this chapter, you will know when you can explicitly cast and when the system will implicitly convert on your behalf.

Primitives and Conversion

The two broad categories of Java data types are primitives and objects. *Primitive* data types are `int`s, `float`s, `boolean`s, and so on. (There are eight primitive data types in all; see Chapter 1, "Language Fundamentals," for a complete explanation of Java's primitives.) *Object* data types (or more properly, *object reference* data types) are the hundreds of classes and interfaces provided with the JDK, plus the infinitude of classes and interfaces to be invented by Java programmers.

Both primitive values and object references can be converted and cast, so you must consider four general cases:

- Conversion of primitives
- Casting of primitives
- Conversion of object references
- Casting of object references

The simplest topic is implicit conversion of primitives. All conversion of primitive data types takes place at compile time; this is the case because all the information needed to determine whether the conversion is legal is available at compile time.

Conversion of a primitive might occur in three contexts or situations:

- Assignment
- Method call
- Arithmetic promotion

The following sections deal with each of these contexts in turn.

Primitive Conversion: Assignment

Assignment conversion happens when you assign a value to a variable of a different type from the original value. For example:

```
1. int i;
2. double d;
3. i = 10;
4. d = i;  // Assign an int value to a double variable
```

Obviously, `d` cannot hold an integer value. At the moment that the fourth line of code is executed, the integer 10 that is stored in variable `i` gets converted to the double-precision value 10.0000000000000 (remaining zeros omitted for brevity).

The previous code is perfectly legal. Some assignments, on the other hand, are illegal. For example:

```
1. double d;
2. short s;
3. d = 1.2345;
4. s = d;   // Assign a double value to a short variable
```

This code will not compile. (The error message says, "Incompatible type for =.") The compiler recognizes that trying to cram a `double` value into a `short` variable is like trying to pour a quart of coffee into an eight-ounce teacup, as shown in Figure 4.1. It can be done (that is, the larger-to-smaller value assignment can be done; the coffee thing is impossible), but you must use an explicit cast, which will be explained in the following section.

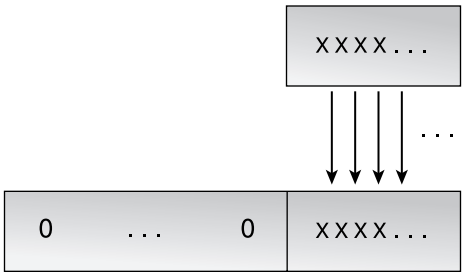
FIGURE 4.1 Illegal conversion of a quart to a cup, with loss of data

The general rules for primitive assignment conversion can be stated as follows:

- A `boolean` cannot be converted to any other type.
- A non-`boolean` can be converted to another non-`boolean` type, provided the conversion is a *widening conversion*.
- A non-`boolean` cannot be converted to another non-`boolean` type if the conversion would be a *narrowing conversion*.

Widening conversions change a value to a type that accommodates a wider range of values than the original type can accommodate. In most cases, the new type has more bits than the original and can be visualized as being "wider" than the original, as shown in Figure 4.2.

FIGURE 4.2 Widening conversion of a positive value

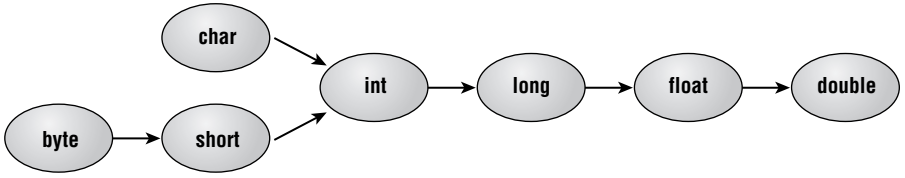


Widening conversions do not lose information about the magnitude of a value. In the first example in this section, an `int` value was assigned to a `double` variable. This conversion was legal because `doubles` are wider (represented by more bits) than `int`s, so there is room in a `double` to accommodate the information in an `int`. Java's widening conversions are

- From a `byte` to a `short`, an `int`, a `long`, a `float`, or a `double`
- From a `short` to an `int`, a `long`, a `float`, or a `double`
- From a `char` to an `int`, a `long`, a `float`, or a `double`
- From an `int` to a `long`, a `float`, or a `double`
- From a `long` to a `float` or a `double`
- From a `float` to a `double`

Figure 4.3 illustrates all the widening conversions. The arrows can be taken to mean "can be widened to." To determine whether it is legal to convert from one type to another, find the first type in the figure and see if you can reach the second type by following the arrows.

FIGURE 4.3 Widening conversions



The figure shows, for example, that it is perfectly legal to assign a `byte` value to a `float` variable, because you can trace a path from `byte` to `float` by following the arrows (`byte` to `short` to `int` to `long` to `float`). You cannot, on the other hand, trace a path from `long` to `short`, so it is not legal to assign a `long` value to a `short` variable.

Figure 4.3 is easy to memorize. The figure consists mostly of the numeric data types in order of size. The only extra piece of information is `char`, but that goes in the only place it could go: a 16-bit `char` "fits inside" a 32-bit `int`. (Note that you can't convert a `byte` to a `char` or a `char` to a `short`, even though it seems reasonable to do so.)

Any conversion between primitive types that is not represented by a path of arrows in Figure 4.3 is a narrowing conversion. These conversions lose information about the magnitude of the value being converted and are not allowed in assignments. It is graphically impossible to portray the narrowing conversions in a figure like Figure 4.3, but they can be summarized as follows:

- From a `byte` to a `char`
- From a `short` to a `byte` or a `char`
- From a `char` to a `byte` or a `short`
- From an `int` to a `byte`, a `short`, or a `char`
- From a `long` to a `byte`, a `short`, a `char`, or an `int`
- From a `float` to a `byte`, a `short`, a `char`, an `int`, or a `long`
- From a `double` to a `byte`, a `short`, a `char`, an `int`, a `long`, or a `float`

You do not really need to memorize this list. It simply represents all the conversions not shown in Figure 4.3, which is easier to memorize.

Assignment Conversion, Narrower Primitives, and Literal Values

Java's assignment conversion rule is sometimes inconvenient when a literal value is assigned to a primitive. By default, a numeric literal is either a `double` or an `int`, so the following line of code generates a compiler error:

```
float f = 1.234;
```

The literal value 1.234 is a `double`, so it cannot be assigned to a `float` variable.

You might assume that assigning a literal `int` to some narrower integral type (`byte`, `short`, or `char`) would fail to compile in a similar way. For example, it would be reasonable to assume that all of the following lines generate compiler errors:

```
byte  b = 1;
short s = 2;
char  c = 3;
```

In fact, all three of these lines compile without error. The reason is that Java relaxes its assignment conversion rule when a literal `int` value is assigned to a narrower primitive type (`byte`, `short`, or `char`), provided the literal value falls within the legal range of the primitive type.

This relaxation of the rule applies only when the assigned value is an integral literal. Thus the second line of the following code will *not* compile:

```
int i = 12;
byte b = I
```

Primitive Conversion: Method Call

Another kind of conversion is *method-call conversion*. A method-call conversion happens when you pass a value of one type as an argument to a method that expects a different type. For example, the `cos()` method of the `Math` class expects a single argument of type `double`. Consider the following code:

```
1. float frads;
2. double d;
3. frads = 2.34567f;
4. d = Math.cos(frads);  // Pass float to method
                         // that expects double
```

The `float` value in `frads` is automatically converted to a `double` value before it is handed to the `cos()` method. Just as with assignment conversions, strict rules govern which conversions are allowed and which conversions will be rejected by the compiler. The following code quite reasonably generates a compiler error (assuming there is a vector called `myVector`):

```
1. double d = 12.0;
2. Object ob = myVector.elementAt(d);
```

The compiler error message says, "Incompatible type for method. Explicit cast needed to convert double to int." This means the compiler can't convert the `double` argument to a type that is supported by a version of the `elementAt()` method. It turns out that the only version of `elementAt()` is the version that takes an integer argument. Thus a value can be passed to `elementAt()` only if that value is an `int` or can be converted to an `int`.

Fortunately, the rule that governs which method-call conversions are permitted is the same rule that governs assignment conversions. Widening conversions (as shown in Figure 4.3) are permitted; narrowing conversions are forbidden.

Primitive Conversion: Arithmetic Promotion

The last kind of primitive conversion to consider is *arithmetic promotion*. Arithmetic-promotion conversions happen within arithmetic statements while the compiler is trying to make sense out of many different possible kinds of operand.

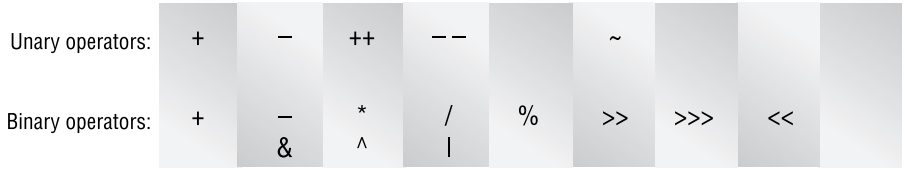
Consider the following fragment:

```
1. short s = 9;
2. int i = 10;
3. float f = 11.1f;
4. double d = 12.2;
5. if (-s * i  >=  f / d)
6.   System.out.println(i
7. else
8.   System.out.println(i<<<<i);
```

The code on line 5 multiplies a negated `short` by an `int`; then it divides a `float` by a `double`; finally, it compares the two results. Behind the scenes, the system is doing extensive type conversion to ensure that the operands can be meaningfully incremented, multiplied, divided, and compared. These conversions are all widening conversions. Thus they are known as *arithmetic-promotion conversions* because values are *promoted* to wider types.

The rules that govern arithmetic promotion distinguish between unary and binary operators. Unary operators operate on a single value. Binary operators operate on two values. Figure 4.4 shows Java's unary and binary arithmetic operators.

FIGURE 4.4 Unary and binary arithmetic operators



For unary operators, two rules apply, depending on the type of the single operand:

- If the operand is a `byte`, a `short`, or a `char`, it is converted to an `int` (unless the operator is ++ or --, in which case no conversion happens).
- Else there is no conversion.

For binary operators, there are four rules, depending on the types of the two operands:

- If one of the operands is a `double`, the other operand is converted to a `double`.
- Else if one of the operands is a `float`, the other operand is converted to a `float`.
- Else if one of the operands is a `long`, the other operand is converted to a `long`.
- Else both operands are converted to `int`s.

With these rules in mind, it is possible to determine what really happens in the code example given at the beginning of this section:

1. The `short s` is promoted to an `int` and then negated.
2. The result of step 1 (an `int`) is multiplied by the `int i`. Because both operands are of the same type, and that type is not narrower than an `int`, no conversion is necessary. The result of the multiplication is an `int`.
3. Before `float f` is divided by `double d`, `f` is widened to a `double`. The division generates a double-precision result.
4. The result of step 2 (an `int`) is to be compared to the result of step 3 (a `double`). The `int` is converted to a `double`, and the two operands are compared. The result of a comparison is always of type `boolean`.

Primitives and Casting

So far, this chapter has shown that Java is perfectly willing to perform widening conversions on primitives. These conversions are implicit and behind the scenes; you don't need to write any explicit code to make them happen.

Casting is explicitly telling Java to make a conversion. A casting operation may widen or narrow its argument. To cast, just precede a value with the parenthesized name of the desired type. For example, the following lines of code cast an `int` to a `double`:

```
1. int i = 5;
2. double d = (double)i;
```

Of course, the cast is not always necessary. The following code, in which the cast has been omitted, would do an assignment conversion on `i`, with the same result as the previous example:

```
1. int i = 5;
2. double d = i;
```

Casts are required when you want to perform a narrowing conversion. Such conversion will never be performed implicitly; you have to program an explicit cast to convince the compiler that what you really want is a narrowing conversion. Narrowing runs the risk of losing information; the cast tells the compiler that you accept the risk.

For example, the following code generates a compiler error:

```
1. short s = 259;
2. byte b = s;   // Compiler error
3. System.out.println("s = " + s + ", b = " + b);
```

The compiler error message for the second line will say (among other things), "Explicit cast needed to convert short to byte." Adding an explicit cast is easy:

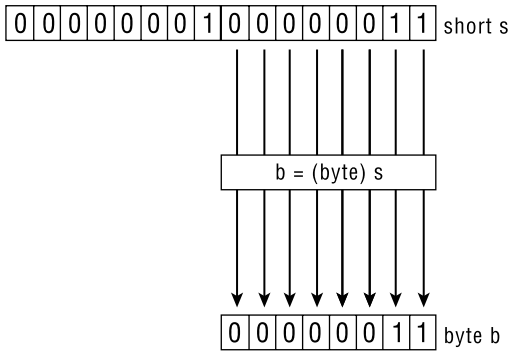
```
1. short s = 259;
2. byte b = (byte)s;     // Explicit cast
3. System.out.println(ib = î + b);
```

When this code is executed, the number 259 (binary 100000011) must be squeezed into a single `byte`. This is accomplished by preserving the low-order byte of the value and discarding the rest. The code prints out the (perhaps surprising) message:

```
b = 3
```

The 1 bit in bit position 8 is discarded, leaving only 3, as shown in Figure 4.5. Narrowing conversions can result in radical value changes; this is why the compiler requires you to cast explicitly. The cast tells the compiler, "Yes, I really want to do it."

FIGURE 4.5 Casting a short to a byte



Casting a value to a wider value (as shown in Figure 4.3) is always permitted but never required; if you omit the cast, an implicit conversion will be performed on your behalf. However, explicitly casting can make your code a bit more readable. For example:

```
  1. int i = 2;
  2. double radians;
     .      // Hundreds of
     .      // lines of
     .      // code
600. radians = (double)i;
```

The cast in the last line is not required, but it serves as a good reminder to any readers (including yourself) who might have forgotten the type of `radians`.

Two simple rules govern casting of primitive types:

- You can cast any non-`boolean` type to any other non-`boolean` type.
- You cannot cast a `boolean` to any other type; you cannot cast any other type to a `boolean`.

Note that although casting is ordinarily used when narrowing, it is perfectly legal to cast when widening. The cast is unnecessary, but provides a bit of clarity.

Real World Scenario

Legal and Illegal Casts

Write an application that illustrates legal and illegal casts. Work with the following class (and interface) hierarchy:

```
class Fruit class Apple extends Fruit interface Squeezable class Citrus extends
Fruit implements Squeezableclass Orange extends Citrus
```

Your application should construct one instance of each of the following classes:

- `Object`
- `Fruit`
- `Apple`
- `Citrus`
- `Orange`

Try to cast each of these objects to each of the following types:

- `Fruit`
- `Apple`
- `Squeezable`
- `Citrus`
- `Orange`

For each attempted cast, print out a message stating whether the attempted cast succeeded. (A `ClassCastException` is thrown if the cast failed; if no exception is thrown, the cast succeeded.) A fragment of the output of the sample solution looks like this:

```
Checking casts for Fruit        Fruit: OK
        Apple: NO        Squeezable: NO        Citrus: NO        Orange: NO
Checking casts for Apple       Fruit: OK       Apple: OK       Squeezable: NO
Citrus: NO        Orange: NO
```

Object Reference Conversion

Object reference variables, like primitive values, participate in assignment conversion, method-call conversion, and casting. (There is no arithmetic promotion of object references, because references cannot be arithmetic operands.) Object reference conversion is more complicated than primitive conversion, because there are more possible combinations of old and new types—and more combinations mean more rules.

Reference conversion, like primitive conversion, takes place at compile time, because the compiler has all the information it needs to determine whether the conversion is legal. Later you will see that this is not the case for object casting.

The following sections examine object reference assignment, method-call, and casting conversions.

Object Reference Assignment Conversion

Object reference assignment conversion happens when you assign an object reference value to a variable of a different type. There are three general kinds of object reference type:

- A class type, such as `Button` or `FileWriter`
- An interface type, such as `Cloneable` or `LayoutManager`
- An array type, such as `int[][]` or `TextArea[]`

Generally speaking, assignment conversion of a reference looks like this:

```
1. Oldtype x = new Oldtype();
2. Newtype y = x;   // reference assignment conversion
```

This is the general format of an assignment conversion from an `Oldtype` to a `Newtype`. Unfortunately, `Oldtype` can be a class, an interface, or an array; `Newtype` can also be a class, an interface, or an array. Thus there are nine (= 3 ∴ 3) possible combinations to consider. Figure 4.6 shows the rules for all nine cases.

FIGURE 4.6 The rules for object reference assignment conversion

Converting `Oldtype` to `Newtype`:

	`Oldtype` is a class	`Oldtype` is an interface	`Oldtype` is an array
`Newtype` is a class	`Oldtype` must be a subclass of `Newtype`	`Newtype` must be `Object`	`Newtype` must be `Object`
`Newtype` is an interface	`Oldtype` must implement interface `Newtype`	`Oldtype` must be a subinterface of `Newtype`	`Newtype` must be `Cloneable` or `Serializable`
`Newtype` is an array	Compiler error	Compiler error	`Oldtype` must be an array of some object reference type that can be converted to whatever `Newtype` is an array of

It would be difficult to memorize the nine rules shown in Figure 4.6. Fortunately, there is a rule of thumb.

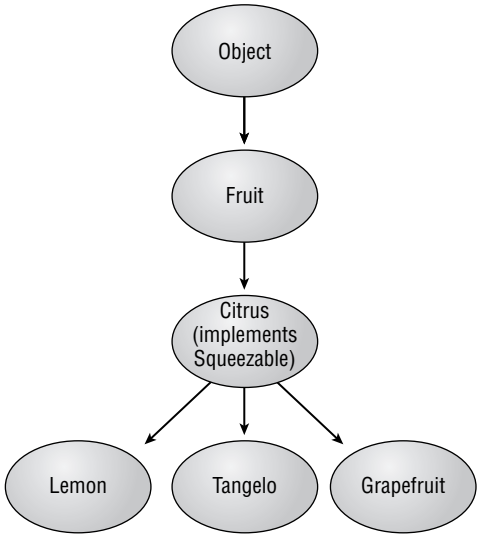
Recall that with primitives, conversions were permitted, provided they were widening conversions. The notion of widening does not really apply to references, but a similar principle is at work. In general, object reference conversion is permitted when the direction of the conversion is "up" the inheritance hierarchy; that is, the old type should inherit from the new type. This rule of thumb does not cover all nine cases, but it is a helpful way to look at things.

The rules for object reference conversion can be stated as follows:

- An interface type can be converted only to an interface type or to `Object`. If the new type is an interface, it must be a superinterface of the old type.
- A class type can be converted to a class type or to an interface type. If converting to a class type, the new type must be a superclass of the old type. If converting to an interface type, the old class must implement the interface.
- An array may be converted to the class `Object`, to the interface `Cloneable` or `Serializable`, or to an array. Only an array of object reference types can be converted to an array, and the old element type must be convertible to the new element type.

To illustrate these rules, consider the inheritance hierarchy shown in Figure 4.7 (assume there is an interface called `Squeezable`).

FIGURE 4.7 A simple class hierarchy



As a first example, consider the following code:

```
1. Tangelo tange = new Tangelo();
2. Citrus cit = tange;
```

This code is fine. A `Tangelo` is being converted to a `Citrus`. The new type is a superclass of the old type, so the conversion is allowed. Converting in the other direction ("down" the hierarchy tree) is not allowed:

```
1. Citrus cit = new Citrus();
2. Tangelo tange = cit;
```

This code will result in a compiler error.

What happens when one of the types is an interface?

```
1. Grapefruit g = new Grapefruit();
2. Squeezable squee = g;   // No problem
3. Grapefruit g2 = squee;  // Error
```

The second line ("No problem") changes a class type (`Grapefruit`) to an interface type. This is correct, provided `Grapefruit` really implements `Squeezable`. A glance at Figure 4.7 shows that this is indeed the case, because `Grapefruit` inherits from `Citrus`, which implements `Squeezable`. The third line is an error, because an interface can never be implicitly converted to any reference type other than `Object`.

Finally, consider an example with arrays:

```
1. Fruit fruits[];
2. Lemon lemons[];
3. Citrus citruses[] = new Citrus[10];
4. for (int i = 0; i < 10; i++) {
5.   citruses[i] = new Citrus();
6. }
7. fruits = citruses;    // No problem
8. lemons = citruses;    // Error
```

Line 7 converts an array of `Citrus` to an array of `Fruit`. This is fine, because `Fruit` is a superclass of `Citrus`. Line 8 converts in the other direction and fails, because `Lemon` is not a superclass of `Citrus`.

Object Method-Call Conversion

Fortunately, the rules for method-call conversion of object reference values are the same as the rules described earlier for assignment conversion of objects. The general rule of thumb is that converting to a superclass is permitted and converting to a subclass is not permitted.

To see how the rules make sense in the context of method calls, consider the extremely useful `java.lang.Vector` class. You can store anything you like in a vector (anything nonprimitive, that is) by calling the method `add (Object ob)`. For example, the following code stores a `Tangelo` in a vector:

```
1. Vector myVec = new Vector();
2. Tangelo tange = new Tangelo();
3. myVec.add (tange);
```

The `tange` argument will automatically be converted to type `Object`. The automatic conversion means that the people who wrote the `java.lang.Vector` class didn't have to write a separate

method for every possible type of object that anyone might conceivably want to store in a vector. This is fortunate: the `Tangelo` class was developed years after the invention of the vector, so the developer of the `Vector` class could not possibly have written specific `Tangelo`-handling code. An object of any class (and even an array of any type) can be passed into the single `add(Object ob)` method.

Object Reference Casting

Object reference casting is like primitive casting: by using a cast, you convince the compiler to let you do a conversion that otherwise might not be allowed.

Any kind of conversion that is allowed for assignments or method calls is allowed for explicit casting. For example, the following code is legal:

```
1. Lemon lem = new Lemon();
2. Citrus cit = (Citrus)lem;
```

The cast is legal but not needed; if you leave it out, the compiler will do an implicit assignment conversion. The power of casting appears when you explicitly cast to a type that is not allowed by the rules of implicit conversion.

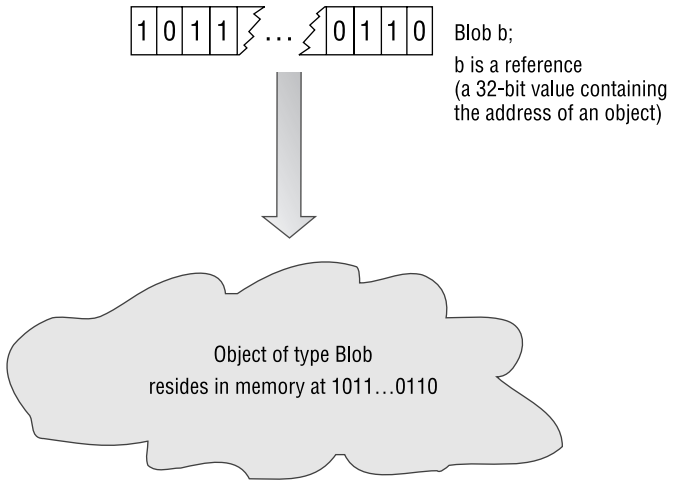
To understand how object casting works, it is important to understand the difference between objects and object reference variables. Every object (well, nearly every object—there are some obscure cases) is constructed via the `new` operator. The class name following `new` determines for all time the true class of the object. For example, if an object is constructed by calling `new Color(222, 0, 255)`, then throughout that object's lifetime, its class will be `Color`.

Java programs do not deal directly with objects. They deal with references to objects. For example, consider the following code:

```
Color purple = new Color(222, 0, 255);
```

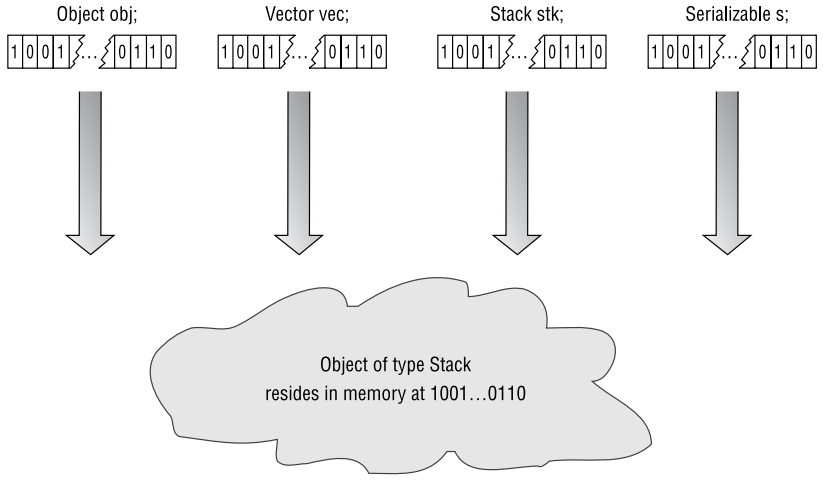
The variable `purple` is not an object; it is a reference to an object. The object itself lives in memory somewhere in the Java Virtual Machine (JVM). The variable `purple` contains something similar to the address of the object. This address is known as a *reference* to the object. The difference between a reference and an object is illustrated in Figure 4.8. References are stored in variables, and variables have types that are specified by the programmer at compile time. Object reference variable types can be classes (such as `Graphics` or `FileWriter`), interfaces (such as `Runnable` or `LayoutManager`), or arrays (such as `int[][]` or `Vector[]`).

FIGURE 4.8 Reference and object



Although an object's class is unchanging, it may be referenced by variables of many different types. For example, consider a stack. It is constructed by calling `new Stack()`, so its class really is `Stack`. Yet at various moments during the lifetime of this object, it may be referenced by variables of type `Stack` (of course), or of type `Vector` (because `Stack` inherits from `Vector`), or of type `Object` (because everything inherits from `Object`). It may even be referenced by variables of type `Serializable`, which is an interface, because the `Stack` class implements the `Serializable` interface. This situation is shown in Figure 4.9.

FIGURE 4.9 Many variable types, one class



The type of a reference variable is obvious at compile time. However, the class of an object referenced by such a variable cannot be known until runtime. This lack of knowledge is not a

shortcoming of Java technology; it results from a fundamental principle of computer science. The distinction between compile-time knowledge and runtime knowledge was not relevant to our discussion of conversions; however, the difference becomes important with reference value casting. The rules for casting are a bit broader than those for conversion. Some of these rules concern reference type and can be enforced by the compiler at compile time; other rules concern object class and can be enforced only during execution.

Quite a few rules govern object casting because a large number of obscure cases must be covered. For the exam, the important rules to remember when casting from `Oldtype` to `Newtype` are as follows:

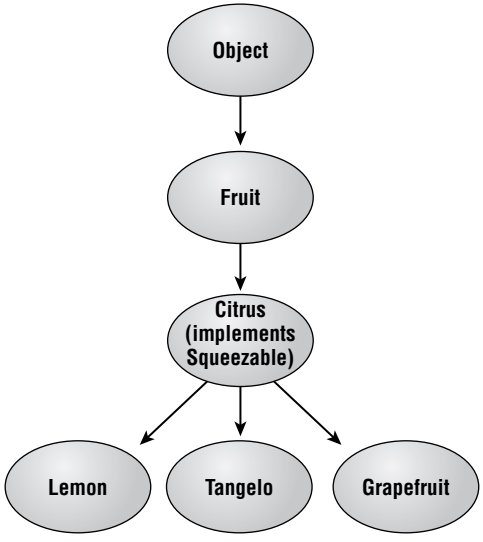
- When both `Oldtype` and `Newtype` are classes, one class must be a subclass of the other.
- When both `Oldtype` and `Newtype` are arrays, both arrays must contain reference types (not primitives), and it must be legal to cast an element of `Oldtype` to an element of `Newtype`.
- You can always cast between an interface and a nonfinal object.

Assuming that a desired cast survives compilation, a second check must occur at runtime. The second check determines whether the class of the object being cast is compatible with the new type. (This check could not be made at compile time, because the object being cast did not exist then.) Here, *compatible* means that the class can be converted according to the conversion rules discussed in the previous two sections. The following rules cover the most common runtime cases:

- If `Newtype` is a class, the class of the expression being converted must be `Newtype` or must inherit from `Newtype`.
- If `Newtype` is an interface, the class of the expression being converted must implement `Newtype`.

It is definitely time for some examples! Look once again at the `Fruit`/`Citrus` hierarchy that you saw earlier in this chapter, which is repeated in Figure 4.10.

FIGURE 4.10 Fruit hierarchy (reprise)



First, consider the following code:

```
1. Grapefruit g, g1;
2. Citrus c;
3. Tangelo t;
4. g = new Grapefruit(); // Class is Grapefruit
5. c = g;                // Legal assignment conversion,
                         // no cast needed
6. g1 = (Grapefruit)c;   // Legal cast
7. t = (Tangelo)c;       // Illegal cast
                         // (throws an exception)
```

This code has four references but only one object. The object's class is `Grapefruit`, because `Grapefruit`'s constructor is called on line 4. The assignment `c = g` on line 5 is a perfectly legal assignment conversion ("up" the inheritance hierarchy), so no explicit cast is required. In lines 6 and 7, the `Citrus` is cast to a `Grapefruit` and to a `Tangelo`. Recall that for casting between class types, one of the two classes (it doesn't matter which one) must be a subclass of the other. The first cast is from a `Citrus` to its subclass `Grapefruit`; the second cast is from a `Citrus` to its subclass `Tangelo`. Thus both casts are legal—at compile time. The compiler cannot determine the class of the object referenced by `c`, so it accepts both casts and lets fate determine the outcome at runtime.

When the code is executed, eventually the JVM attempts to execute line 6: `g1 = (Grapefruit)c;`. The class of `c` is determined to be `Grapefruit`, and there is no objection to converting a `Grapefruit` to a `Grapefruit`.

Line 7 attempts (at runtime) to cast `c` to type `Tangelo`. The class of `c` is still `Grapefruit`, and a `Grapefruit` cannot be cast to a `Tangelo`. In order for the cast to be legal, the class of `c` would have to be `Tangelo` itself or some subclass of `Tangelo`. Because this is not the case, a runtime exception (`java.lang.ClassCastException`) is thrown.

Now take an example where an object is cast to an interface type. Begin by considering the following code fragment:

```
1. Grapefruit g, g1;
2. Squeezable s;
3. g = new Grapefruit();
4. s = g;      // Convert Grapefruit to Squeezable (OK)
5. g1 = s;     // Convert Squeezable to Grapefruit
               // (Compile error)
```

This code will not compile. Line 5 attempts to convert an interface (`Squeezable`) to a class (`Grapefruit`). It doesn't matter that `Grapefruit` implements `Squeezable`. Implicitly converting an interface to a class is never allowed; it is one of those cases where you have to use

an explicit cast to tell the compiler that you really know what you're doing. With the cast, line 5 becomes

```
5. g1 = (Grapefruit)s;
```

Adding the cast makes the compiler happy. At runtime, the JVM checks whether the class of `s` (which is `Grapefruit`) can be converted to `Grapefruit`. It certainly can, so the cast is allowed.

For a final example, involving arrays, look at the following code:

```
1. Grapefruit g[];
2. Squeezable s[];
3. Citrus c[];
4. g = new Grapefruit[500];
5. s = g;           // Convert Grapefruit array to
                    // Squeezable array (OK)
6. c = (Citrus[])s; // Cast Squeezable array to Citrus
                    // array (OK)
```

Line 6 casts an array of `Squeezables` (`s`) to an array of `Citruses` (`c`). An array cast is legal if casting the array element types is legal (and if the element types are references, not primitives). In this example, the question is whether a `Squeezable` (the element type of array `s`) can be cast to a `Citrus` (the element type of the cast array). The previous example showed that this is a legal cast.

Summary

Primitive values and object references are very different kinds of data. Both can be converted (implicitly) or cast (explicitly). Primitive type changes are caused by assignment conversion, method-call conversion, arithmetic-promotion conversion, or explicit casting.

Primitives can be converted only if the conversion widens the data. Primitives can be narrowed by casting, as long as neither the old nor the new type is `boolean`.

Object references can be converted or cast; the rules that govern these activities are extensive because many combinations of cases must be covered. In general, going "up" the inheritance tree can be accomplished implicitly through conversion; going "down" the tree requires explicit casting. Object reference type changes are caused by assignment conversion, method-call conversion, or explicit casting.

Exam Essentials

Understand when primitive conversion takes place. Assignment and method-call conversion take place when the new data type is the same as or wider than the old type. Type widths are summarized in Figure 4.3.

Understand when arithmetic promotion takes place. You should know the type of result of unary and binary arithmetic operations performed on operands of any type.

Understand when primitive casting is required. Casting is required when the new data type is neither the same as nor wider than the old type.

Understand when object reference conversion takes place. The rules are summarized in Figure 4.6. The most common case is when the new type is a parent class of the old type.

Understand when object reference casting is required. The most common case is when the new type inherits from the old type.

Key Terms

Before you take the exam, be certain you are familiar with the following terms:

arithmetic promotion

arithmetic-promotion conversion

assignment conversion

casting

conversion

method-call conversion

narrowing conversion

reference

widening conversion

Review Questions

1. Which of the following statements is correct? (Choose one.)
   - **A.** Only primitives are converted automatically; to change the type of an object reference, you have to do a cast.
   - **B.** Only object references are converted automatically; to change the type of a primitive, you have to do a cast.
   - **C.** Arithmetic promotion of object references requires explicit casting.
   - **D.** Both primitives and object references can be both converted and cast.
   - **E.** Casting of numeric types may require a runtime check.

2. Which one line in the following code will not compile?
   - **A.** `byte b = 5;`
   - **B.** `char c = '5';`
   - **C.** `short s = 55;`
   - **D.** `int i = 555;`
   - **E.** `float f = 555.5f;`
   - **F.** `b = s;`
   - **G.** `i = c;`
   - **H.** `if (f > b)`
   - **I.** `f = i;`

3. Will the following code compile?

```
1. byte b = 2;
2. byte b1 = 3;
3. b = b * b1;
```

   - **A.** Yes
   - **B.** No

4. In the following code, what are the possible types for variable `result`? (Choose the most complete true answer.)

```
1. byte b = 11;
2. short s = 13;
```

```
3. result = b * ++s;
```

A. `byte`, `short`, `int`, `long`, `float`, `double`

B. `boolean`, `byte`, `short`, `char`, `int`, `long`, `float`, `double`

C. `byte`, `short`, `char`, `int`, `long`, `float`, `double`

D. `byte`, `short`, `char`

E. `int`, `long`, `float`, `double`

5. Consider the following class:

```
1.  class Cruncher {
2.    void crunch(int i) {
3.      System.out.println(ìint versionî);
4.    }
5.    void crunch(String s) {
6.      System.out.println(ìString versionî);
7.    }
8.
9.    public static void main(String args[]) {
10.     Cruncher crun = new Cruncher();
11.     char ch = 'p';
12.     crun.crunch(ch);
13.   }
14. }
```

Which of the following statements is true? (Choose one.)

A. Line 5 will not compile, because `void` methods cannot be overridden.

B. Line 12 will not compile, because no version of `crunch()` takes a `char` argument.

C. The code will compile but will throw an exception at line 12.

D. The code will compile and produce the following output: `int version`.

E. The code will compile and produce the following output: `String version`.

6. Which of the following statements is true? (Choose one.)

A. Object references can be converted in assignments but not in method calls.

B. Object references can be converted in method calls but not in assignments.

C. Object references can be converted in both method calls and assignments, but the rules governing these conversions are very different.

D. Object references can be converted in both method calls and assignments, and the rules governing these conversions are identical.

E. Object references can never be converted.

7. Consider the following code. Which line will not compile?

   A. `Object ob = new Object();`

   B. `String stringarr[] = new String[50];`

   C. `Float floater = new Float(3.14f);`

   D.

   E. `ob = stringarr;`

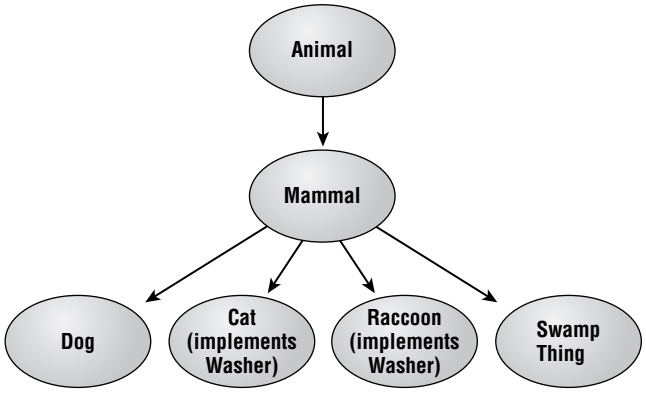
   F. `ob = stringarr[5];`

   G. `floater = ob;`

   H. `ob = floater;`

8. Questions 8–10 refer to the class hierarchy shown in Figure 4.11.

FIGURE 4.11 Class hierarchy for questions 8, 9, and 10



Consider the following code:

```
1. Dog      rover, fido;
2. Animal   anim;
3.
4. rover = new Dog();
5. anim = rover;
6. fido = (Dog)anim;
```

Which of the following statements is true? (Choose one.)

A. Line 5 will not compile.

B. Line 6 will not compile.

C. The code will compile but will throw an exception at line 6.

D. The code will compile and run.

E. The code will compile and run, but the cast in line 6 is not required and can be eliminated.

9. Consider the following code:

```
1. Cat sunflower;
2. Washer wawa;
3. SwampThing pogo;
4.
5. sunflower = new Cat();
6. wawa = sunflower;
7. pogo = (SwampThing)wawa;
```

Which of the following statements is true? (Choose one.)

A. Line 6 will not compile; an explicit cast is required to convert a `Cat` to a `Washer`.

B. Line 7 will not compile, because you cannot cast an interface to a class.

C. The code will compile and run, but the cast in line 7 is not required and can be eliminated.

D. The code will compile but will throw an exception at line 7, because runtime conversion from an interface to a class is not permitted.

E. The code will compile but will throw an exception at line 7, because the runtime class of `wawa` cannot be converted to type `SwampThing`.

10. Consider the following code:

```
1. Raccoon rocky;
2. SwampThing pogo;
3. Washer w;
4.
5. rocky = new Raccoon();
6. w = rocky;
7. pogo = w;
```

Which of the following statements is true? (Choose one.)

A. Line 6 will not compile; an explicit cast is required to convert a `Raccoon` to a `Washer`.

B. Line 7 will not compile; an explicit cast is required to convert a `Washer` to a `SwampThing`.

C. The code will compile and run.

D. The code will compile but will throw an exception at line 7, because runtime conversion from an interface to a class is not permitted.

E. The code will compile but will throw an exception at line 7, because the runtime class of `w` cannot be converted to type `SwampThing`.

Answers to Review Questions

1. D. D is correct because in Java primitives and object references can be both converted and cast. A and B are wrong because they contradict D. C is wrong because objects do not take part in arithmetic operations. E is wrong because only casting of object references potentially requires a runtime check.

2. F. The code `b = s` will not compile, because converting a `short` to a `byte` is a narrowing conversion, which requires an explicit cast. The other assignments in the code are widening conversions.

3. B. Surprisingly, the code will fail to compile at line 3. The two operands, which are originally `byte`s, are converted to `int`s before the multiplication. The result of the multiplication is an `int`, which cannot be assigned to `byte b`.

4. E. The result of the calculation on line 2 is an `int` (because all arithmetic results are `int`s or wider). An `int` can be assigned to an `int`, `long`, `float`, or `double`.

5. D. At line 12, the `char` argument `ch` is widened to type `int` (a method-call conversion) and passed to the `int` version of method `crunch()`.

6. D. Method-call and assignment conversions are governed by the same rules concerning the legal relationships between the old and new types.

7. G. Changing an `Object` to a `Float` is going "down" the inheritance hierarchy tree, so an explicit cast is required.

8. D. The code will compile and run. The cast in line 6 is required, because changing an `Animal` to a `Dog` is going "down" the tree.

9. E. The cast in line 7 is required. Answer D is a preposterous statement expressed in a tone of authority.

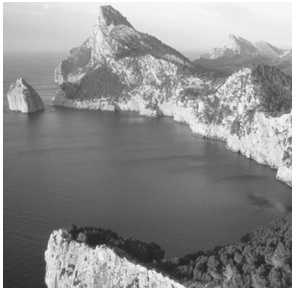
10. B. The conversion in line 6 is fine (class to interface), but the conversion in line 7 (interface to class) is not allowed. A cast in line 7 will make the code compile, but then at runtime a `ClassCastException` will be thrown, because `Washer` and `SwampThing` are incompatible.

Chapter 5

Flow Control, Assertions, and Exception Handling

JAVA CERTIFICATION EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- ✓ **2.1 Write code using `if` and `switch` statements, and identify legal argument types for these statements.**
- ✓ **2.2 Write code using all forms of loops including labeled and unlabeled use of `break` and `continue`, and state the values taken by loop counter variables during and after loop execution.**
- ✓ **2.3 Write code that makes proper use of exceptions and exception handling clauses (`try`, `catch`, `finally`) and declares methods and overriding methods that throw exceptions.**
- ✓ **2.4 Recognize the effect of an exception arising at a specified point in a code fragment. Note: The exception may be a runtime exception, a checked exception, or an error (the code may include `try`, `catch`, or `finally` clauses in any legitimate combination).**
- ✓ **2.5 Write code that makes proper use of assertions, and distinguish appropriate from inappropriate uses of assertions.**
- ✓ **2.6 Identify correct statements about the assertion mechanism.**



Flow control is a fundamental facility of almost any programming language. Sequence, iteration, and selection are the major elements of flow control, and Java provides these in forms that are familiar to C and C++ programmers. Additionally, Java provides for exception handling.

Sequence control is provided simply by the specification that, within a single block of code, execution starts at the top and proceeds toward the bottom. Iteration is catered for by three styles of loop: the `for()`, `while()`, and `do` constructions. Selection occurs when either the `if()/else` or `switch()` construct is used.

Java omits one common element of flow control: the idea of a `goto` statement. When Java was being designed, the team responsible did some analysis of a large body of existing code and determined that the use of `goto` was appropriate in two situations in new code: when code breaks out of nested loops and during the handling of exception conditions or errors. The designers left out `goto` and, in its place, provided alternative constructions to handle these particular conditions. The `break` and `continue` statements that control the execution of loops were extended to handle nested loops, and formalized exception handling was introduced, using ideas similar to those of C++.

This chapter discusses the flow-control facilities of Java. We will look closely at the exception mechanism, because this area commonly causes some confusion. But first, we will discuss the loop mechanisms.

The Loop Constructs

Java provides three loop constructions. Taken from C and C++, these are the `while()`, `do`, and `for()` constructs. Each provides the facility for repeating the execution of a block of code until some condition occurs. We will discuss the `while()` loop, which is perhaps the simplest, first.

The *while()* Loop

The general form of the `while()` loop is

```
1. while (boolean_condition)
2.   repeated_statement_or_block
```

In such a construct, the element *`boolean_condition`* must be an expression that returns a `boolean` result. Notice that this differs from C and C++, where a variety of types may be used: In Java you can *only* use a `boolean` expression. Typically, you might use a comparison of some kind, such as `x > 5`.

The *repeated_statement_or_block* will be executed again and again as long as the *boolean_condition* is true. If the condition never becomes false, then the loop will repeat forever. In practice, this really means that the loop will repeat until the program is stopped or the machine is turned off.

Notice that we've described the loop body as a "repeated statement or block." We need to make two important points here. The first is one of coding style, and as such is not directly related to the Programmer's Exam (although it might be relevant to the Developer's Exam when you take that). The second is the strict interpretation of the language specification, and as such might be needed in the Programmer's Exam. The two issues are related, so we will discuss them over the next few paragraphs.

The first point is that you would be well advised always to write a block to contain the code for the body of a loop or an `if()` statement. That is, always use a pair of braces so your code will look like this:

```
1. while (boolean_condition) {
2.   statement(s);
3. }
```

You should do so even where the loop contains only a single statement. The reason is that in many situations, you will change from a single line to multiple lines, and if the braces are in position already, that is one less thing to forget. One typical situation where this arises is when you add debug output to the middle of a loop to see how many times the loop is executed. It's very frustrating to realize after 20 minutes of messing about that the loop was executed 10 times, although the message was printed only on exit from the loop. It's perhaps worse to see the message printed 10 times but to have moved the proper body of the loop outside of it entirely.

The second point is that, from the position of strict correctness, you need to know that a single statement without braces is allowed in loops and `if` statements. So, the following code is correct and prints "five times" five times, but "once" only once:

```
1.  int i = 0;
2.  while (i++ < 5)
3.    System.out.println("five times");
4.  System.out.println("once");
```

It is highly unlikely that you will be presented with code that uses a single nonblocked statement as the body of a loop or the conditional part of an `if` statement, but if you do, you need to recognize how it will behave, and that it is not incorrect.

The exact position of the opening curly brace that marks a block of code is a matter of near-religious contention. Some programmers put it at the end of a line, as in most of the examples in this book. Others put it on a line by itself. Provided it is otherwise placed in the correct sequence, it does not matter how many space, tab, and newline characters are placed before or after the opening curly brace. In other words, this positioning is not relevant to syntactic correctness. You should be aware, however, that the style used in presenting the exam questions, as well as that used for the code in the developer-level exam, is the style shown here, where the opening brace is placed at the end of the line.

Observe that if the *boolean_condition* is already false when the loop is first encountered, then the body of the loop will never be executed. This fact relates to the main distinguishing feature of the do loop, which we will discuss next.

The *do* Loop

The general form of the do loop is

```
1. do
2.   repeated_statement_or_block
3. while (boolean_condition);
```

It is similar to the while() loop just discussed, and as before, it is best to have a loop body formed with a block:

```
1. do {
2.   do_something
3.   do_more
4. } while (boolean_condition);
```

Again, repetition of the loop is terminated when the *boolean_condition* becomes false. The significant difference is that this loop always executes the body of the loop at least once, because the test is performed at the end of the body.

Notice that the do loop (as opposed to the while loop) is guaranteed to run at least once, regardless of the value of the conditional expression. The do loop is probably used less frequently than the while() loop, but the third loop format is perhaps the most common. The third form is the for() loop, which we will discuss next.

The *for()* Loop

A common requirement in programming is to perform a loop so that a single variable is incremented over a range of values between two limits. This ability is frequently provided by a loop that uses the keyword for. Java's while() loop can achieve this effect, but it is most commonly achieved using the for() loop. However, as with C and C++, using the for() loop is more general than simply providing for iteration over a sequence of values.

The general form of the for() loop is

```
1. for (statement ; condition ; expression)
2.   loop_body
```

Again, a block should normally be used as the *loop_body* part, like this:

```
1. for (statement ; condition ; expression) {
2.   do_something
3.   do_more
4. }
```

The keys to this loop are in the three parts contained in the brackets following the `for` keyword:

- The *statement* is executed immediately before the loop itself is started. It is often used to set up starting conditions. You will see shortly that it can also contain variable declarations.
- The *condition* must be a `boolean` expression and is treated exactly the same as in the `while()` loop. The body of the loop will be executed repeatedly until the condition ceases to be true. As with the `while()` loop, it is possible that the body of a `for()` loop might never be executed. This occurs if the condition is already false at the start of the loop.
- The *expression* (short for "iteration expression") is executed immediately after the body of the loop, just before the test is performed again. Commonly, it is used to increment a loop counter.

If you have already declared an `int` variable `x`, you can code a simple sequence-counting loop like this:

```
1. for (x = 0; x < 10; x++) {
2.   System.out.println(ìvalue is ì + x);
3. }
```

This code would result in 10 lines of output, starting with

```
value is 0
```

and ending with

```
value is 9
```

In fact, because `for()` loops commonly need a counting variable, you are allowed to declare variables in the *statement* part. The scope of such a variable is restricted to the statement or block following the `for()` statement and the `for()` part itself. This limitation protects loop counter variables from interfering with each other and prevents leftover loop count values from accidental re-use. The result is code like this:

```
1. for (int x = 0; x < 10; x++) {
2.   System.out.println(ìvalue is ì + x);
3. }
```

It might be useful to look at the equivalent of this code implemented using a `while()` loop:

```
1. {
2.   int x = 0;
3.   while (x < 10) {
4.     System.out.println(ìvalue is ì + x);
5.     x++;
6.   }
7. }
```

This version reinforces a couple of points. First, the scope of the variable `x`, declared in the *`statement`* part of the `for()` loop, is restricted to the loop and its control parts (that is, the *`statement`*, *`condition`*, and *`expression`*). Second, the *`expression`* is executed after the rest of the loop body, effectively before control comes back to the test condition.

Empty *for()* Loops

Any part of a `for()` loop's control may be omitted if you wish. Omitting the test is equivalent to a perpetually true test, so the construct

```
for(;;) {}
```

creates a loop that repeats forever. Notice that both semicolons must still be included for correct syntax, even though the statement, condition, and expression are omitted.

The *for()* Loop and the Comma Separator

The `for()` loop allows the use of the comma separator in a special way. The *`statement`* and *`expression`* parts described previously can contain a sequence of expressions rather than just a single one. If you want such a sequence, you should separate those expressions, not with a semicolon (which would be mistaken as the separator between the three parts of the `for()` loop control structure), but with a comma. This behavior is borrowed from C and C++, where the comma is an operator; in Java the comma serves only as a special case separator for conditions where the semicolon would be unsuitable. This example demonstrates:

```
1. int j, k;
2. for (j = 3, k = 6; j + k < 20; j++, k +=2) {
3.   System.out.println("j is " + j + " k is " + k);
4. }
```

Note that although you can use the comma to separate several expressions, you cannot mix expressions with variable declarations, nor can you have multiple declarations of different types. So these would be illegal:

```
1. int i = 7;
2. for (i++, int j = 0; i < 10; j++) { } // illegal!


1. for (int i = 7, long j = 0; i < 10; j++) { } // illegal!
```

A final note on this issue is that the use of the comma to separate multiple declarations of a single type is allowed, like this:

```
1. for (int i = 7, j = 0; i < 10; j++) { }
```

This line declares two `int` variables, `i` and `j`, and initializes them to 7 and 0 respectively. This, however, is a standard feature of declarations and is not specific to the `for()` loop.

We have now discussed the three loop constructions in their basic forms. The next section looks at more advanced flow control in loops, specifically the use of the `break` and `continue` statements.

The *break* and *continue* Statements in Loops

Sometimes you need to abandon execution of the body of a loop—or perhaps a number of nested loops. The Java development team recognized this situation as a legitimate use for a `goto` statement. Java provides two statements, `break` and `continue`, which can be used instead of goto to achieve this effect.

Using *continue*

Suppose you have a loop that is processing an array of items that each contain two `String` references. The first `String` is always non-`null`, but the second might not be present. To process this, you might decide that you want, in pseudocode, something along these lines:

```
for each element of the array
  process the first String
  if the second String exists
    process the second String
  endif
endfor
```

You will recognize that this can be coded easily by using an `if` block to control processing of the second `String`. However, you can also use the `continue` statement like this:

```
1. for (int i = 0; i < array.length; i++) {
2.   // process first string
3.   if (array[i].secondString == null) {
4.     continue;
5.   }
6.   // process second string
7. }
```

In this case, the example is sufficiently simple that you probably do not see any advantage over using the `if()` condition to control the execution of the second part. If the second `String` processing was long, and perhaps heavily indented in its own right, you might find that the use of `continue` was slightly simpler visually.

The real strength of `continue` is that it is able to skip out of multiple levels of loop. Suppose the example, instead of being two `String` objects, is two-dimensional arrays of `char` values. Now you will need to nest your loops. Consider this sample:

```
1. mainLoop: for (int i = 0; i < array.length; i++) {
2.   for (int j = 0; j < array[i].length; j++) {
3.     if (array[i][j] == '\u0000') {
4.       continue mainLoop;
5.     }
6.   }
7. }
```

Notice particularly the label `mainLoop` that has been applied to the `for()` on line 1. The fact that this is a label is indicated by the trailing colon. You typically apply labels of this form to the opening loop statements: `while()`, `do`, or `for()`.

Here, when the processing of the second array comes across a 0 value, it abandons the whole processing not just for the inner loop, but for the current object in the main array. This is equivalent to jumping to the statement `i++` in the first `for()` statement.

You might still think this is not really any advantage over using `if()` statements, but imagine that further processing was done between lines 6 and 7, and that finding the 0 character in the array was required to avoid that further processing. To achieve that without using `continue`, you would have to set a flag in the inner loop and use it to abandon the outer loop processing. It can be done, but it is rather messy.

Using *break*

The `break` statement, when applied to a loop, is somewhat similar to the `continue` statement. However, instead of prematurely completing the current iteration of a loop, `break` causes the entire loop to be abandoned. Consider this example:

```
1. for (int j = 0; j < array.length; j++) {
2.   if (array[j] == null) {
3.     break; //break out of inner loop
4.   }
5.   // process array[j]
6. }
```

In this case, instead of simply skipping some processing for `array[j]` and proceeding directly to processing `array[j+1]`, this version quits the entire inner loop as soon as a `null` element is found.

You can also use labels on `break` statements, and as before, you must place a matching label on one of the enclosing blocks. The `break` and `continue` statements provide a convenient way to make parts of a loop conditional, especially when used in their labeled formats.

In fact, labels may be applied to any statements, and `break` can be used to jump out of any labeled block, whether that block is the body of a loop statement or not. Thus the `break` statement is a close imitation of a full-fledged `goto`. In the Certification Exam, you will not be expected to use `break` for any purpose other than jumping out of loops. In daily programming, you should probably avoid using it for any other purpose, too.

The next section discusses the `if()/else` and `switch()` constructions, which provide the normal means of implementing conditional code.

The Selection Statements

Java provides a choice of two selection constructs: the `if()/else` and `switch()` mechanisms. You can easily write simple conditional code for a choice of two execution paths based on the value of a `boolean` expression using `if()/else`. If you need more complex choices between multiple execution paths, and if an integral argument is available to control the choice, then you can use `switch()`; otherwise you can use either nests or sequences of `if()/else`.

The *if()/else* Construct

The `if()/else` construct takes a `boolean` argument as the basis of its choice. Often you will use a comparison expression to provide this argument. For example:

```
1. if (x > 5) {
2.   System.out.println("x is more than 5");
3. }
```

This sample executes line 2, provided the test `(x > 5)` in line 1 returns true. Notice that we used a block even though there is only a single conditional line, just as we suggested you should generally do with the loops discussed earlier.

You can use an `else` block to give code that is executed under the conditions that the test returns false. For example:

```
1. if (x > 5) {
2.   System.out.println("x is more than 5");
3. }
4. else {
5.   System.out.println("x is not more than 5");
6. }
```

You can also use `if()/else` in a nested fashion, refining conditions to more specific, or narrower, tests at each point.

The `if()/else` construction makes a test between only two possible paths of execution. However, you can also use the `if()/else` construction to choose between multiple possible execution paths by using the `if()/else if()` varation of the construction. For example:

```
1. if (hours > 1700) {
2.   System.out.println("good evening");
3. }
4. else if (hours > 1200){
```

```
5.   System.out.println("good afternoon");
6. }
7. else {
8.   System.out.println("good morning");
9. }
```

The code snippet above can be rewritten using the `switch()` statement. The next section discusses the `switch()` statement, which allows a single value to select between multiple possible execution paths.

The *switch()* Construct

If you need to make a choice between multiple alternative execution paths, and the choice can be based upon an `int` value, you can use the `switch()` construct. Consider this example:

```
 1. switch (x) {
 2.   case 1:
 3.     System.out.println("Got a 1");
 4.     break;
 5.   case 2:
 6.   case 3:
 7.     System.out.println("Got 2 or 3");
 8.     break;
 9.   default:
10.     System.out.println("Not a 1, 2, or 3");
11.     break;
12. }
```

Note that, although you cannot determine the fact by inspection of this code, the variable `x` must be either `byte`, `short`, `char`, or `int`. It must not be `long`, either of the floating-point types, `boolean`, or an object reference. Strictly, the value must be "assignment compatible" with `int`.

The comparison of values following `case` labels with the value of the expression supplied as an argument to `switch()` determines the execution path. The arguments to `case` labels must be constants, or at least constant expressions that can be fully evaluated at compile time. You cannot use a variable or an expression involving variables.

Each `case` label takes only a single argument, but when execution jumps to one of these labels, it continues downward until it reaches a `break` statement. This occurs even if execution passes another `case` label or the `default` label. So, in the previous example, if `x` has the value 2, execution goes through lines 1, 5, 6, 7, and 8, and continues beyond line 12. This requirement for `break` to indicate the completion of the `case` part is important. More often than not, you do not want to omit the `break`, because you do not want execution to "fall through." However,

to achieve the effect shown in the example, where more than one particular value of `x` causes execution of the same block of code, you use multiple `case` labels with only a single `break`.

The `default` statement is comparable to the `else` part of an `if()/else` construction. Execution jumps to the `default` statement if none of the explicit `case` values matches the argument provided to `switch()`. Although the `default` statement is shown at the end of the `switch()` block in the example (and this is both a conventional and reasonably logical place to put it), no rule requires this placement.

Now that you have examined the constructions that provide for iteration and selection under normal program control, let's look at the flow of control under exception conditions—that is, conditions when some runtime problem has arisen.

Exceptions

Sometimes when a program is executing, something occurs that is not quite normal from the point of view of the goal at hand. For example, a user might enter an invalid filename; a file might contain corrupted data; a network link could fail; or a bug in the program might cause it to try to make an illegal memory access, such as referring to an element beyond the end of an array.

Circumstances of this type are called *exception* conditions in Java and are represented using objects. A subtree of the class hierarchy starting with the class `java.lang.Throwable` is dedicated to describing them.

The process of an exception "appearing" either from the immediate cause of the trouble, or because a method call is abandoned and passes the exception up to its caller, is called *throwing* an exception in Java. You will hear other terms used, particularly an exception being *raised*.

If you take no steps to deal with an exception, execution jumps to the end of the current method. The exception then appears in the caller of that method, and execution jumps to the end of the calling method. This process continues until execution reaches the "top" of the affected thread, at which point the thread dies.

Flow of Control in Exception Conditions

When you write your own exception-based code, and when you prepare for the exam, it is vital to understand exactly how control proceeds, whether or not an exception gets thrown. The following sections examine control in the common `try/catch` case, when a `finally` block is included, and when multiple exception handlers are provided.

Using *try{} catch() {}*

To intercept, and thereby control, an exception, you use a `try/catch/finally` construction. You place lines of code that are part of the normal processing sequence in a *try* block. You then

add code to deal with an exception that might arise during execution of the `try` block in a *catch* block. If multiple exception classes might arise in the `try` block, then several `catch` blocks are allowed to handle them. Code that must be executed no matter what happens can be placed in a `finally` block. Let's take a moment to consider an example:

```
 1. int x = (int)(Math.random() * 5);
 2. int y = (int)(Math.random() * 10);
 3. int [] z = new int[5];
 4. try {
 5.   System.out.println("y/x gives " + (y/x));
 6.   System.out.println("y is "es " + (y/x));
 7.    + y + " z[y] is " + z[y]);
 8. }
 9. catch (ArithmeticException e) {
10.   System.out.println("Arithmetic problem " + e);
11. }
12. catch (ArrayIndexOutOfBoundsException e) {
13.   System.out.println("Subscript problem " + e);
14. }
```

In this example, an exception is possible at line 5 and at line 6. Line 5 has the potential to cause a division by 0, which in integer arithmetic results in an `ArithmeticException` being thrown. Line 6 will sometimes throw an `ArrayIndexOutOfBoundsException`.

If the value of `x` happens to be 0, then line 5 will result in the construction of an instance of the `ArithmeticException` class that is then thrown. Execution continues at line 9, where the variable `e` takes on the reference to the newly created exception. At line 10, the message printed includes a description of the problem, which comes directly from the exception itself. A similar flow occurs if line 5 executes without a problem but the value of `y` is 5 or greater, causing an out-of-range subscript in line 6. In that case, execution jumps directly to line 12.

In either of these cases, where an exception is thrown in a `try` block and is caught by a matching `catch` block, the exception is considered to have been handled: Execution continues after the last `catch` block as if nothing had happened. If, however, no `catch` block names either the class of exception that has been thrown or a class of exception that is a parent class of the one that has been thrown, then the exception is considered to be unhandled. In such conditions, execution generally leaves the method directly, just as if no `try` had been used.

You cannot write a `catch` block for exceptions that would never be thrown in the `try` block. This will generate a compiler error.

Table 5.1 summarizes the flow of execution that occurs in the exception-handling scenarios discussed up to this point. You should not rely on this table for exam preparation, because it only describes the story so far.

TABLE 5.1 Outline of Flow in Simple Exception Conditions

Exception	*try {}*	Matching *catch() {}*	Behavior
No	N/A	N/A	Normal flow
Yes	No	N/A	Method terminates
Yes	Yes	No	Method terminates
Yes	Yes	Yes	1. Terminate `try {}` block
			2. Execute body of matching `catch` block
			3. Continue normal flow after `catch` blocks

Using *finally*

The generalized exception-handling code has one more part to it than you saw in the last example. This is the `finally` block. If you put a `finally` block after a `try` and its associated `catch` blocks, then once execution enters the `try` block, the code in that `finally` block will definitely be executed whatever the circumstances—well, nearly definitely. If an exception arises with a matching `catch` block, then the `finally` block is executed after the `catch` block. If no exception arises, the `finally` block is executed after the `try` block. If an exception arises for which there is no appropriate `catch` block, then the `finally` block is executed after the `try` block.

The circumstances that can prevent execution of the code in a `finally` block are

- An exception arising in the `finally` block itself
- The death of the thread
- The use of `System.exit()`
- Turning off the power to the CPU

Notice that an exception in the `finally` block behaves exactly like any other exception; it can be handled via a `try/catch`. If no `catch` is found, then control jumps out of the method from the point at which the exception is raised, perhaps leaving the `finally` block incompletely executed.

Catching Multiple Exceptions

When you define a `catch` block, that block will catch exceptions of the class specified, including any exceptions that are subclasses of the one specified. In this way, you can handle categories of exceptions in a single `catch` block. If you specify one exception class in one particular `catch` block and a parent class of that exception in another `catch` block, you can handle the more specific exceptions—those of the subclass—separately from others of the same general parent class. Under such conditions these rules apply:

- A more specific `catch` block must precede a more general one in the source. Failure to meet this ordering requirement causes a compiler error.
- Only one `catch` block, the first applicable one, will be executed.

Now let's look at the overall framework for `try`, multiple `catch` blocks, and `finally`:

```
 1. try {
 2.   // statements
 3.   // some are safe, some might throw an exception
 4. }
 5. catch (SpecificException e) {
 6.   // do something, perhaps try to recover
 7. }
 8. catch (OtherException e) {
 9.   // handling for OtherException
10. }
11. catch (GeneralException e) {
12.   // handling for GeneralException
13. }
14. finally {
15.   // code that must be executed under
16.   // successful or unsuccessful conditions.
17. }
18. // more lines of method code
```

In this example, `GeneralException` is a parent class of `Specific Exception`. Several scenarios can arise under these conditions:

- No exceptions occur.
- A `SpecificException` occurs.
- A `GeneralException` occurs.
- An entirely different exception occurs, which we will call an `UnknownException`.

If no exceptions occur, execution completes the `try` block, lines 1–4, and then proceeds to the `finally` block, lines 14–17. The rest of the method, line 18 onward, is then executed.

If a `SpecificException` occurs, execution abandons the `try` block at the point the exception is raised and jumps into the `SpecificException` `catch` block. Typically, this might result in lines 1 and 2, then 5, 6, and 7 being executed. After the `catch` block, the `finally` block and the rest of the method are executed, lines 14–17 and line 18 onward.

If a `GeneralException` that is not a `SpecificException` occurs, then execution proceeds out of the `try` block, into the `GeneralException` `catch` block at lines 11–13. After that `catch` block, execution proceeds to the `finally` block and the rest of the method, just as in the last example.

If an `UnknownException` occurs, execution proceeds out of the `try` block directly to the `finally` block. After the `finally` block is completed, the rest of the method is abandoned. This is an uncaught exception; it will appear in the caller just as if there had never been any `try` block in the first place.

Now that we have discussed what happens when an exception is thrown, let's proceed to how exceptions are thrown and the rules that relate to methods that might throw exceptions.

Throwing Exceptions

The last section discussed how exceptions modify the flow of execution in a Java program. We will continue by examining how exceptions are issued in the first place and how you can write methods that use exceptions to report difficulties.

The *throw* Statement

Throwing an exception, in its most basic form, is simple. You need to do two things. First, you create an instance of an object that is a subclass of `java.lang.Throwable`. Next you use the `throw` keyword to actually throw the exception. These two are normally combined into a single statement like this:

```
throw new IOException(îFile not foundî);
```

There is an important reason why the `throw` statement and the construction of the exception are normally combined. The exception builds information about the point at which it was created, and that information is shown in the stack trace when the exception is reported. It is convenient if the line reported as the origin of the exception is the same line as the `throw` statement, so it is a good idea to combine the two parts; `throw new` *`xxx`*`()` becomes the norm.

The *throws* Statement

You saw how easy it is to generate and throw an exception; however, the overall picture is more complex. First, as a general rule, Java requires that any method that might throw an exception must declare the fact. In a way, this is a form of enforced documentation, but you will see that there is a little more to it than that.

If you write a method that might throw an exception (and this includes unhandled exceptions that are generated by other methods called from your method), then you must declare the possibility using a `throws` statement. For example, the (incomplete) method shown here can throw a `MalformedURLException` or an `EOFException`:

```
 1. public void doSomeIO(String targetUrl)
 2.   throws MalformedURLException, EOFException {
 3.   // new URL might throw MalformedURLException
 4.   URL url = new URL(targetUrl);
 5.   // open the url and read from it...
 6.   // set flag 'completed' when IO is successful
 7.   //....
 8.   // so if we get here with completed == false,
 9.   // we got unexpected end of file.
10.   if (!completed) {
11.     throw new EOFException("Invalid file contents");
12.   }
13. }
```

Line 11 demonstrates the use of the `throw` statement—it is usual for a `throw` statement to be conditional in some way; otherwise the method has no way to complete successfully. Line 2 shows the use of the `throws` statement. In this case, two distinct exceptions are listed that the method might throw under different failure conditions. The exceptions are given as a comma-separated list.

The section "Catching Multiple Exceptions" earlier in this chapter explained that the class hierarchy of exceptions is significant in `catch` blocks. The hierarchy is also significant in the `throws` statement. In this example, line 2 could be shortened to `throws IOException`, because both `MalformedURLException` and `EOFException` are subclasses of `IOException`.

It is important to recognize that declaring that a method throws an exception does not mean the method will fail with that exception, only that it might do so. In fact, it is perfectly legitimate—and in some situations that you will see later, actually necessary—to make such declarations, even though they appear to be redundant.

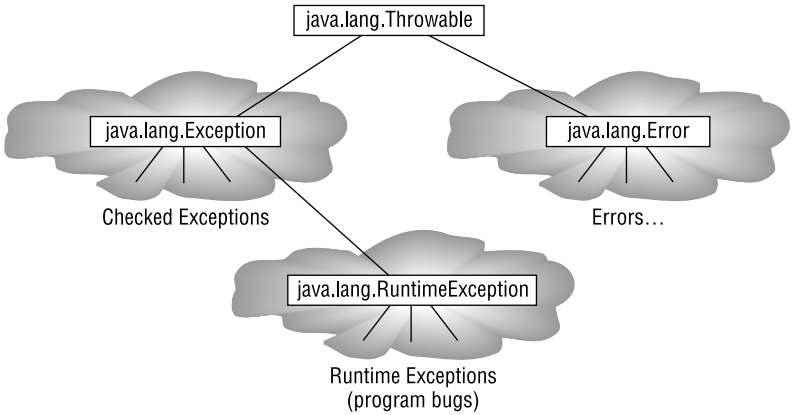
Checked Exceptions

So far we have discussed throwing exceptions and declaring methods that might throw exceptions. We have stated that any method that throws an exception should use the `throws` statement to declare the fact. The whole truth is slightly subtler.

The class hierarchy that exists under the class `java.lang.Throwable` is divided into three parts. One part contains the errors, which are `java. lang.Error` and all subclasses. Another part is called the runtime exceptions, which are `java.lang.RuntimeException` and all its subclasses. The third

part contains the checked exceptions, which are all subclasses of `java.lang.Exception` (except for `java.lang.RuntimeException` and its subclasses). Figure 5.1 shows this diagrammatically.

FIGURE 5.1 Categories of exceptions



You might well ask why the hierarchy is divided up and what these various names mean.

The *checked exceptions* describe problems that can arise in a correct program—typically, difficulties with the environment such as user mistakes or I/O problems. For example, attempting to open a socket can fail if the remote machine does not exist, is not responding, or is not providing the requested service. Neither of these problems indicates a programming error; it's more likely to be a problem with the machine name (the user mistyped it) or with the remote machine (perhaps it is incorrectly configured). Because these conditions can arise at any time, in a commercial-grade program you must write code to handle and recover from them. In fact, the Java compiler checks that you have indeed stated what is to be done when they arise, and because of this checking, they are called checked exceptions.

Runtime exceptions typically describe program bugs. You could use a runtime exception as deliberate flow control, but it would be an odd way to design code and rather poor style. Runtime exceptions generally arise from things like out-of-bounds array accesses, and normally a correctly coded program would avoid them. Because runtime exceptions should never arise in a correct program, you are not required to handle them. After all, it would only clutter your program if you had to write code that your design states should never be executed.

Errors generally describe problems that are sufficiently unusual and sufficiently difficult to recover from that you are not required to handle them. They might reflect a program bug, but more commonly they reflect environmental problems, such as running out of memory. As with runtime exceptions, Java does not require that you state how these are to be handled. Although errors behave just like exceptions, they typically should not be caught, because it is impossible to recover from them.

An approach to program design and implementation that is highly effective in producing robust and reliable code is known as *programming by contract*. Briefly, this approach requires clearly defined responsibilities for methods and the callers of those methods. For example, a square-root method could require that it must be called only with a non-negative argument. If called with a negative argument, the method would react by throwing an exception, because the contract between it and its caller has been broken. This approach simplifies code, because methods only attempt to handle properly formulated calls. It also brings bugs out into the open as quickly as possible, thereby ensuring that they get fixed. You should use runtime exceptions to implement this approach because it is clearly inappropriate for the caller to have to check for programming errors; the programmer should fix them.

Checking Checked Exceptions

We stated that of the three categories of exceptions, the checked exceptions make certain demands of the programmer: you are obliged to state how the exception is to be handled. In fact, you have two choices. You can put a `try` block around the code that might throw the exception and provide a corresponding `catch` block that will apply to the exception in question. Doing so handles the exception so it effectively goes away. Alternatively, you might decide that if this exception occurs, your method cannot proceed and should be abandoned. In this case, you do not need to provide the `try/catch` construction, but you must instead make sure the method declaration includes a `throws` part that informs potential callers that the exception might arise. Notice that by insisting the method be declared in this way, the responsibility for handling the exception is explicitly passed to the caller of the method, which must then make the same choice—whether to declare or handle the exception. The following example demonstrates this choice:

```
 1. public class DeclareOrHandle {
 2.   // This method makes no attempt to recover from the
 3.   // exception, rather it declares that it might
 4.   // throw it and uses no try block
 5.   public void declare(String s) throws IOException {
 6.     URL u = new URL(s); // might throw IOException
 7.     // do things with the URL object u...
 8.   }
 9.
10.   // This method handles the exception that might
11.   // arise when it calls the method declare().
12.   // Therefore, it does not throw any exceptions
13.   // and so does not use any throws declaration
14.   public void handle(String s) {
```

```
15.     boolean success = false;
16.     while (!success) {
17.       try {
18.         declare(s);  // might throw an IOException
19.         // execute this if declare() succeeded
20.         success = true;
21.       }
22.       catch (IOException e) {
23.         // Advise user that String s is somehow
24.         // unusable and ask for a new one
25.       }
26.     } // end while loop, exits when success is true
27.   }
28. }
```

The method `declare()` does not attempt to handle the exception that might arise during construction of the `URL` object. Instead, the `declare()` method states that it might throw the exception. By contrast, the `handle()` method uses a `try/catch` construction to ensure that control remains inside the `handle()` method itself until it becomes possible to recover from the problem.

We have now discussed the handling of exceptions and the constructions that allow you to throw exceptions of your own. Before we finish with exceptions, you must consider a rule relating to overriding methods and exceptions. The next section discusses this rule.

Exceptions and Overriding

When you extend a class and override a method, the Java compiler insists that all exception classes thrown by the new method must be the same as, or subclasses of, the exception classes thrown by the original method. Consider these examples (assume they are declared in separate source files; the line numbers are simply for reference):

```
 1. public class BaseClass {
 2.   public void method() throws IOException {
 3.   }
 4. }
 5.
 6. public class LegalOne extends BaseClass {
 7.   public void method() throws IOException {
 8.   }
 9. }
10.
11. public class LegalTwo extends BaseClass {
12.   public void method() {
13.   }
```

```
14. }
15.
16. public class LegalThree extends BaseClass {
17.   public void method()
18.   throws EOFException, MalformedURLException {
19.   }
20. }
21.
22. public class IllegalOne extends BaseClass {
23.   public void method()
24.   throws IOException, IllegalAccessException {
25.   }
26. }
27.
28. public class IllegalTwo extends BaseClass {
29.   public void method()
30.   throws Exception {
31.   }
32. }
```

Notice that the original `method()` in `BaseClass` is declared as throwing `IOException`. This declaration allows it, and any overriding method defined in a subclass, to throw an `IOException` or any object that is a subclass of `IOException`. Overriding methods cannot, however, throw any checked exceptions that are not subclasses of `IOException`.

Given these rules, you will see that line 7 in `LegalOne` is correct, because `method()` is declared exactly the same way as the original that it overrides. Similarly, line 18 in `LegalThree` is correct, because both `EOFException` and `MalformedURLException` are subclasses of `IOException`—so this adheres to the rule that nothing may be thrown that is not a subclass of the exceptions already declared. Line 12 in `LegalTwo` is correct, because it throws no exceptions and therefore cannot throw any exceptions that are not subclasses of `IOException`.

The methods at lines 23 and 29 are not permissible, because both of them throw checked exceptions that are not subclasses of `IOException`. In `IllegalOne`, `IllegalAccessException` is a subclass of `Exception`; in `IllegalTwo`, `Exception` itself is a superclass of `IOException`. Both `IllegalAccessException` and `Exception` are checked exceptions, so the methods that attempt to throw them are illegal as overriding methods of `method()` in `BaseClass`.

The point of this rule relates to the use of base class variables as references to objects of subclass type. Chapter 4, "Converting and Casting," explains that you can declare a variable of a class `X` and then use that variable to refer to any object that is of class `X` or any subclass of `X`.

Imagine that in the examples just described, you declared a variable `myBaseObject` of class `BaseClass`; you can use it to refer to objects of any of the classes `LegalOne`, `LegalTwo`, and `LegalThree`. (You can't use it to refer to objects of class `IllegalOne` or `IllegalTwo`, because those objects cannot be created in the first place: their code won't compile.) The compiler

imposes checks on how you call `myBaseObject.method()`. Those checks ensure that for each call, you have either enclosed the call in a `try` block and provided a corresponding `catch` block, or you have declared that the calling method itself might throw an `IOException`. Now suppose that at runtime, the variable `myBaseObject` was used to refer to an object of class `IllegalOne`. Under these conditions, the compiler would still believe that the only exceptions that must be dealt with are of class `IOException`, because it believes that `myBaseObject` refers to an object of class `BaseClass`. The compiler would therefore not insist that you provide a `try/catch` construct that catches the `IllegalAccessException`, nor that you declare the calling method as throwing that exception. Thus if the class `IllegalOne` were permitted, overriding methods would be able to bypass the enforced checks for checked exceptions.

It is important to consider the likely needs of subclasses whenever you define a class. Recall from the earlier section "The `throws` Statement" that it is entirely permissible to declare that a method throws an exception even if no code exists to actually throw that exception. Now that you know an overriding method cannot throw exceptions that were not declared in the parent method, you will recognize that some parent classes need to declare exceptions in methods that do not in fact throw any exceptions. For example, the `InputStream` class cannot, of itself, actually throw any exceptions, because it doesn't interact with real devices that could fail. However, it is used as the base class for a whole hierarchy of classes that do interact with physical devices: `FileInputStream` and so forth. It is important that the `read()` methods of those subclasses be able to throw exceptions, so the corresponding `read()` methods in the `InputStream` class itself must be declared as throwing `IOException`.

We have now looked at all the aspects of exception handling that you will need to prepare for the Certification Exam and to make effective use of exceptions in your programs.

Assertions

The Java 1.4 release includes a new facility called *assertion*. Assertions provide a convenient mechanism for verifying that a class's methods are called correctly. This mechanism can be enabled or disabled at runtime. The intention is that assertions typically will be enabled during development and disabled in the field.

The new `assert` keyword has the following syntax:

```
assert Expression1;
assert Expression1:Expression2;
```

`Expression1` must have `boolean` type. *`Expression2`* may have any type. If assertions are disabled at runtime (the default state), the `assert` statement does absolutely nothing. If assertions are enabled at runtime (via a command-line argument to the JVM), then *`Expression1`* is evaluated. If its value is true, no further action is taken. If its value is false, then an `AssertionError` is thrown. If *`Expression2`* is present, it is passed into the constructor of the `AssertionError`, where it is converted to a `String` and used as the error's message.

Assertions and Compilation

Sun is generally reluctant to expand the Java language, and with good reason. Unbridled language expansion would compromise Java's simplicity, and could also create compatibility problems with existing code.

For example, the introduction of the `assert` keyword is inconvenient for developers who have used `assert` as an identifier in pre-1.4 code (perhaps to implement their own home-grown assertion facility). Thus it was necessary to introduce a compiler flag to control whether `assert` should be treated as an identifier or as a keyword. To treat it as a keyword (that is, to take advantage of the new facility), compile with `-source 1.4` as in the following example:

```
javac -source 1.4 UsefulApplication.java
```

If the flag is omitted, the 1.4 compiler treats source code as if the `assert` keyword did not exist; thus `assert` can be used as an identifier.

Runtime Enabling of Assertions

Assertions are disabled by default. To enable assertions at runtime, use the `-enableassertions` or `-ea` flag on the Java command line as in the following example:

```
java -ea UsefulApplication
```

Additional runtime flags enable or disable assertions at the class level, but they are beyond the scope of the exam and of this book. If assertions are disabled, `assert` statements have no effect.

The `-ea` flag means that code can be developed with heavy reliance on assertions for debugging. The code can be shipped without removing the `assert` statements; they will have negligible effect on performance in the field, where the code is run with assertions disabled. This functionality demonstrates the benefit of incorporating assertions into the Java language. Assertion support could certainly have been built from scratch based on pre-1.4 platforms (and indeed it probably has been, in many different forms at many different sites). However, it would be difficult to field-disable the facility. The new Java 1.4 functionality ensures that assertions are simple and consistent.

Using Assertions

Assertions are commonly used to check *preconditions*, *postconditions*, and *class invariants*. Before going further, we should define these terms.

A precondition is a constraint that must be met on entry of a method. If a method's preconditions are not met, the method should terminate at once before it can do any damage. A method's preconditions are typically functions of its arguments and the state of its object. Argument range checking at the start of a method is a common form of precondition testing.

A postcondition is a constraint that must be met on return from a method. If a method's postconditions are not met, the method should not be allowed to return. A method's postconditions

are typically functions of its return value and the state of its object. In a general sense, if a precondition fails, the problem lies in the method's caller, whereas if a postcondition fails, the problem lies in the method itself.

A class invariant is a constraint on a class's state that must be met before and after execution of any non-private method of a class. (Private methods might be used to restore the required state after execution of a non-private method.)

To see how assertions can be used to enforce pre- and postconditions, imagine a class called `Library` that models a library (not a software library, but the kind where you can borrow books). Such a class might have a method called `reserveACopy()` that reserves a copy of a book on behalf of a library member. This method might look as follows, assuming the existence of classes `Member` (representing a person who is a member of the library) and `Book` (representing a single copy of a book):

```
1. private Book reserveACopy(String title, Member member) {
2.   assert isValidTitle(title);
3.
4.   Book book = getAvailableCopy(title);
5.   reserve(book, member);
6.
7.   assert bookIsInStock(book);
8.   return book;
9. }
```

Line 2 enforces a precondition. If the title is not valid (perhaps someone accidentally typed "Moby-Duck"), then the method should terminate as soon as possible, before any damage can be done. In fact, if the precondition fails, the failure indicates that the class needs more work. The code that called `reserveACopy()` with bad arguments needs to be fixed. The assertion failed (we hope) during in-house testing. Eventually the `Library` class would be debugged so that `reserveACopy()` would never be called with bad arguments. At this point (and not before this point), the class would be ready for shipment to the field, where assertions would be disabled.

Line 7 enforces a postcondition. The body of the method is supposed to find an available copy of the desired book. If the book that was found is not available after all, then a problem exists with the method's algorithm. The method should be terminated immediately before the library's data gets hopelessly corrupted, and the method should be debugged. When the author of the method has faith in the algorithm's correctness, the method can be shipped to an environment where assertions can be disabled.

There is a subtle point to be made about the appropriateness of using assertions to check preconditions of public methods. Note that the method in our example was private, so it could be called only from within its own class. Thus if the assertion on line 2 failed, you could only point the finger of blame at yourself or at a colleague down the hall; nobody else could call `reserveACopy()`. However, if the method were public, it could be called by anyone, including a customer who bought the class for re-use in a separate product. Such a programmer is beyond the control of your quality assurance system. A call to `reserveACopy()` with bad arguments would not necessarily indicate

an internal problem with the `Library` class. So, if the `reserveACopy()` method were public, preconditions would have to be checked without the assertion mechanism, because the bad call would happen in the field with assertions disabled. The following code shows how to use an exception to indicate precondition failure:

```
1. public Book reserveACopy(String title, Member member) {
2.   if (!isValidTitle(title))
3.     throw new IllegalArgumentException("Bad title: "tle: + title);
4.
5.   Book book = getAvailableCopy(title);
6.   reserve(book, member);
7.
8.   assert bookIsInStock(book);
9.   return book;
10. }
```

`IllegalArgumentException` is a runtime exception, so `reserveACopy()` does not need a `throws` clause and callers do not need to use the `try/catch` mechanism.

This example demonstrates that assertions are not appropriate for checking preconditions in public methods.

Summary

This chapter covered various aspects of flow control in Java, including loop constructs, selection statements, exception handling and throwing, and assertions. The main points of flow control are summarized here.

Early in the chapter we touched on Java's three loop constructions: `while()`, `do`, and `for()`. Recall that each loop provides the facility for repeating the execution of a block of code until some condition occurs.

The `continue` statement causes the current iteration of the loop to be abandoned, and flow restarts at the top of the loop. The `break` statement abandons the loop altogether. Both `break` and `continue` can take a label that causes them to skip out of multiple levels of a nested loop.

After we covered loop constructions, we discussed Java's two selection constructs: the `if()/else` and `switch()` statements. Both the `if()/else` and `switch()` statements provide a way to conditionally execute code. The `if()` statement takes a `boolean` argument, and the optional `else` statement is executed if the value of that `boolean` argument is false.

The `switch()` statement takes an argument that is compatible to `int` (that is, one of `byte`, `short`, `char`, or `int`). The argument to `case` must be a constant or constant expression that can be calculated at compile time.

As we neared the end of the chapter, we discussed flow in exception handling. An exception causes a jump to the end of the enclosing `try` block even if the exception occurs within a method

called from the `try` block (in which case the called method is abandoned). If no appropriate `catch` block is found, the exception is considered unhandled.

Regardless of whether an exception occurred, or whether it was handled, execution proceeds next to the `finally` block associated with the `try` block, if such a `finally` block exists. If no exception occurred, or if the exception was handled, execution continues after the `finally` block.

If the exception was not handled, then the thread is killed and a message and stack trace are dumped to `System.err`.

Finally, we examined how exceptions are issued in the first place, and how you can write methods that use exceptions to report problems. Any object that is of class `java.lang.Exception`, or any subclass of `java.lang.Exception` (except subclasses of `java.lang.RuntimeException`) is a checked exception. A method cannot throw any `Throwable` other than `RuntimeException`, `Error`, and subclasses of these, unless a `throws` declaration is attached to the method to indicate that this might happen. An overriding method may not throw a checked exception unless the overridden method also throws that exception or a superclass of that exception.

Exam Essentials

Understand the operation of Java `while`, `do`, and `for` loops. Understand labeled loops, labeled breaks, and labeled continues in these loops. You should be able to construct each kind of loop and know when blocks are executed and conditions are evaluated. Know how flow control proceeds in each of these structures.

Know the legal argument types for `if` and `switch()` statements. The argument of an `if` statement must be of type `boolean`. The argument of a `switch()` must be of type `byte`, `short`, `char`, or `int`.

Recognize and create correctly constructed `switch()` statements. You should be able to create regular and default cases, with or without `break` statements.

Analyze code that uses a `try` block, and understand the flow of control no matter what exception types are thrown. You should be completely familiar with all the functionality of the `try`, `catch`, and `finally` blocks.

Understand the difference between checked exceptions and runtime exceptions. Know the inheritance of these families of exception types and know which kinds must be explicitly handled in your code.

Understand all of your exception-handling options when calling methods that throw checked exceptions. You should know how to create `try` blocks and how to declare that a method throws exceptions.

Know what exception types may be thrown when you override a method that throws exceptions. You need to be familiar with the required relationships between the superclass version's exception types and the subclass version's exception types.

Know how to use the assertions facility. You need to know the syntax of `assert` statements, and behavior when the `boolean` statement is true or false. You also need to know how to enable assertions at compile- and runtime.

Key Terms

Before you take the exam, be certain you are familiar with the following terms:

assertion
catch
checked exception
class invariant
condition
error
exception
expression
postcondition
precondition
raised
runtime exception
statement
throw
try

Review Questions

1. Consider the following code:

```
1. for (int i = 0; i < 2; i++) {
2.   for (int j = 0; j < 3; j++) {
3.     if (i == j) {
4.       continue;
5.     }
6.     System.out.println("i = " + i + " j = " + j);
7.   }
8. }
```

Which lines would be part of the output? (Choose all that apply.)

A. `i = 0 j = 0`

B. `i = 0 j = 1`

C. `i = 0 j = 2`

D. `i = 1 j = 0`

E. `i = 1 j = 1`

F. `i = 1 j = 2`

2. Consider the following code. Assume that `i` and `j` have been declared as ints and initialized.

```
1. outer: for (int i = 0; i < 2; i++) {
2.   for (int j = 0; j < 3; j++) {
3.     if (i == j) {
4.       continue outer;
5.     }
6.     System.out.println("i = " + i + " j = " + j);
7.   }
8. }
```

Which lines would be part of the output? (Choose all that apply.)

A. `i = 0 j = 0`

B. `i = 0 j = 1`

C. `i = 0 j = 2`

D. `i = 1 j = 0`

E. `i = 1 j = 1`

F. `i = 1 j = 2`

3. Which of the following are legal loop constructions? (Choose all that apply.)

A.
```
1. while (int i < 7) {
2.   i++;
3.   System.out.println("i is " + i);
4. }
```

B.
```
1. int i = 3;
2. while (i) {
3.   System.out.println("i is " + i);
4. }
```

C.
```
1. int j = 0;
2. for (int k = 0; j + k != 10; j++, k++) {
3.   System.out.println("j is " + j + " k is " + k);
4. }
```

D.
```
1. int j = 0;
2. do {
3.   System.out.println("j is " + j++);
4.   if (j == 3) { continue loop; }
5. } while (j < 10);
```

4. What would be the output from this code fragment?

```
1. int x = 0, y = 4, z = 5;
2. if (x > 2) {
3.   if (y < 5) {
4.     System.out.println("message one");
5.   }
6.   else {
7.     System.out.println("message two");
8.   }
9. }
10. else if (z > 5) {
11.   System.out.println("message three");
12. }
13. else {
14.   System.out.println("message four");
15. }
```

A. `message one`

B. `message two`

C. `message three`

D. `message four`

5. Which statement is true about the following code fragment?

```
1. int j = 2;
2. switch (j) {
3.   case 2:
4.     System.out.println(ìvalue is twoî);
5.   case 2 + 1:
6.     System.out.println(ìvalue is threeî);
7.     break;
8.   default:
9.     System.out.println(ìvalue is ì + j);
10.    break;
11. }
```

A. The code is illegal because of the expression at line 5.

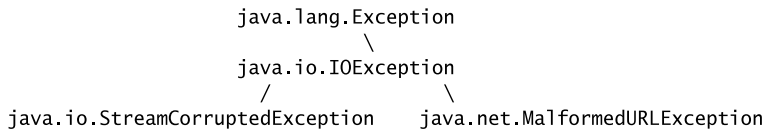
B. The acceptable types for the variable `j`, as the argument to the `switch()` construct, could be any of `byte`, `short`, `int`, or `long`.

C. The output would be the text `value is two`.

D. The output would be the text `value is two` followed by the text `value is three`.

E. The output would be the text `value is two`, followed by the text `value is three`, followed by the text `value is 2`.

6. Consider the following class hierarchy and code fragment:




```
1. try {
2.   // assume s is previously defined
3.   URL u = new URL(s);
4.   // in is an ObjectInputStream
5.   Object o = in.readObject();
6.   System.out.println("Success");
7. }
8. catch (MalformedURLException e) {
9.   System.out.println("Bad URL");
10. }
11. catch (StreamCorruptedException e) {
12.   System.out.println("Bad file contents");
13. }
14. catch (Exception e) {
```

```
15.   System.out.println("General exception");
16. }
17. finally {
18.   System.out.println("Doing finally part");
19. }
20. System.out.println("Carrying on");
```

What lines are output if the constructor at line 3 throws a `MalformedURLException`? (Choose all that apply.)

A. `Success`

B. `Bad URL`

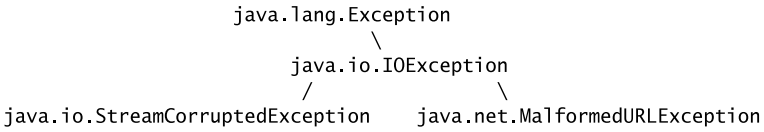
C. `Bad file contents`

D. `General exception`

E. `Doing finally part`

F. `Carrying on`

7. Consider the following class hierarchy and code fragment:




```
1. try {
2.   // assume s is previously defined
3.   URL u = new URL(s);
4.   // in is an ObjectInputStream
5.   Object o = in.readObject();
6.   System.out.println("Success");
7. }
8. catch (MalformedURLException e) {
9.   System.out.println("Bad URL");
10. }
11. catch (StreamCorruptedException e) {
12.   System.out.println("Bad file contents");
13. }
14. catch (Exception e) {
15.   System.out.println("General exception");
16. }
17. finally {
18.   System.out.println("Doing finally part");
```

```
19. }
20. System.out.println("Carrying on");
```

What lines are output if the methods at lines 3 and 5 complete successfully without throwing any exceptions? (Choose all that apply.)

A. `Success`

B. `Bad URL`

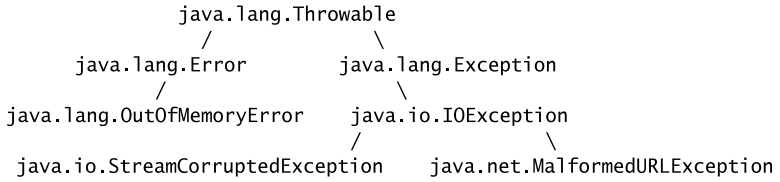
C. `Bad file contents`

D. `General exception`

E. `Doing finally part`

F. `Carrying on`

8. Consider the following class hierarchy and code fragment:




```
1. try {
2.     // assume s is previously defined
3.   URL u = new URL(s);
4.     // in is an ObjectInputStream
5.   Object o = in.readObject();
6.   System.out.println("Success");
7. }
8. catch (MalformedURLException e) {
9.   System.out.println("Bad URL");
10. }
11. catch (StreamCorruptedException e) {
12.   System.out.println("Bad file contents");
13. }
14. catch (Exception e) {
15.   System.out.println("General exception");
16. }
17. finally {
18.   System.out.println("Doing finally part");
19. }
20. System.out.println("Carrying on");
```

What lines are output if the method at line 5 throws an `OutOfMemoryError`? (Choose all that apply.)

A. `Success`

B. `Bad URL`

C. `Bad file contents`

D. `General exception`

E. `Doing finally part`

F. `Carrying on`

9. The method `risky()` might throw a `java.io.IOException`, `java.lang.RuntimeException`, or `java.net.MalformedURLException` (which is a subclass of `java.io.IOException`). Appropriate imports have been declared for each of those exceptions. Which of the following classes and sets of classes are legal? (Choose all that apply.)

A.
```
1. public class SomeClass {
2.   public void aMethod() {
3.     risky();
4.   }
5. }
```

B.
```
1. public class SomeClass {
2.   public void aMethod() throws
3.     IOException {
4.     risky();
5.   }
6. }
```

C.
```
1. public class SomeClass {
2.   public void aMethod() throws
3.     RuntimeException {
4.     risky();
5.   }
6. }
```

D.
```
1. public class SomeClass {
2.   public void aMethod() {
3.     try {
4.       risky();
5.     }
6.     catch (IOException e) {
7.       e.printStackTrace();
8.     }
9.   }
10. }
```

E.
```
1. public class SomeClass {
2.   public void aMethod()
3.     throws MalformedURLException {
4.     try { risky(); }
5.     catch (IOException e) {
6.       // ignore it
7.     }
8.   }
9. }
10.
11. public class AnotherClass
12.   extends SomeClass {
13.   public void aMethod()
14.     throws java.io.IOException {
15.     super.aMethod();
16.   }
17. }
```

10. Consider the following code:

```
1. public class Assertification {
2.   public static void main(String[] args) {
3.     assert args.length == 0;
4    }
5. }
```

Which of the following conditions must be true in order for the code to throw an `AssertionError`? (Choose all that apply.)

A. The source code must be compiled with the `-source 1.4` flag.

B. The application must be run with the `-enableassertions` flag or another assertion-enabling flag.

C. The `args` array must have exactly zero elements.

D. The `args` array must have one or more elements.

Answers to Review Questions

1. B, C, D, F. The loops iterate `i` from 0 to 1 and `j` from 0 to 2. However, the inner loop executes a `continue` statement whenever the values of `i` and `j` are the same. Because the output is generated inside the inner loop, after the `continue` statement, no output is generated when the values are the same. Therefore, the outputs suggested by answers A and E are skipped.

2. D. It seems that the variable `i` will take the values 0 and 1, and for each of these values, `j` will take values 0, 1, and 2. However, whenever `i` and `j` have the same value, the outer loop is continued before the output is generated. Because the outer loop is the target of the `continue` statement, the whole of the inner loop is abandoned. So, for the value pairs, this table shows what happens:

`i`	`j`	Effect
0	0	Continues at line 4
1	0	Prints at line 6
1	1	Continues at line 4
2	1	Exits loops at line 1

   Therefore, the only line to be output is that shown in D.

3. C. In A, the variable declaration for `i` is illegal. This type of declaration is permitted only in the first part of a `for()` loop. The absence of initialization should also be a clue here. In B, the loop control expression—the variable `i` in this case—is of type `int`. A `boolean` expression is required. C is valid. Despite the complexity of declaring one value inside the `for()` construction and one outside (along with the use of the comma operator in the end part), this code is entirely legitimate. D would be correct, except that the label has been omitted from line 2, which should read `loop: do {`.

4. D. The first test at line 2 fails, which immediately causes control to skip to line 10, bypassing both the possible tests that might result in the output of `message one` or `message two`. So, even though the test at line 3 would be true, it is never made; A is not correct. At line 10, the test is again false, so the message at line 11 is skipped, but `message four`, at line 14, is output.

5. D. A is incorrect because the code is legal despite the expression at line 5; the expression itself is a constant. B is incorrect because it states that the `switch()` part can take a `long` argument. Only `byte`, `short`, `char`, and `int` are acceptable. The output results from the value 2 like this: first, the option `case 2:` is selected, which outputs `value is two`. However, there is no `break` statement between lines 4 and 5, so the execution falls into the next `case` and outputs `value is three` from line 6. The `default:` part of a `switch()` is executed only when no other options have been selected, or if no `break` precedes it. In this case, neither of these situations holds true, so the output consists only of the two messages listed in D.

6. B, E, F. The exception causes a jump out of the `try` block, so the message `Success` from line 6 is not printed. The first applicable `catch` is at line 8, which is an exact match for the thrown exception. This results in the message at line 9 being printed, so B is one of the required answers. Only one `catch` block is ever executed, so control passes to the `finally` block, which results in the message at line 18 being output; so E is part of the correct answer. Because the exception was caught, it is considered to have been handled, and execution continues after the `finally` block. This results in the output of the message at line 20, so F is also part of the correct answer.

7. A, E, F. With no exceptions, the `try` block executes to completion, so the message `Success` from line 6 is printed and A is part of the correct answer. No `catch` is executed, so B, C, and D are incorrect. Control then passes to the `finally` block, which results in the message at line 18 being output, so E is part of the correct answer. Because no exception was thrown, execution continues after the `finally` block, resulting in the output of the message at line 20; so, F is also part of the correct answer.

8. E. The thrown error prevents completion of the `try` block, so the message `Success` from line 6 is not printed. No `catch` is appropriate, so B, C, and D are incorrect. Control then passes to the `finally` block, which results in the message at line 18 being output; so option E is part of the correct answer. Because the error was not caught, execution exits the method and the error is rethrown in the caller of this method; so, F is not part of the correct answer.

9. B, D. A does not handle the exceptions, so the method `aMethod()` might throw any of the exceptions that `risky()` might throw. However, the exceptions are not declared with a `throws` construction. In B, declaring `throws IOException` is sufficient; because `java.lang.RuntimeException` is not a checked exception and because `IOException` is a superclass of `MalformedURLException`, it is unnecessary to mention the `MalformedURLException` explicitly (although it might make better "self-documentation" to do so). C is unacceptable because its `throws` declaration fails to mention the checked exceptions—it is not an error to declare the runtime exception, although it is strictly redundant. D is also acceptable, because the `catch` block handles `IOException`, which includes `MalformedURLException`. `RuntimeException` will still be thrown by the method `aMethod()` if it is thrown by `risky()`, but because `RuntimeException` is not a checked exception, this is not an error. E is not acceptable, because the overriding method in `anotherClass` is declared as throwing `IOException`, whereas the overridden method in `aClass` was only declared as throwing `MalformedURLException`. It would have been correct for the base class to declare that it throws `IOException` and then the derived class to throw `MalformedURLException`, but as it is, the overriding method is attempting to throw exceptions not declared for the original method. The fact that the only exception that can arise is the `MalformedURLException` is not enough to rescue this code—the compiler only checks the declarations, not the semantics of the code.

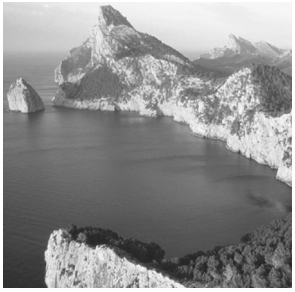
10. A, B, D. If the source is not compiled with the `-source 1.4` flag, `assert` will be treated as an identifier rather than as a keyword. If the application is not run with assertions explicitly enabled, all `assert` statements will be ignored. If the `args` array does not have exactly zero arguments, no `AssertionError` will be thrown.

Chapter 6

Objects and Classes

JAVA CERTIFICATION EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- ✓ **1.4 Identify legal return types for any method given the declarations of all related methods in this or parent classes.**
- ✓ **6.1 State the benefits of encapsulation in object-oriented design and write code that implements tightly encapsulated classes and the relationships "is a" and "has a".**
- ✓ **6.2 Write code to invoke overridden or overloaded methods and parental or overloaded constructors; and describe the effect of invoking these methods.**
- ✓ **6.3 Write code to construct instances of any concrete class including normal top-level classes and nested classes.**



This chapter discusses the object-oriented features of Java. Good coding in Java requires a sound understanding of the object-oriented (OO) paradigm, and this in turn requires a good grasp of the language features that implement objects and classes. The many benefits of object orientation have been the subject of considerable public debate, but for many programmers these benefits have not been realized. In most cases, the reason the promise has not been fulfilled is simply that programmers have not been writing objects. Instead, many programmers have been writing hybrid applications with a mixture of procedural and object-oriented code. Unfortunately, while such an approach has given rise to *some* of the benefits of OO, it has also engendered *all* the disadvantages of both styles.

Benefits of Object-Oriented Implementation

The Programmer's and Developer's Exams require you to understand the benefits of object-oriented design. These benefits accrue from two particular features of the OO paradigm. The first of these, and perhaps the most important, is the notion of *encapsulation*; the second and perhaps better known is the extensibility provided by *inheritance*.

Encapsulation

Encapsulation is really just a fancy name for the aggregation of data and behavior. Consider the primitive data types of any programming language you have ever used. You do not know how these data items are stored and, for the most part, you do not care. What matters are the operations that you can perform on these data items and the boundary conditions within which you can expect those operations to properly work. These primitive types are in fact reference types, albeit not user-defined.

Your first goal in defining a good class should be to clearly define the data members that describe instances of that class, keeping in mind that this should be done only with variables of private accessibility. Next, consider how to represent the behavior associated with these data. All behavior should be accessed only via methods. By insisting that the variables inside an object are inaccessible outside the object, you ensure that the nature of those variables is irrelevant outside the object. This in turn means that you can freely change the nature of the storage for maintenance purposes, performance improvement, or any other reason. This is the essence of encapsulation.

Sometimes, perhaps as a consequence of the way you have stored the state in a class, boundary conditions must be applied to its methods. A *boundary condition* is a limit on the range of arguments for which a method can operate properly. As examples, a square-root function cannot operate on a negative number unless imaginary numbers are included in its range; an add operation cannot operate if both of its arguments are more than half the maximum value for the operation's return type.

When you encounter a boundary condition that results from your choice of storage format, you must make a choice. If you consider that the boundary conditions are reasonable, then you should do two things. First, document the boundary condition. Next, test the boundary conditions at the entry to the method and, if the boundary condition has been exceeded, throw a runtime exception of some kind. Alternatively, you might decide that the boundary condition is not acceptable, in which case you should redesign the storage used in the class.

Now, consider this: if you had allowed access to any of the variables used to represent the object state, then redefining the way the object's state is stored would immediately cause any other code that uses these variables to have to be rewritten. However, by using only private member variables, you have insisted that all interaction with this object is made through methods and never by direct variable access—so you have eliminated this problem. In consequence, you are able to redesign your internal storage freely and, provided the signatures of all the methods remain the same, no other code needs to change.

Encapsulation and Perceived Efficiency

Many programmers have such deep-seated concerns about performance that they cannot bring themselves to force all access to their objects to be made through methods, and they resist creating classes with entirely private member variables. This approach is unwise for several reasons. First, fully encapsulated classes are more likely to be used correctly, especially if boundary conditions are properly flagged with exceptions—therefore, code using them is more likely to be correct. Second, bug fixes and maintenance changes are less likely to break the program as a whole, because the effects of the change are confined to the affected class. These reasons fall under the broad heading, "Would your customer prefer a slow program that works and is delivered on time (and that can be made faster later) or a program that is delivered late, works incorrectly, but runs quickly?"

There are more reasons fully encapsulated classes are the right way to begin a design. An optimizing virtual machine such as Sun's HotSpot can transparently optimize simple variable access methods by "inlining." This approach allows the program all the robustness, reliability, and maintainability that results from full encapsulation, while giving the runtime performance associated with direct variable access. Furthermore, if you decide that a program's slow performance is attributable to the use of private variables and accessor/mutator methods, then changing the variable to be more accessible does not require any changes to the rest of the code, either inside or outside the class. On the other hand, if you have code that fails to run properly as a result of making direct variable access, you will find that reducing the accessibility of the relevant variable will require considerable collateral changes in many other pieces of code (all code that makes such direct access).

Re-use

We discussed how tight encapsulation can make code that is more reliable and robust. Now we will consider the second most significant advantage of object-oriented programming: code re-use.

Writing good, encapsulated classes usually requires more work in the initial stages than would be required to produce the same functionality with a traditional programming approach. However, you will normally find that using rigorous OO techniques will actually reduce the overall time required to produce finished code. This is the case for two reasons. First, the robust classes you produce require less time to integrate into the final program and less time to fix bugs. Second, with careful design, you can re-use classes even in some circumstances that are different from the original intent of the class.

This re-use is possible in two ways, using either composition (the "has a" relation) or inheritance (the "is a" relation). Composition is probably safer and easier to control, although inheritance—perhaps because it is perceived as "pure OO"—seems to be more interesting and appealing to most programmers.

The Java Certification Exam does not require you to discuss details of object-oriented design techniques or the relative merits and weaknesses of composition versus inheritance. However, you should appreciate one significant sequence of facts: if a class is well-encapsulated, it will be easier to re-use successfully. The more a class is re-used, the better tested it will be and the fewer bugs it will have. Better-tested, less-buggy classes are easier to re-use. This sequence leads to a positive spiral of quality because the better the class, the easier and safer it becomes to re-use. All these benefits come from tight encapsulation.

Now that we've discussed why you would want to write object-oriented code, let's look at how this is achieved.

Implementing Object-Oriented Relationships

This section is not intended to discuss object-oriented design; rather, it considers the implementation of classes for which you have been given a basic description.

Two clauses are commonly used when describing a class in plain English: "is a" and "has a." As a working simplification, they are used to describe the superclass and member variables, respectively. For example, consider this description:

> "A home is a house that has a family and a pet."

This description would give rise to the outline of a Java class in this form:

```
1. public class Home extends House {
2.   Family inhabitants;
3.   Pet thePet;
4. }
```

Notice the direct correspondence between the "is a" clause and the `extends` clause. In this example, a direct correspondence also exists between the items listed after "has a" and the member variables. Such a correspondence is representative in simple examples and in a test situation; however, you should be aware that in real examples, there are other ways you can provide a class with attributes. Probably the most important of these alternatives is the approach taken by JavaBeans, which is to supply accessor and mutator methods that operate on private data members.

The example shown is simplified to focus on the knowledge and understanding that is required by the exam. In a real situation, the variables should generally be private (or at least some specific rationale should apply to whatever accessibility they have), and some methods will be needed in the class.

Overloading and Overriding

As you construct classes and add methods to them, in some circumstances you will want to re-use the same name for a method. You can do so two ways with Java. Re-using the same method name with different arguments and perhaps a different return type is known as *overloading*. Using the same method name with identical arguments and return type is known as *overriding*.

A method name can be re-used anywhere, as long as certain conditions are met:

- In an unrelated class, no special conditions apply, and the two methods are not considered related in any way.
- In the class that defines the original method, or a subclass of that class, the method name can be re-used if the argument list differs in terms of the type of at least one argument. This is overloading. It is important to realize that a difference in return type or list of thrown exceptions is insufficient to constitute an overload and is illegal.
- In a strict subclass of the class that defines the original method, the method name can be re-used with identical argument types and order and with identical return type. This is overriding. In this case, additional restrictions apply to the accessibility of, and exceptions that may be thrown by, the method.

In general, a class is considered to be a subclass of itself. That is, if classes `A`, `B`, and `C` are defined so that `C` extends `B` and `B` extends `A`, then the subclasses of `A` are `A`, `B`, and `C`. The term *strict subclass* is used to describe the subclasses excluding the class itself. So the strict subclasses of `A` are only `B` and `C`.

Now let's take a look at these ideas in detail. First, we will consider overloading method names.

Overloading Method Names

In Java, a method is uniquely identified by the combination of its fully qualified class name, the method name, and the exact sequence of its argument types. Overloading is the re-use of a method name in the one class or subclass for a different method. It is not related to object orientation, although a purely coincidental correlation shows that object-oriented languages are more likely to support overloading. Notice that overloading is essentially a trick with names; hence this section's title is "Overloading Method Names" rather than "Overloading Methods." The following are all different methods:

```
1. public void aMethod(String s) { }
2. public void aMethod() { }
3. public void aMethod(int i, String s) { }
4. public void aMethod(String s, int i) { }
```

These methods all have identical return types and names, but their argument lists are different either in the types of the arguments that they take or in the order. Only the argument *types* are considered, not their names, so a method such as

```
public void aMethod(int j, String name) { }
```

would *not* be distinguished from the method defined in line 3.

What Is Overloading For?

Why is overloading useful? Sometimes you will be creating several methods that perform closely related functions under different conditions. For example, imagine methods that calculate the area of a triangle. One such method might take the Cartesian coordinates of the three vertices, and another might take the polar coordinates. A third method might take the lengths of all three sides, whereas a fourth might take three angles and the length of one side. These methods would all be performing the same essential function, so it is entirely proper to use the same name for the methods. In languages that do not permit overloading, you would have to think up four different method names, such as:

```
areaByCoord(Point p, Point q, Point r)
areaByPolarCoord(PolarPt p, PolarPt q, PolarPt r)
areaBySideLengths(int l1, int l2, int l3)
areaByAnglesAndASide(int l1, int angle1, int angle2, int angle3)
```

Overloading is really nothing new. Almost every language that has a type system has used overloading in a way, although most have not allowed the programmer free use of it. Consider the arithmetic operators +, -, *, and /. In most languages, they can be used with integer or floating-point operands. The implementation of, say, multiplication for integer and floating-point operands generally involves completely different code, and yet the compiler permits the same symbol to be used. Because the operand types are different, the compiler can decide which version of the operation

should be used. This process is known as *operator overloading* and is the same principle as method overloading.

It is quite useful, for thinking up method names and for improving program readability, to be able to use one method name for several related methods requiring different implementations. However, you should restrict your use of overloaded method names to situations where the methods really are performing the same basic function with different data sets. Methods that perform different jobs should have different names.

One last point to consider is the return type of an overloaded method. The language treats methods with overloaded names as totally different methods, and as such they *can* have different return types (you will see shortly that overriding methods do not have this freedom).

Invoking Overloaded Methods

When you write multiple methods that perform the same basic function with different arguments, you often find that it would be useful to call one of these methods as support for another version. Consider a method called `printRJ()` that is to be provided in versions that take a `String` or an `int` value. The version that takes an `int` could most easily be coded so that it converts the `int` to a `String` and then calls the version that operates on `String` objects.

You can do this easily. Remember that the compiler decides which method to call simply by looking at the argument list, and that the various overloaded methods are in fact unrelated. All you have to do is write the method call exactly as normal—the compiler will do the rest. Consider this example:

```
 1. public class RightJustify {
 2.   // Declare a String of 80 spaces
 3.   private static final String padding =
 4.     "                    " +
 5.     "                    " +
 6.     "                    " +
 7.     "                    ";
 8.   public static void printRJ(String s, int w) {
 9.     System.out.print(
10.       padding.substring(0, w - s.length()));
11.     System.out.print(s);
12.   }
13.   public static void printRJ(int i, int w) {
14.     printRJ("" + ", w);
15.   }
16. }
```

At line 14, the `int` argument is converted to a `String` object by adding it to an empty `String`. The method call at this same line is then seen by the compiler as a call to a method called `print()` that takes a `String` as the first argument, which results in selection of the method at line 8.

To summarize, these are the key points about overloading methods:

- The identity of a method is determined by the combination of its fully qualified class; its name; and the type, order, and count of arguments in the argument list.
- Two or more methods in the same class (including methods inherited from a superclass) with the same name but different argument lists are called *overloaded*.
- Methods with overloaded names are effectively independent methods—using the same name is really just a convenience to the programmer. Return type, accessibility, and exception lists may vary freely.

Now that we have considered overloading thoroughly, let's look at overriding.

Method Overriding

You have just seen that overloading is essentially a trick with names, effectively treating the argument list as part of the method identification. Overriding is somewhat more subtle, relating directly to subclassing and hence to the object-oriented nature of a language.

When you extend one class to produce a new one, you inherit and have access to certain nonprivate methods of the original class (as dictated by access modifiers and package relationships). Sometimes, however, you might need to modify the behavior of one of these methods to suit your new class. In this case, you actually want to redefine the method, and this is the essential purpose of overriding.

There are a number of key distinctions between overloading and overriding:

- Overloaded methods supplement each other; an overriding method replaces the method it overrides.
- Overloaded methods can exist, in any number, in the same class. Each method in a parent class can be overridden at most once in any one subclass.
- Overloaded methods must have *different* argument lists; overriding methods must have argument lists of *identical* type and order (otherwise they are simply treated as overloaded methods).
- The return type of an overloaded method may be chosen freely; the return type of an overriding method must be *identical* to that of the method it overrides.
- The exception list of an overloaded method may be chosen according to the rules defined earlier in this chapter.
- The access modifiers of an overloaded method may be chosen according to the rules defined earlier in this chapter.

What Is Overriding For?

Overloading allows multiple implementations of the same essential functionality to use the same name. Overriding, on the other hand, modifies the implementation of a particular piece of behavior for a subclass.

Consider a class that describes a rectangle. Imaginatively, we'll call it `Rectangle`. We're talking about an abstract rectangle here, so no visual representation is associated with it. This class has a

method called `setSize()`, which is used to set width and height values. In the `Rectangle` class, the implementation of the `setSize()` method simply sets the value of the private width and height variables for later use. Now, imagine you create a `DisplayedRectangle` class that is a subclass of the original `Rectangle`. When the `setSize()` method is called, you need to arrange a new behavior. Specifically, the width and height variables must be changed, but also the visual representation must be redrawn. This is achieved by overriding.

If you define a method that has exactly the same name and exactly the same argument types as a method in a parent class, then you are overriding the method. Under these conditions, the method must also have the identical return type and follow the accessibility and exception list rules for that of the method it overrides. Consider this example:

```
 1. class Rectangle {
 2.   int x, y, w, h;
 3.
 4.   public void setSize(int w, int h) {
 5.     this.w = w; this.h = h;
 6.   }
 7. }
 8. class DisplayedRectangle extends Rectangle {
 9.   public void setSize(int w, int h) {
10.     this.w = w; this.h = h;
11.     redisplay(); // implementation
12.   }
13.   public void redisplay() {
14.     // implementation not shown
15.   }
16. }
17.
18. public class TestRectangle {
19.   public static void main(String args[]) {
20.     Rectangle [] recs = new Rectangle[4];
21.     recs[0] = new Rectangle();
22.     recs[1] = new DisplayedRectangle();
23.     recs[2] = new DisplayedRectangle();
24.     recs[3] = new Rectangle();
25.     for (int r=0; r<4; r++) {
26.       int w = ((int)(Math.random() * 400));
27.       int h = ((int)(Math.random() * 200));
28.       recs[r].setSize(w, h);
29.     }
30.   }
31. }
```

Clearly this example is incomplete, because no code exists to cause the display of the `DisplayedRectangle` objects, but it is complete enough for us to discuss.

At line 20, the array `recs` is created as an array of `Rectangle` objects; yet at lines 21–24, the array is used to hold not only two instances of `Rectangle` but also two instances of `DisplayedRectangle`. Subsequently, when the `setSize()` method is called, it will be important that the executed code be the code associated with the actual object referred to by the array element, rather than always being the code of the `Rectangle` class. This is exactly what Java does, and this is the essential point of overriding methods. It is as if you ask an object to perform certain behavior, and that object makes its own interpretation of the request. C++ programmers should take particular note of this point, because it differs significantly from the default behavior of overriding methods in that language.

In order for any particular method to override another correctly, some requirements must be met. Some of them have been mentioned before in comparison with overloading, but all are listed here for completeness:

- The method name and the type and order of arguments must be identical to those of a method in a parent class. If this is the case, then the method is an attempt to override the corresponding parent class method and the remaining points listed here must be adhered to, or a compiler error arises. If these criteria are not met, then the method is not an attempt to override and the following rules are irrelevant.
- The return type must be identical.
- Methods marked `final` may not be overridden.
- The accessibility must not be more restrictive than that of the original method.
- The method must not throw new or broader checked exceptions of classes than are thrown by the original method.

The first three points have been covered, but the last two are new. The accessibility of an overriding method must not be less than that of the method it overrides, simply because it is considered to be the replacement method in conditions like those of the rectangles example earlier. So, imagine that the `setSize()` method of `DisplayedRectangle` was inaccessible from the `main()` method of the `TestRectangle` class. The calls to `recs[1].setSize()` and `recs[2].setSize()` would be illegal, but the compiler would be unable to determine this because it only knows that the elements of the array are `Rectangle` objects. The `extends` keyword literally requires that the subclass be an extension of the parent class: if methods could be removed from the class or made less accessible, then the subclass would not be a simple extension but would potentially be a reduction. Under those conditions, the idea of treating `DisplayedRectangle` objects as being `Rectangle` objects when used as method arguments or elements of a collection would be severely flawed.

A similar logic gives rise to the final rule relating to checked exceptions. Checked exceptions are those that the compiler ensures are handled in the source you write. As with accessibility, it must be possible for the compiler to make correct use of a variable of the parent class even if that variable really refers to an object of a derived class. For checked exceptions, this requirement means that an overriding method must not be able to throw exceptions that would not be thrown by the original method. Chapter 5, "Flow Control, Assertions, and Exception Handling," discusses checked exceptions and this rule in more detail.

Late Binding

Normally, when a compiler for a non-object-oriented language comes across a method (or function or procedure) invocation, it determines exactly what target code should be called and builds machine language to represent that call. In an object-oriented language, this behavior is not possible because the proper code to invoke is determined based upon the class of the object being used to make the call, not the type of the variable. Instead, code is generated that will allow the decision to be made at runtime. This delayed decision-making is variously referred to as *late binding* (*binding* is one term for the job a linker does when it glues various bits of machine code together to make an executable program file).

The Java Virtual Machine (JVM) has been designed from the start to support an object-oriented programming system, so there are machine-level instructions for making method calls. The compiler only needs to prepare the argument list and produce one method invocation instruction; the job of identifying and calling the proper target code is performed by the JVM.

If the JVM is to be able to decide what code should be invoked by a particular method call, it must be able to determine the class of the object upon which the call is based. Again, the JVM design has supported this process from the beginning. Unlike traditional languages or runtime environments, every time the Java system allocates memory, it marks that memory with the type of the data that it has been allocated to hold. So, given any object, and without regard to the type associated with the reference variable acting as a handle to that object, the runtime system can determine the real class of that object by inspection. This process is the basis of the `instanceof` operator, which allows you to program a test to determine the actual class of an object at runtime. The `instanceof` operator is described in Chapter 2, "Operators and Assignments."

Invoking Overridden Methods

When we discussed overloading methods, you saw how to invoke one version of a method from another. It is also useful to be able to invoke an overridden method from the method that overrides it. Consider that when you write an overriding method, that method entirely replaces the original method. However, sometimes you wish only to add a little extra behavior and want to retain all the original behavior. This goal can be achieved, although it requires a small trick of syntax to perform. Look at this example:

```
 1. class Rectangle {
 2.   private int x, y, w, h;
 3.   public String toString() {
 4.     return "x = " + x + ", y = " + y +
 5.       ", w = " + w + ", h = " + h;
 6.   }
 7. }
 8. class DecoratedRectangle extends Rectangle {
 9.   private int borderWidth;
10.   public String toString() {
11.     return super.toString() + ", borderWidth = " +
```

```
12.     borderWidth;
13.   }
14. }
```

At line 11, the overriding method in the `DecoratedRectangle` class uses the parental `toString()` method to perform the greater part of its work. Because the variables `x`, `y`, `w`, and `h` in the `Rectangle` class are marked as `private`, it would have been impossible for the overriding method in `DecoratedRectangle` to achieve its work directly.

A call of the form `super.xxx()` always invokes the behavior that would have been used if the current overriding method had not been defined. It does not matter whether the parental method is defined in the immediate superclass or in some ancestor class further up the hierarchy: `super` invokes the version of this method that is "next up the tree."

To summarize, these are the key points about overriding methods:

- A method that has an identical name and identical number, types, and order of arguments as a method in a parent class is an overriding method.
- Each parent class method may be overridden once at most in any one subclass. (That is, you cannot have two identical methods in the same class.)
- An overriding method must return exactly the same type as the method it overrides.
- An overriding method must not be less accessible than the method it overrides.
- An overriding method must not throw any checked exceptions (or subclasses of those exceptions) that are not declared for the overridden method.
- An overridden method is completely replaced by the overriding method unless the overridden method is deliberately invoked from within the subclass.

This is quite a lot to think about, so you might like to take a break before you move on to the next topic: constructors.

Constructors and Subclassing

Inheritance generally makes the code and data defined in a parent class available for use in a subclass. This is subject to accessibility controls so that, for example, private items in the parent class are not directly accessible in the methods of the subclass, even though they exist. In fact, constructors are not inherited in the normal way but must be defined for each class in the class itself.

A constructor is invoked with a call of the form `new MyClass(arg1, arg2, ...)`. If the argument list is empty, the constructor is called a *no-arguments* (or *no-args*) *constructor*. If you do not explicitly code any constructors for a class, the compiler automatically creates a default constructor that does nothing except invoke the superclass's default constructor, via a mechanism described in the next section. This "freebie" constructor is called the *default constructor*. It has public access if the class is public; otherwise its access mode is default.

Often you will define a constructor that takes arguments and will want to use those arguments to control the construction of the parent part of the object. You can pass control to a constructor in the parent class by using the keyword `super`. To control the particular constructor that is used, you simply provide the appropriate arguments. Consider this example:

```
 1. class Base {
 2.   public Base(String s) {
 3.     // initialize this object using s
 4.   }
 5.   public Base(int i) {
 6.     // initialize this object using i
 7.   }
 8. }
 9.
10. class Derived extends Base {
11.   public Derived(String s) {
12.     // pass control to Base constructor at line 2
13.     super(s);
14.   }
15.   public Derived(int i) {
16.     // pass control to Base constructor at line 5
17.     super(i);
18.   }
19. }
```

The code at lines 13 and 17 demonstrates the use of `super()` to control the construction of the parent class part of an object. The definitions of the constructors at lines 11 and 15 select an appropriate way to build their inherited part by invoking `super()` with an argument list that matches one of the constructors for the parent class. It is important to know that the superclass constructor must be called before any reference is made to any part of this object. This rule is imposed to guarantee that nothing is ever accessed in an uninitialized state. Generally, the rule means that if `super()` is to appear at all in a constructor, then it must be the first statement.

Although the example shows the invocation of parental constructors with argument lists that match those of the original constructor, this is not a requirement. It would be perfectly acceptable, for example, if line 17 read:

```
17.     super("Value is " + i);
```

This would have caused control to be passed to the constructor at line 2, which takes a `String` argument, rather than the one at line 5.

Overloading Constructors

Although you just saw that constructors are not inherited in the same way as methods, the overloading mechanisms apply quite normally. In fact, the example discussing the use of `super()` to control the invocation of parental constructors showed overloaded constructors. You saw earlier how you could invoke one method from another that overloads its name simply by calling the method with an appropriate parameter list. There are also times when it's useful to invoke one constructor from another. Imagine you have a constructor that takes five arguments and does considerable processing to initialize the object. You wish to provide another constructor that takes only two arguments and sets the remaining three to default values. It would be nice to avoid re-coding the body of the first constructor and instead simply set up the default values and pass control to the first constructor. You can do so using a small trick of syntax.

Usually, you would invoke a method by using its name followed by an argument list in parentheses, and you would invoke a constructor by using the keyword `new`, followed by the name of the class, followed again by an argument list in parentheses. Thus you might try to use the `new ClassName(args)` construction to invoke another constructor of your own class. Unfortunately, although this is legal syntax, it results in an entirely separate object being created. The approach Java takes is to provide another meaning for the keyword `this`. Look at this example:

```
1. public class AnyClass {
2.   public AnyClass(int a, String b, float c, Date d) {
3.     // complex processing to initialize
4.     // based on arguments
5.   }
6.   public AnyClass(int a) {
7.     this(a, "default", 0.0F, new Date());
8.   }
9. }
```

The constructor at line 6 takes a single argument and uses that, along with three other default values, to call the constructor at line 2. The call is made using the `this()` construction at line 7. As with `super()`, `this()` must be positioned as the first statement of the constructor.

We have said that any use of either `super()` or `this()` in a constructor must be placed at the first line. Clearly, you cannot put both on the first line. If you write a constructor that has neither a call to `super()` nor a call to `this()`, then the compiler automatically inserts a call to the parent class constructor with no arguments. If an explicit call to another constructor is made using `this()`, then the superclass constructor is not called until the other constructor runs. It is permitted for that other constructor to start with a call to either `this()` or `super()`, if desired. Java insists that the object is initialized from the top of the class hierarchy downward; that is why the call to `super()` or `this()` must occur at the start of a constructor. This point has an important consequence. We just said that if there is no call to either `this()` or `super()`,

then the compiler puts in a call to the no-argument constructor in the parent. As a result, if you try to extend a class that does not have a no-argument constructor, then you *must* explicitly call `super()` with one of the argument forms that are supported by constructors in the parent class.

Let's summarize the key points about constructors before we move on to inner classes:

- Constructors are not inherited in the same way as normal methods. You can only create an object if the class defines a constructor with an argument list that matches the one your `new` call provides.
- If you define no constructors in a class, then the compiler provides a default that takes no arguments. If you define even a single constructor, this default is not provided.
- It is common to provide multiple overloaded constructors—that is, constructors with different argument lists. One constructor can call another using the syntax `this(`*`arguments`*`)`.
- A constructor delays running its body until the parent parts of the class have been initialized. This commonly happens because of an implicit call to `super()` added by the compiler. You can provide your own call to `super(`*`arguments`*`)` to control the way the parent parts are initialized. If you do so, it must be the first statement of the constructor.
- A constructor can use overloaded constructor versions to support its work. These are invoked using the syntax `this(`*`arguments`*`)` and if supplied, this call must be the first statement of the constructor. In such conditions, the initialization of the parent class is performed in the overloaded constructor.

Inner Classes

The material we have looked at so far has been part of Java since its earliest versions. Inner classes are a feature added with the release of JDK 1.1. *Inner classes*, which are sometimes called *nested classes*, can give your programs additional clarity and make them more concise.

Fundamentally, an inner class is the same as any other class, but is declared inside (that is, between the opening and closing curly braces of) some other class. In fact, you can declare nested classes in any block, including blocks that are part of a method. Classes defined inside a method differ slightly from the more general case of inner classes that are defined as members of a class; we'll look at these differences in detail later. For now, when we refer to a "member class," we mean a class that is *not* defined in a method but rather in a class. In this context, the use of the term member is closely parallel to its use in the context of member variables and member methods.

The complexity of inner classes relates to scope and access—particularly access to variables in enclosing scopes. Before we consider these matters, let's look at the syntax of a basic inner class, which is really quite simple. Consider this example:

```
1. public class OuterOne {
2.   private int x;
```

```
 3.   public class InnerOne {
 4.     private int y;
 5.     public void innerMethod() {
 6.       System.out.println("y is " + y);
 7.     }
 8.   }
 9.   public void outerMethod() {
10.     System.out.println("x is " + x);
11.   }
12.   // other methods...
13. }
```

In this example, there is no obvious benefit in having declared the class called `InnerOne` as an inner class; so far we are only looking at the syntax. When an inner class is declared like this, the enclosing class name becomes part of the fully qualified name of the inner class. In this case, the two classes' full names are `OuterOne` and `OuterOne.InnerOne`. This format is reminiscent of a class called `InnerOne` declared in a package called `OuterOne`. This point of view is not entirely inappropriate, because an inner class belongs to its enclosing class in a fashion similar to the way a class belongs to a package. It is illegal for a package and a class to have the same name, so there can be no ambiguity.

Although the dotted representation of inner class names works for the declaration of the type of an identifier, it does not reflect the filename of the class. If you try to load this class using the `Class.forName()` method, the call will fail. On the disk, and from the point of view of the `Class` class and class loaders, the name of the class is `OuterOne$InnerOne`. The dollar-separated name is also used if you print out the class name by using the methods `getClass().getName()` on an instance of the inner class. You probably recall that classes are located in directories that reflect their package names. The dollar-separated convention is adopted for inner class names to ensure that there is no ambiguity on the disk between inner classes and package members. It also reduces conflicts with file systems and shell interpreters that treat the dot character as special, perhaps limiting the number of characters that can follow it.

Although for the purpose of naming, being able to define a class inside another class provides some organizational benefit, this is not the end of the story. Objects that are instances of the inner class generally retain the ability to access the members of the outer class. This behavior is discussed in the next section.

Real World Scenario

Inner Class Details

Write an application that answers the following two questions:

1. Suppose an enclosing class contains a private nonanonymous inner class, which implements a particular interface. Suppose a method of the enclosing class returns an instance of the inner class. Can that method be called from code outside the enclosing class?

2. What if the inner class in Question 1 is anonymous?

Think about how the code that calls the method will reference the return value. The reference type clearly cannot be the inner class, because the inner class is invisible to the calling class. But remember that the inner class implements an interface.

The Enclosing *this* Reference and Construction of Inner Classes

When an instance of an inner class is created, normally a preexisting instance of the outer class must act as context. This instance of the outer class will be accessible from the inner object. Consider this example, which is expanded from the earlier one:

```
 1. public class OuterOne {
 2.   private int x;
 3.   public class InnerOne {
 4.     private int y;
 5.     public void innerMethod() {
 6.       System.out.println("enclosing x is " + x);
 7.       System.out.println("y is " + y);
 8.     }
 9.   }
10.   public void outerMethod() {
11.     System.out.println("x is " + x);
12.   }
13.   public void makeInner() {
14.     InnerOne anInner = new InnerOne();
15.     anInner.innerMethod();
16.   }
17.   // other methods...
18. }
```

You will see two changes in this code when you compare it to the earlier version. First, `innerMethod()` now not only outputs the value of `y`, which is defined in `InnerOne`, but also, at line 6, outputs the value of `x`, which is defined in `OuterOne`. The second change is that in lines 13–16, the code creates an instance of the `InnerOne` class and invokes `innerMethod()` upon it.

The accessibility of the members of the enclosing class is crucial and very useful. It is possible because the inner class has a hidden reference to the outer class instance that was the current context when the inner class object was created. In effect, it ensures that the inner class and the outer class belong together, rather than the inner instance being just another member of the outer instance.

Sometimes you might want to create an instance of an inner class from a static method, or in some other situation where no `this` object is available. The situation arises in a `main()` method or if you need to create the inner class from a method of some object of an unrelated class. You can achieve this by using the `new` operator as though it were a member method of the outer class. Of course, you still must have an instance of the outer class. The following code, which is a `main()` method in isolation, could be added to the code seen so far to produce a complete example:

```
1. public static void main(String args[]) {
2.   OuterOne.InnerOne i = new OuterOne().new InnerOne();
3.   i.innerMethod();
4. }
```

From the point of view of the inner class instance, this use of two `new` statements on the same line is a compacted way of doing the following:

```
1. public static void main(String args[]) {
2.   OuterOne o = new OuterOne();
3.   OuterOne.InnerOne i = o.new InnerOne();
4.   i.innerMethod();
5. }
```

If you attempt to use the `new` operation to construct an instance of an inner class without a prefixing reference to an instance of the outer class, the implied prefix `this.` is assumed. This behavior is identical to that which you find with ordinary member accesses and method invocations. As with member access and method invocation, it is important that the `this` reference be valid when you try to use it. A `static` method contains no `this` reference, which is why you must take special efforts in these conditions.

Member Classes

To this point, we have not distinguished between classes defined directly in the scope of a class—that is, *member classes*—and classes defined inside of methods. There are important distinctions between these two scopes that you will need to have clear in your mind when you sit for the Certification Exam. First, we'll look at the features that are unique to member classes.

Access Modifiers

Members of a class, whether they are variables, methods, or nested classes, may be marked with modifiers that control access to those members. This means that member classes can be marked `private`, `public`, `protected`, or default access. The meaning of these access modifiers is the same for member classes as it is for other members, and therefore we won't spend time on those issues here. Instead, refer to Chapter 3, "Modifiers," if you need to revisit these concepts.

Static Inner Classes

Just like any other member, a member inner class may be marked `static`. When applied to a variable, `static` means that the variable is associated with the class, rather than with any particular instance of the class. When applied to an inner class, the meaning is similar. Specifically, a static inner class does *not* have any reference to an enclosing instance. As a result, methods of a static inner class cannot use the keyword `this` (either implied or explicit) to access instance variables of the enclosing class; those methods can, however, access static variables of the enclosing class. This is just the same as the rules that apply to static methods in ordinary classes. As you would expect, you can create an instance of a static inner class without the need for a current instance of the enclosing class. The syntax for this construction is very simple; just use the long name of the inner class—that is, the name that includes the name of the outer class, as in the highlighted part of line 5:

```
1. public class MyOuter {
2.   public static class MyInner {
3.   }
4.   public static void main(String [] args) {
5.     MyInner aMyInner = new MyOuter.MyInner();
6.   }
7. }
```

The net result is that a static inner class is really just a top-level class with a modified naming scheme. In fact, you can use static inner classes as an extension to packaging.

Not only can you declare a class inside another class, but you can also declare a class inside a method of another class. We will discuss this next.

Classes Defined Inside Methods

In the opening of this chapter, we said that nested classes can be declared in any block, and that this means you can define a class inside a method. This is superficially similar to what you have already seen, but in this case there are three particular points to be considered.

The first point is that anything declared inside a method is not a member of the class but is local to the method. The immediate consequence is that classes declared in methods are private to the method and cannot be marked with any access modifier; neither can they be marked as `static`. If you think about this, you'll recognize that these are just the same rules as for any variable declaration you might make in a method.

The second point is that an object created from an inner class within a method can have some access to the variables of the enclosing method. We'll look at how this is done and the restrictions that apply to this access in a moment.

Finally, it is possible to create an anonymous class—literally, a class with no specified name—and doing so can be very eloquent when working with event listeners. We will discuss this technique after covering the rules governing access from an inner class to method variables in the enclosing blocks.

Accessing Method Variables

The rule that governs access to the variables of an enclosing method is simple. Any variable, either a local variable or a formal parameter, can be accessed by methods within an inner class, provided that variable is marked `final`. A final variable is effectively a constant, so this might seem to be quite a severe restriction, but the point is simply this: an object created inside a method is likely to outlive the method invocation. Because local variables and method arguments are conventionally destroyed when their method exits, these variables would be invalid for access by inner class methods after the enclosing method exits. By allowing access only to final variables, it becomes possible to copy the values of those variables into the object, thereby extending their lifetimes. The other possible approaches to this problem would be writing to two copies of the same data every time it was changed or putting method local variables onto the heap instead of the stack. Either of these approaches would significantly degrade performance.

Let's look at an example:

```
 1. public class MOuter {
 2.   private int m = (int)(Math.random() * 100);
 3.   public static void main(String args[]) {
 4.     MOuter that = new MOuter();
 5.     that.go((int)(Math.random() * 100),
 6.       (int)(Math.random() * 100));
 7.   }
 8.
 9.   public void go(int x, final int y) {
10.     int a = x + y;
11.     final int b = x - y;
12.     class MInner {
13.       public void method() {
14.         System.out.println("m is " + m);
15. //        System.out.println("x is " + x); //Illegal!
16.         System.out.println("y is " + y);
17. //        System.out.println("a is " + a); //Illegal!
18.         System.out.println("b is " + b);
19.       }
20.         }
21.
```

```
22.     MInner that = new MInner();
23.     that.method();
24.   }
25. }
```

In this example, the class `MInner` is defined in lines 12–20 (in bold). Within it, `method()` has access to the member variable `m` in the enclosing class (as with the previous examples) but also to the final variables of `go()`. The commented-out code on lines 15 and 17 would be illegal, because it attempts to refer to nonfinal variables in `go()`; if these lines were included in the source proper, they would cause compiler errors.

Anonymous Classes

Some classes that you define inside a method do not need a name. A class defined in this way without a name is called an *anonymous class*. Anonymous classes can be declared to extend another class or to implement a single interface. The syntax does not allow you to do both at the same time, nor to implement more than one interface explicitly (of course, if you extend a class and the parent class implements interfaces, then so does the new class). If you declare a class that implements a single explicit interface, then it is a direct subclass of `java.lang.Object`.

Because you do not know the name of an anonymous inner class, you cannot use the `new` keyword in the usual way to create an instance of that class. In fact, the definition, construction, and first use (often in an assignment) of an anonymous class all occur in the same place. The next example shows a typical creation of an anonymous inner class that implements a single interface, in this case `ActionListener`. The essential parts of the declaration and construction are in bold on lines 3–7:

```
1. public void aMethod() {
2.   theButton.addActionListener(
3.     new ActionListener() {
4.       public void actionPerformed(ActionEvent e) {
5.         System.out.println("The action has occurred");
6.       }
7.     }
8.   );
9. }
```

In this fragment, the variable used at line 2, `theButton`, is a reference to a `Button` object. Notice that the action listener attached to the button is defined in lines 3–7. The entire declaration forms the argument to the `addActionListener()` method call at line 2; the closing parenthesis that completes this method call is on line 8.

The declaration and construction both start on line 3. Notice that the name of the interface is used immediately after the `new` keyword. This pattern is used for both interfaces and classes. The class has no visible name of its own in the source but is referred to simply using the class or interface name from which the new anonymous class is derived. The effect of this syntax is to state that you are defining a class and you do not want to think up a name for that class. Further, the class implements

the specified interface or extends the specified class without using the either the `implements` or `extends` keyword.

An anonymous class gives you a convenient way to avoid having to think up trivial names for classes, but the facility should be used with care. Clearly, you cannot instantiate objects of this class anywhere except in the code shown. Further, anonymous classes should be small. If the class defines methods other than those of a simple, well-known interface such as an AWT event listener, it probably should not be anonymous. Similarly, if the class has methods containing more than one or two lines of straightforward code or if the entire class has more than about 10 lines, it probably should not be anonymous. These are not absolute rules; rather, the point here is that if you do not give the class a name, you have only the "self-documenting" nature of the code to explain what it is for. If, in fact, the code is not simple enough to be genuinely self-documenting, then you probably should give it a descriptive name.

When the compiler comes across an anonymous inner class, it creates a separate class file for it called *EnclosingClassName*$*n*.class, where *EnclosingClassName* is the name of the class that contains the anonymous inner class, and *n* is the integer counter for the anonymous inner classes in the enclosing class (starting at 1).

Construction and Initialization of Anonymous Inner Classes

You need to understand a few points about the construction and initialization of anonymous inner classes to succeed in the Certification Exam and in real life. Let's look at these issues.

As you have already seen, the class is instantiated and declared in the same place. This means that anonymous inner classes are unique to method scopes; you cannot have anonymity with a member class.

You cannot define any specific constructor for an anonymous inner class. This is a direct consequence of the fact that you do not specify a name for the class, and therefore you cannot use that name to specify a constructor. However, an inner class can be constructed with arguments under some conditions, and an inner class can have an initializer if you wish.

Anonymous Class Declarations

As you have already seen, the central theme of the code that declares and constructs an anonymous inner class is

```
new Xxxx() { /* class body. */ }
```

where `Xxxx` is a class or interface name. It is important to grasp that code of this form is an *expression* that returns a reference to an object. Thus the previous code is incomplete by itself but can be used wherever you can use an object reference. For example, you might assign the reference to the constructed object into a variable, like this:

```
Xxxx AnXxxx = new Xxxx () { /* class body. */ };
```

Notice that you must be sure to make a complete statement, including the closing semicolon. Alternatively, you might use the reference to the constructed object as an argument to a method call. In that case, the overall appearance is like this:

```
someMethod(new Xxxx () { /* class body. */ });
```

Passing Arguments into the Construction of an Anonymous Inner Class

If the anonymous inner class extends another class, and that parent class has constructors that take arguments, then you can arrange for one of these constructors to be invoked by specifying the argument list to the construction of the anonymous inner class. An example follows:

```
// Assume this code appears in some method
Button b = new Button("Anonymous Button") {
  // behavior for the button
};
// do things with the button b...
...
```

In this situation, the compiler will build a constructor for your anonymous inner class that effectively invokes the superclass constructor with the argument list provided, something like this:

```
// This is not code you write! This exemplifies what the
// compiler creates internally when asked to compile
// something like the previous anonymous example
class AnonymousButtonSubclass extends Button {
  public AnonymousButtonSubclass(String s) {
    super(s);
  }
}
```

Note that this isn't the actual code that would be created—specifically, the class name is made up—but it conveys the general idea.

Initializing an Anonymous Inner Class

Sometimes you will want to perform some kind of initialization when an inner class is constructed. In normal classes, you would create a constructor. In an anonymous inner class, you cannot do this, but you can use the initializer feature that was added to the language at JDK 1.1. If you provide an unnamed block in class scope, then it will be invoked as part of the construction process, like this:

```
public MyClass {
  { // initializer
    System.out.println("Creating an instance");
  }
}
```

This is true of any class, but the technique is particularly useful with anonymous inner classes, where it is the only tool you have that provides some control over the construction process.

A Complex Example of Anonymous Inner Classes

Now let's look at a complete example following the pattern of the earlier example using a `Button`. This example uses two anonymous inner classes, one nested inside the other; an initializer; and a constructor that takes an argument:

```
 1. import java.awt.*;
 2. import java.awt.event.*;
 3.
 4. public class X extends Frame {
 5.   public static void main(String args[]) {
 6.     X x = new X();
 7.     x.pack();
 8.     x.setVisible(true);
 9.   }
10.
11.   private int count;
12.
13.   public X() {
14.     final Label l = new Label("Count = " + count);
15.     add(l, BorderLayout.SOUTH);
16.
17.     add(
18.       new Button("Hello " + 1) {
19.         { // initializer
20.           addActionListener(
21.             new ActionListener() {
22.               public void actionPerformed(
23.                 ActionEvent ev) {
24.                 count++;
25.                 l.setText("Count = " + count);
26.               }
27.             }
28.           );
29.         }
30.       }, BorderLayout.NORTH
31.     );
32.   }
33. }
```

Lines 19–29 form the initializer and set up a listener on the `Button`. The listener is another anonymous inner class; as we said earlier, you can arbitrarily nest these things. Notice how the label variable declared at line 14 is final; this allows it to be accessed from the inner classes, and specifically, from the listener defined in the initializer of the first anonymous inner class.

Summary

We have covered a lot of material in this chapter, but all of it is important. Let's look again at the key points.

We began by covering object-oriented design and implementation. The benefits of object-oriented design and implementation include reusability (through composition and inheritance) and data protection (through encapsulation). We discussed the concept of overloading methods, which allows the programmer to write several methods by the same name in the same class with different argument lists, return types, accessibility modifiers, and lists of exceptions to be thrown. We also discussed overriding methods, which allows the programmer to define new behavior in a subclass method that differs from that of the superclass method. Late binding ensures that the correct behavior is executed at runtime.

We defined when and how constructors are defined with respect to subclassing. Constructors are not inherited, but a single default constructor is provided by the compiler for all classes, including subclasses. A constructor in a subclass can call a constructor in its superclass (by using the `super()` reference) or another constructor in the same class (by using the `this()` reference).

Inner classes is an updated feature in the JDK 1.1 release. Because they can be declared in any scope, they can give your programs additional clarity and make them more concise. An inner class in class scope can have any accessibility, including private. Inner classes defined as local to a block may not be static. However, an inner class declared local to a block (for example, in a method), must not have any access modifier. Such a class is effectively private to the block. Classes defined in methods can be anonymous, in which case they must be instantiated at the same point they are defined. Anonymous inner classes can implement an interface or extend a class, but they cannot have any explicit constructors.

Exam Essentials

Be familiar with the way the Java language realizes the "is a" and "has a" relationships. The "is a" relationship implies class extension. The "has a" relationship implies ownership of a reference to a different object.

Be able to identify legally overloaded methods and constructors. The methods/constructors must have different argument lists.

Be able to identify legally overridden methods. The methods must have the same name, argument list, and return type.

Know the legal return types for overloaded and overridden methods. There are no restrictions for an overloaded method; an overriding method must have the same return type as the overridden version.

Know that the compiler generates a default constructor when a class has no explicit constructors. When a class has constructor code, no default constructor is generated.

Understand the chain of calls to parental constructors. Each constructor invocation begins by invoking a parental constructor.

Know how to create a constructor that invokes a nondefault parental constructor. Understand the use of the `super` keyword.

Be able to identify correctly constructed inner classes, including inner classes in methods and anonymous inner classes. The syntax for each of these forms is explained in previous sections of this chapter.

Know which data and methods of an enclosing class are available to an inner class. Understand that the inner class can access all data and methods of its enclosing class.

Understand the restrictions on static inner classes. Understand that a static inner class cannot access nonstatic features of its enclosing class.

Know how to use a nonstatic inner class from a static method of the enclosing class. Be able to recognize the `new Outer().new Inner()` format.

Key Terms

Before you take the exam, be certain you are familiar with the following terms:

anonymous class

default constructor

encapsulation

inheritance

inner class

late binding

member class

overloading

overriding

Review Questions

1. Consider this class:

```
1. public class Test1 {
2.   public float aMethod(float a, float b) {
3.   }
4.
5. }
```

Which of the following methods would be legal if added (individually) at line 4? (Choose all that apply.)

A. `public int aMethod(int a, int b) { }`

B. `public float aMethod(float a, float b) { }`

C. `public float aMethod(float a, float b, int c) throws Exception { }`

D. `public float aMethod(float c, float d) { }`

E. `private float aMethod(int a, int b, int c) { }`

2. Consider these classes, defined in separate source files:

```
1. public class Test1 {
2.   public float aMethod(float a, float b) throws
3.   IOException {...
4.   }
5. }
```

```
1. public class Test2 extends Test1 {
2.
3. }
```

Which of the following methods would be legal (individually) at line 2 in class `Test2`? (Choose all that apply.)

A. `float aMethod(float a, float b) {...}`

B. `public int aMethod(int a, int b) throws Exception {...}`

C. `public float aMethod(float a, float b) throws  Exception {...}`

D. `public float aMethod(float p, float q) {...}`

3. You have been given a design document for a veterinary registration system for implementation in Java. It states:

"A pet has an owner, a registration date, and a vaccination-due date. A cat is a pet that has a flag indicating whether it has been neutered, and a textual description of its markings."

Given that the `Pet` class has already been defined, which of the following fields would be appropriate for inclusion in the `Cat` class as members? (Choose all that apply.)

A. `Pet thePet;`

B. `Date registered;`

C. `Date vaccinationDue;`

D. `Cat theCat;`

E. `boolean neutered;`

F. `String markings;`

4. You have been given a design document for a veterinary registration system for implementation in Java. It states:

"A pet has an owner, a registration date, and a vaccination-due date. A cat is a pet that has a flag indicating if it has been neutered, and a textual description of its markings."

Given that the `Pet` class has already been defined and you expect the `Cat` class to be used freely throughout the application, how would you make the opening declaration of the `Cat` class, up to but not including the first opening brace? Use only these words and spaces: `boolean`, `Cat`, `class`, `Date`, `extends`, `Object`, `Owner`, `Pet`, `private`, `protected`, `public`, `String`.

A. `protected class Cat extends Owner`

B. `public class Cat extends Object`

C. `public class Cat extends Pet`

D. `private class Cat extends Pet`

5. Consider the following classes, declared in separate source files:

```
1. public class Base {
2.   public void method(int i) {
3.     System.out.print("Value is " + i);
4.   }
5. }
1. public class Sub extends Base {
2.   public void method(int j) {
3.     System.out.print("This value is " + j);
4.   }
5.   public void method(String s) {
```

```
 6.     System.out.print("I was passed " + s);
 7.   }
 8.   public static void main(String args[]) {
 9.     Base b1 = new Base();
10.     Base b2 = new Sub();
11.     b1.method(5);
12.     b2.method(6);
13.   }
14. }
```

What output results when the main method of the class `Sub` is run?

A. `Value is 5Value is 6`

B. `This value is 5This value is 6`

C. `Value is 5This value is 6`

D. `This value is 5Value is 6`

E. `I was passed 5I was passed 6`

6. Consider the following class definition:

```
1. public class Test extends Base {
2.   public Test(int j) {
3.   }
4.   public Test(int j, int k) {
5.     super(j, k);
6.   }
7. }
```

Which of the following are legitimate calls to construct instances of the `Test` class? (Choose all that apply.)

A. `Test t = new Test();`

B. `Test t = new Test(1);`

C. `Test t = new Test(1, 2);`

D. `Test t = new Test(1, 2, 3);`

E. `Test t = (new Base()).new Test(1);`

7. Consider the following class definition:

```
1. public class Test extends Base {
2.   public Test(int j) {
3.   }
```

```
4.   public Test(int j, int k) {
5.     super(j, k);
6.   }
7. }
```

Which of the following forms of constructor must exist explicitly in the definition of the `Base` class? Assume `Test` and `Base` are in the same package. (Choose all that apply.)

A. `Base() { }`

B. `Base(int j) { }`

C. `Base(int j, int k) { }`

D. `Base(int j, int k, int l) { }`

8. Which of the following statements are true? (Choose all that apply.)

A. An inner class may be declared `private`.

B. An inner class may be declared `static`.

C. An inner class defined in a method should always be anonymous.

D. An inner class defined in a method can access all the method local variables.

E. Construction of an inner class may require an instance of the outer class.

9. Consider the following definition:

```
1. public class Outer {
2.   public int a = 1;
3.   private int b = 2;
4.   public void method(final int c) {
5.     int d = 3;
6.     class Inner {
7.       private void iMethod(int e) {
8.
9.       }
10.    }
11.  }
12. }
```

Which variables can be referenced correctly at line 8? (Choose all that apply.)

A. `a`

B. `b`

C. `c`

D. `d`

E. `e`

10. Which of the following statements are true? (Choose all that apply.)

A. Given that `Inner` is a nonstatic class declared inside a public class `Outer` and that appropriate constructor forms are defined, an instance of `Inner` can be constructed like this: `new Outer().new Inner()`

B. If an anonymous inner class inside the class `Outer` is defined to implement the interface `ActionListener`, it can be constructed like this: `new Outer().new ActionListener()`

C. Given that `Inner` is a nonstatic class declared inside a public class `Outer` and that appropriate constructor forms are defined, an instance of `Inner` can be constructed in a static method like this: `new Inner()`

D. An anonymous class instance that implements the interface `MyInterface` can be constructed and returned from a method like this:

```
1. return new MyInterface(int x) {
2.   int x;
3.   public MyInterface(int x) {
4.     this.x = x;
5.   }
6. };
```

Answers to Review Questions

1. A, C, E. In each of these answers, the argument list differs from the original, so the method is an overload. Overloaded methods are effectively independent, and there are no constraints on the accessibility, return type, or exceptions that may be thrown. B would be a legal overriding method, except that it cannot be defined in the same class as the original method; rather, it must be declared in a subclass. D is also an override, because the *types* of its arguments are the same: changing the parameter names is not sufficient to count as overloading.

2. B, D. A is illegal because it is less accessible than the original method; the fact that it throws no exceptions is perfectly acceptable. B is legal because it overloads the method of the parent class, and as such it is not constrained by any rules governing its return value, accessibility, or argument list. The exception thrown by C is sufficient to make that method illegal. D is legal because the accessibility and return type are identical, and the method is an override because the types of the arguments are identical—remember that the names of the arguments are irrelevant. The absence of an exception list in D is not a problem: An overriding method may legitimately throw fewer exceptions than its original, but it may not throw more.

3. E, F. The `Cat` class is a subclass of the `Pet` class, and as such should extend `Pet`, rather than containing an instance of `Pet`. B and C should be members of the `Pet` class and as such are inherited into the `Cat` class; therefore, they should not be declared in the `Cat` class. D would declare a reference to an instance of the `Cat` class, which is not generally appropriate inside the `Cat` class (unless, perhaps, you were asked to give the `Cat` a member that refers to its mother). Finally, the `neutered` flag and markings descriptions, E and F, are the items called for by the specification; these are correct items.

4. C. The class should be public, because it is to be used freely throughout the application. The statement "A cat is a pet" tells you that the `Cat` class should subclass `Pet`. The other words offered are required for the body of the definitions of either `Cat` or `Pet`—for use as member variables—but are not part of the opening declaration.

5. C. The first message is produced by the `Base` class when `b1.method(5)` is called and is therefore `Value is 5`. Despite the fact that variable `b2` is declared as being of the `Base` class, the behavior that results when `method()` is invoked upon it is the behavior associated with the class of the actual object, not with the type of the variable. Because the object is of class `Sub`, not of class `Base`, the second message is generated by line 3 of class `Sub`: `This value is 6`.

6. B, C. Because the class has explicit constructors defined, the default constructor is suppressed, so A is not possible. B and C have argument lists that match the constructors defined at lines 2 and 4 respectively, and so are correct constructions. D has three integer arguments, but there are no constructors that take three arguments of any kind in the `Test` class, so D is incorrect. Finally, E is a syntax used for construction of inner classes and is therefore wrong.

7. A, C. The constructor at lines 2 and 3 includes no explicit call to either `this()` or `super()`, which means that the compiler will generate a call to the zero-argument superclass constructor, as in A. The explicit call to `super()` at line 5 requires that the `Base` class must have a constructor as in C. This requirement has two consequences. First, C must be one of the required constructors and therefore one of the answers. Second, the `Base` class must have at least that constructor defined explicitly, so the default constructor is not generated, but must be added explicitly. Therefore the constructor of A is also required and must be a correct answer. At no point in the `Test` class is there a call to either a superclass constructor with one or three arguments, so B and D need not explicitly exist.

8. A, B, E. Member inner classes may be defined with any accessibility, so `private` is entirely acceptable and A is correct. Similarly, the `static` modifier is permitted on a member inner class, which causes it not to be associated with any particular instance of the outer class; thus B is also correct. Inner classes defined in methods may be anonymous—and indeed often are—but this is not required, so C is wrong. D is wrong because it is not possible for an inner class defined in a method to access the local variables of the method, except for those variables that are marked as `final`. Constructing an instance of a static inner class does not need an instance of the enclosing object, but all nonstatic inner classes do require such a reference, and that reference must be available to the `new` operation. The reference to the enclosing object is commonly implied as `this`, which is why it is commonly not explicit. These points make E true.

9. A, B, C, E. Because `Inner` is not a static inner class, it has a reference to an enclosing object, and all the variables of that object are accessible. Therefore A and B are correct, despite the fact that `b` is marked `private`. Variables in the enclosing method are accessible only if those variables are marked `final`, so the method argument `c` is correct, but the variable `d` is not. Finally, the parameter `e` is of course accessible, because it is a parameter to the method containing line 8.

10. A. Construction of a normal (that is, a named and nonstatic) inner class requires an instance of the enclosing class. Often this enclosing instance is provided via the implied `this` reference, but an explicit reference can be used in front of the `new` operator, as shown in A. Anonymous inner classes can be instantiated only at the same point they are declared, like this:

```
return new ActionListener() {
  public void actionPerformed(ActionEvent e) { }
}
```

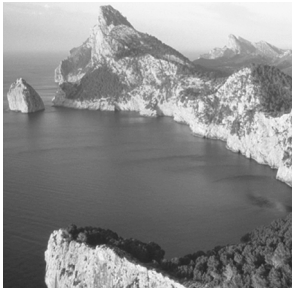
Hence, B is illegal; it attempts to instantiate the interface `ActionListener` as if that interface were an inner class inside `Outer`. C is illegal because `Inner` is a nonstatic inner class, and so it requires a reference to an enclosing instance when it is constructed. The form shown suggests the implied `this` reference, but because the method is `static`, there is no `this` reference and the construction is illegal. D is illegal because it attempts to use arguments to the constructor of an anonymous inner class that implements an interface. The clue is in the attempt to define a constructor at line 3. This would be a constructor for the interface `MyInterface`, not for the inner class—this is wrong on two counts. First, interfaces do not define constructors, and second, you need a constructor for your anonymous class, not for the interface.

Chapter 7

Threads

JAVA CERTIFICATION EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- ✓ **4.2 Identify classes that correctly implement an interface where that interface is either** `java.lang.Runnable` **or a fully specified interface in the question.**
- ✓ **7.1 Write code to define, instantiate, and start new threads using both** `java.lang.Thread` **and** `java.lang.Runnable`**. Recognize conditions that might prevent a thread from executing.**
- ✓ **7.2 Write code using synchronized** `wait`**,** `notify`**, and** `notifyAll` **to protect against concurrent access problems and to communicate between threads.**
- ✓ **7.3 Define the interaction among threads and object locks when executing synchronized** `wait`**,** `notify`**, or** `notifyAll`**.**
- ✓ **7.4 Define the interaction among threads and object locks when executing synchronized** `wait`**,** `notify` **or** `notifyAll`**.**



Threads are Java's way of making a single Java Virtual Machine (JVM) look like many machines, all running at the same time. This effect, usually, is an illusion: there is only one JVM and usually only one CPU, but the CPU switches among the JVM's various threads to give the impression that there are multiple CPUs. JVM threads work behind the scenes on your behalf, listening for user input, managing garbage collection, and performing a variety of other tasks.

As a Java programmer, you can choose between a *single-threaded* and a *multithreaded* programming paradigm. A single-threaded Java program has one entry point (the `main()` method) and one exit point. All instructions are run serially, from start to finish. A multithreaded program has a *first* entry point (the `main()` method), followed by multiple entry and exit points for other methods that may be scheduled to run concurrently with the `main()` method.

Java provides you with tools for creating and managing threads. Threads are valuable tools for allowing unrelated, loosely related, or tightly related work to be programmed separately and executed concurrently.

The Certification Exam objectives require that you be familiar with Java's thread support, including the mechanisms for creating, controlling, and communicating between threads.

Thread Fundamentals

Java's thread support resides in three places:

- The `java.lang.Thread` class
- The `java.lang.Object` class
- The Java language and JVM

Most (but definitely not all) support resides in the `Thread` class. In Java, every thread corresponds to an instance of the `Thread` class. These objects can be in various states: at any moment, at most one object is executing per CPU, while others might be waiting for resources, waiting for a chance to execute, sleeping, or dead.

In order to demonstrate an understanding of threads, you need to be able to answer a few questions:

- When a thread executes, what code does it execute?
- What states can a thread be in?
- How does a thread's state get changed?

The next few sections will look at each of these questions in turn.

What a Thread Executes

To make a thread execute, you call its `start()` method. Doing so registers the thread with a piece of system code called the *thread scheduler*. The scheduler might be part of the JVM or of the host operating system. The scheduler determines which thread is running on each available CPU at any given time. Note that calling your thread's `start()` method doesn't immediately cause the thread to run; it just makes the thread *eligible* to run. The thread must still contend for CPU time with all the other threads. If all is well, then at some point in the future the thread scheduler will permit your thread to execute.

During its lifetime, a thread spends some time executing and some time in any of several non-executing states. In this section, you can ignore (for the moment) the question of how the thread is moved between states. The question at hand is this: When the thread gets to execute, what does it execute?

The simple answer is that it executes a method called `run()`. But which object's `run()` method? You have two choices:

- The thread can execute its own `run()` method.
- The thread can execute the `run()` method of some other object.

If you want the thread to execute its own `run()` method, you need to subclass the `Thread` class and implement the `run()` method. For example:

```
1. public class CounterThread extends Thread {
2.   public void run() {
3.     for (int i = 1; i <= 10; i++) {
4.       System.out.println("Counting: " + i);
5.     }
6.   }
7. }
```

This `run()` method prints out the numbers from 1 to 10. To do this in a thread, you first construct an instance of `CounterThread` and then invoke its `start()` method:

```
1. CounterThread ct = new CounterThread();
2. ct.start();      // start(), not run()
```

What you *don't* do is call `run()` directly; that would just count to 10 in the current thread. Instead, you call `start()`, which the `CounterThread` class inherits from its parent class, `Thread`. The `start()` method registers the thread `ct` with the thread scheduler; eventually the thread will execute, and at that time its `run()` method will be called.

If you want your thread to execute the `run()` method of some object other than itself, you still need to construct an instance of the `Thread` class. The only difference is that when you call the `Thread` constructor, you have to specify which object owns the `run()` method that you want. To do this, you invoke an alternate form of the `Thread` constructor:

```
public Thread(Runnable target)
```

The Runnable interface describes a single method:

```
public void run();
```

Thus you can pass any object you want into the Thread constructor, provided it implements the Runnable interface (so that it really does have a run() method for the thread scheduler to invoke).

Having constructed an instance of Thread, you proceed as before: you invoke the start() method. As before, doing so registers the thread with the scheduler, and eventually the run() method of the target will be called.

For example, the following class has a run() method that counts down from 10 to 1:

```
1. public class DownCounter implements Runnable {
2.   public void run() {
3.     for (int i = 10; i >= 1; i--) {
4.       System.out.println("Counting Down: " + i);
5.     }
6.   }
7. }
```

This class does not extend Thread. However, it has a run() method, and it declares that it implements the Runnable interface. Thus any instance of the DownCounter class is eligible to be passed into the alternative (nondefault) constructor for Thread:

```
1. DownCounter dc = new DownCounter();
2. Thread t = new Thread(dc);
3. t.start();
```

This section has presented two strategies for constructing threads. Superficially, the only difference between these two strategies is the location of the run() method. The second strategy, where a runnable target is passed into the constructor, is perhaps a bit more complicated in the case of the simple examples we have considered. However, there are good reasons why you might choose to make this extra effort. The run() method, like any other member method, is allowed to access the private data, and call the private methods, of the class of which it is a member. Putting run() in a subclass of Thread may mean that the method cannot access features it needs (or cannot access those features in a clean, reasonable manner).

Another reason that might persuade you to implement your threads using runnables rather than subclassing Thread is the single implementation inheritance rule. If you write a subclass of Thread, it cannot be a subclass of anything else; but using Runnable, you can subclass whatever other parent class you choose.

Finally, from an object-oriented point of view, a subclass of Thread combines two unrelated functionalities: support for multithreading inherited from the Thread superclass, and execution behavior provided by the run() method. These functionalities are not closely related, so good object-oriented discipline suggests that they exist in two separate classes. In the jargon of object-oriented analysis, if you create a class that extends Thread, you're saying that your class "is a" thread. If you create a class that implements Runnable, you're saying that your class "is associated with" a thread.

To summarize, you can use two approaches to specify which `run()` method will be executed by a thread:

- Subclass `Thread`. Define your `run()` method in the subclass.
- Write a class that implements `Runnable`. Define your `run()` method in that class. Pass an instance of that class into your call to the `Thread` constructor.

When Execution Ends

When the `run()` method returns, the thread has finished its task and is considered *dead*. There is no way out of this state. Once a thread is dead, it cannot be started again; if you want the thread's task to be performed again, you have to construct and start a new thread instance. The dead thread continues to exist; it is an object like any other object, and you can still access its data and call its methods. You just can't make it run again. In other words:

- You *can't* restart a dead thread by calling it's `start()` or `run()` methods.
- You *can* call other methods (besides `start()` and `run()`) of a dead thread.

The `Thread` methods include a method called `stop()`, which forcibly terminates a thread, putting it into the dead state. This method is deprecated since JDK 1.2, because it can cause data corruption or deadlock if you kill a thread that is in a critical section of code. The `stop()` method is therefore no longer part of the Certification Exam. Instead of using `stop()`, if a thread might need to be killed from another thread, you should call `interrupt()` on it from the killing method.

Although you can't restart a dead thread, if you use runnables, you can submit the old `Runnable` instance to a new thread. However, it is generally poor design to constantly create, use, and discard threads, because constructing a `Thread` is a relatively heavyweight operation, involving significant kernel resources. It is better to create a pool of reusable worker threads that can be assigned chores as needed.

Thread States

When you call `start()` on a thread, the thread does not run immediately. It goes into a "ready-to-run" state and stays there until the scheduler moves it to the "running" state. Then the `run()` method is called. In the course of executing `run()`, the thread may temporarily give up the CPU and enter some other state for a while. It is important to be aware of the possible states a thread might be in and of the triggers that can cause the thread's state to change.

The thread states are

Running The state that all threads aspire to

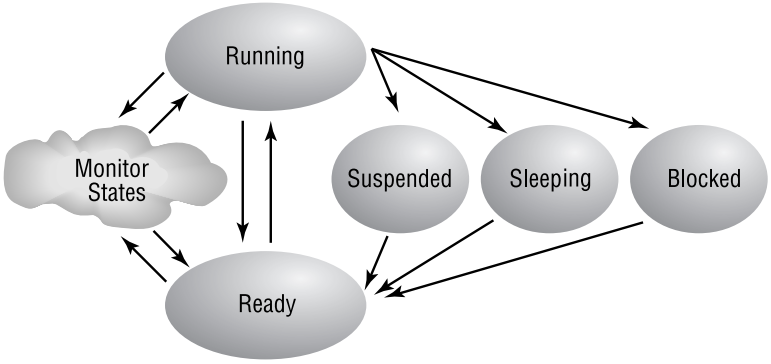
Various nonrunning states Monitor states, Sleeping, Suspended, Blocked

Ready Not waiting for anything except the CPU

Dead All done

Figure 7.1 shows the nondead states. Notice that the figure does not show the dead state.

FIGURE 7.1 Living thread states



At the top of Figure 7.1 is the Running state. At the bottom is the Ready state. In between are the various not-ready states. A thread in one of these intermediate states is waiting for something to happen; when that something eventually happens, the thread moves to the Ready state, and eventually the thread scheduler will permit it to run again. Note that the methods associated with the Suspended state are now deprecated; you will not be tested on this state or its associated methods in the exam. For this reason, we will not discuss them in any detail in this book.

The arrows between the bubbles in Figure 7.1 represent state transitions. Be aware that only the thread scheduler can move a ready thread into the CPU.

Later in this chapter, you will examine in detail the various waiting states. For now, the important thing to observe in Figure 7.1 is the general flow: a running thread enters an intermediate state for some reason; later, whatever the thread was waiting for comes to pass, and the thread enters the Ready state; later still, the scheduler grants the CPU to the thread. The exceptions to this general flow involve synchronized code and the `wait()` / `notify()` sequence—the corresponding portion of Figure 7.1 is depicted as a vague bubble labeled "Monitor States." These monitor states are discussed later in this chapter, in the section "Monitors, `wait()`, and `notify()`."

Thread Priorities

Every thread has a *priority*, which is an integer from 1 to 10; threads with higher priority should get preference over threads with lower priority. The priority is considered by the thread scheduler when it decides which ready thread should execute. The scheduler generally chooses the highest-priority

waiting thread. If more than one thread is waiting, the scheduler chooses one of them. There is no guarantee that the thread chosen will be the one that has been waiting the longest.

The default priority is 5, but all newly created threads have their priority set to that of the creating thread. To set a thread's priority, call the `setPriority()` method, passing in the desired new priority. The `getPriority()` method returns a thread's priority. The following code fragment increments the priority of thread `theThread`, provided the priority is less than 10. Instead of hard-coding the value 10, the fragment uses the constant `MAX_PRIORITY`. The `Thread` class also defines constants for `MIN_PRIORITY` (which is 1), and `NORM_PRIORITY` (which is 5).

```
1. int oldPriority = theThread.getPriority();
2. int newPriority = Math.min(oldPriority+1,
3.   Thread.MAX_PRIORITY);
4. theThread.setPriority(newPriority);
```

The specifics of how thread priorities affect scheduling are platform dependent. The Java specification states that threads must have priorities, but it does not dictate precisely what the scheduler should do about priorities. This vagueness is a problem: algorithms that rely on manipulating thread priorities might not run consistently on all platforms.

Controlling Threads

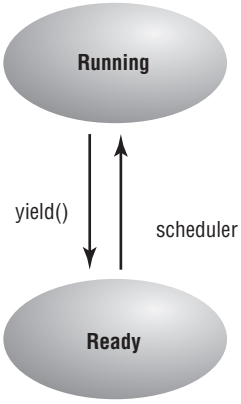
Thread control is the art of moving threads from state to state. You control threads by triggering state transitions. This section examines the various pathways out of the Running state. These pathways are

- Yielding
- Suspending and then resuming
- Sleeping and then waking up
- Blocking and then continuing
- Waiting and then being notified

Yielding

A thread can offer to move out of the virtual CPU by *yielding*. A call to the `yield()` method causes the currently executing thread to move to the Ready state if the scheduler is willing to run any other thread in place of the yielding thread. The state transition is shown in Figure 7.2.

FIGURE 7.2 Yield



A thread that has yielded goes into the Ready state. There are two possible scenarios. If any other threads are in the Ready state, then the thread that just yielded might have to wait a while before it gets to execute again. However, if no other threads are waiting, then the thread that just yielded will get to continue executing immediately. Note that most schedulers do not stop the yielding thread from running in favor of a thread of lower priority.

The `yield()` method is a static method of the `Thread` class. It always causes the currently executing thread to yield.

Yielding allows a time-consuming thread to permit other threads to execute. For example, consider an applet that computes a 300×300 pixel image using a ray-tracing algorithm. The applet might have a Compute button and an Interrupt button. The action event handler for the Compute button would create and start a separate thread, which would call a `traceRays()` method. A first cut at this method might look like this:

```
1. private void traceRays() {
2.   for (int j = 0; j < 300; j++) {
3.     for (int i = 0; i < 300; i++) {
4.       computeOnePixel(i, j);
5.     }
6.   }
7. }
```

There are 90,000 pixel color values to compute. If it takes 0.1 second to compute the color value of one pixel, then it will take two and a half hours to compute the complete image.

Suppose after half an hour the user looks at the partial image and realizes that something is wrong. (Perhaps the viewpoint or zoom factor is incorrect.) The user will then click the Interrupt button, because there is no sense in continuing to compute the useless image. Unfortunately, the thread that handles GUI input might not get a chance to execute until the thread that is executing `traceRays()` gives up the CPU. Thus the Interrupt button will not have any effect for another two hours.

If priorities are implemented meaningfully in the scheduler, then lowering the priority of the ray-tracing thread will have the desired effect, ensuring that the GUI thread will run when it has something useful to do. However, this mechanism is not reliable between platforms (although it is a good course of action anyway, because it will do no harm). The reliable approach is to have the ray-tracing thread periodically yield. If no input is pending when the yield is executed, then the ray-tracing thread will not be moved off the CPU. If, on the other hand, there is input to be processed, the input-listening thread will get a chance to execute.

The ray-tracing thread can have its priority set like this:

```
rayTraceThread.setPriority(Thread.NORM_PRIORITY-1);
```

The `traceRays()` method listed earlier can yield after each pixel value is computed, after line 4. The revised version looks like this:

```
1. private void traceRays() {
2.   for (int j = 0; j < 300; j++) {
3.     for (int i = 0; i < 300; i++) {
4.       computeOnePixel(i, j);
5.       Thread.yield();
6.     }
7.   }
8. }
```

Suspending

Suspending a thread is a mechanism that allows any arbitrary thread to make another thread unready for an indefinite period of time. The suspended thread becomes ready when some other thread resumes it. This might feel like a useful technique, but it is very easy to cause deadlock in a program using these methods—a thread has no control over when it is suspended (the control comes from outside the thread) and it might be in a critical section, holding an object lock at the time. The exact effect of `suspend()` and `resume()` is much better implemented using `wait()` and `notify()`.

The `suspend()` and `resume()` methods are deprecated as of the Java 2 release and do not appear in the Certification Exam, so we will not discuss them any further.

Sleeping

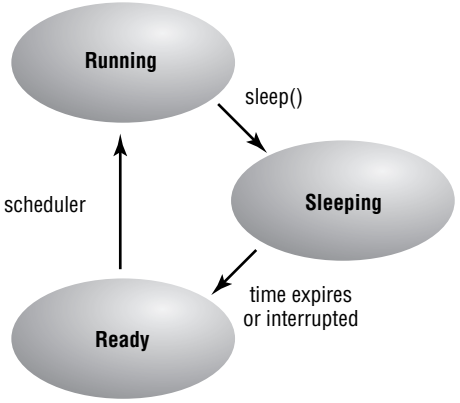
A *sleeping* thread passes time without doing anything and without using the CPU. A call to the `sleep()` method requests the currently executing thread to cease executing for (approximately) a specified amount of time. You can call this method two ways, depending on whether you want to specify the sleep period to millisecond precision or to nanosecond precision:

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

`sleep()`, like `yield()`, is static. Both methods operate on the currently executing thread.

The state diagram for Sleeping is shown in Figure 7.3. Notice that when the thread has finished sleeping, it does not continue execution. As you would expect, it enters the Ready state and will execute only when the thread scheduler allows it to do so. For this reason, you should expect that a `sleep()` call will block a thread for at least the requested time, but it might block for much longer. This behavior suggests that you should give very careful thought to your design before you expect any meaning from the nanosecond accuracy version of the `sleep()` method.

FIGURE 7.3 The Sleeping state



The `Thread` class has a method called `interrupt()`. A sleeping thread that receives an `interrupt()` call moves immediately into the Ready state; when it gets to run, it will execute its `InterruptedException` handler.

Blocking

Many methods that perform input or output have to wait for some occurrence in the outside world before they can proceed; this behavior is known as *blocking*. A good example is reading from a socket:

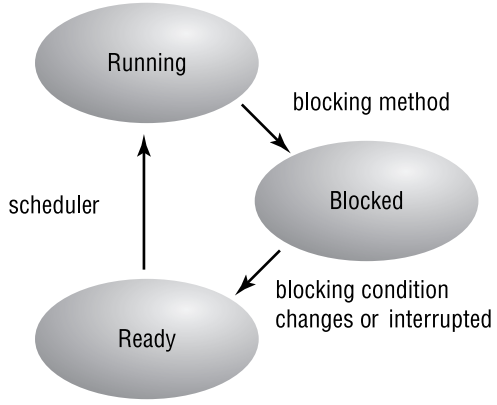
```
1. try {
2.   Socket sock = new Socket("magnesium", 5505);
3.   InputStream istr = sock.getInputStream();
4.   int b = istr.read();
5. }
```

```
6. catch (IOException ex) {
7.   // Handle the exception
8. }
```

It looks like line 4 reads a byte from an input stream that is connected to port 5505 on a machine called "magnesium." Actually, line 4 *tries* to read a byte. If a byte is available (that is, if magnesium has previously written a byte), then line 4 can return immediately and execution can continue. If magnesium has not yet written anything, however, the `read()` call has to wait. If magnesium is busy doing other things and takes half an hour to get around to writing a byte, then the `read()` call has to wait for half an hour.

Clearly, it would be a serious problem if the thread executing the `read()` call on line 4 remained in the Running state for the entire half hour. Nothing else could get done. In general, if a method needs to wait an indeterminable amount of time until some I/O occurrence takes place, then a thread executing that method should graciously step out of the Running state. All Java I/O methods behave this way. A thread that has graciously stepped out in this fashion is said to be *blocked*. Figure 7.4 shows the transitions of the Blocked state.

FIGURE 7.4 The Blocked state



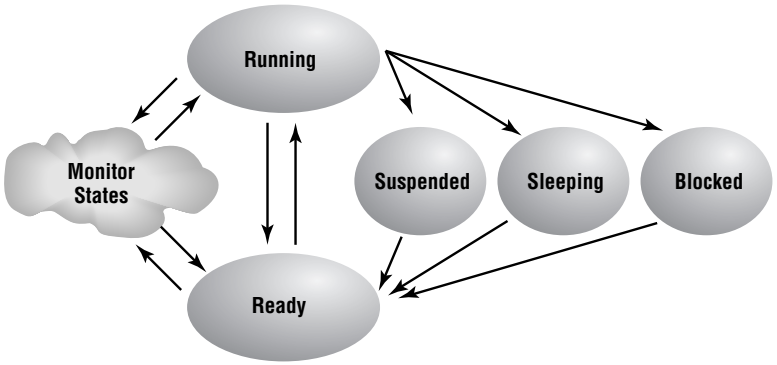
In general, if you see a method with a name that suggests that it might do nothing until something becomes ready—for example, `waitForInput()` or `waitForImages()`—you should expect that the caller thread might be blocked, thus losing the CPU, when the method is called. You do not need to know about all APIs to make this assumption; this is a general principle of APIs, both core and third party, in a Java environment.

A thread can also become blocked if it fails to acquire the lock for a monitor or if it issues a `wait()` call. Locks and monitors are explained in detail later in this chapter, beginning in the section "Monitors, `wait()`, and `notify()`." Internally, most blocking for I/O, like the `read()` calls just discussed, is implemented using `wait()` and `notify()` calls.

Monitor States

Figure 7.5 (which is a rerun of Figure 7.1) shows all the thread-state transitions. The intermediate states on the right side of the figure (Suspended, Sleeping, and Blocked) have been discussed in previous sections. The monitor states are drawn all alone on the left side of the figure to emphasize that they are very different from the other intermediate states.

FIGURE 7.5 Thread states (reprise)



The `wait()` method puts an executing thread into the *Waiting* state, and the `notify()` and `notifyAll()` methods move waiting threads out of the Waiting state. However, these methods are very different from `suspend()`, `resume()`, and `yield()`. For one thing, they are implemented in the `Object` class, not in `Thread`. For another, they can only be called in synchronized code. The Waiting state and its associated issues and subtleties are discussed in the final sections of this chapter. But first, let's look at one more topic concerning thread control.

Scheduling Implementations

Historically, two approaches have emerged for implementing thread schedulers:

- *Preemptive scheduling*
- *Time-sliced scheduling* or *round-robin scheduling*

So far, the facilities described in this chapter have been preemptive. In preemptive scheduling, there are only two ways for a thread to leave the Running state without explicitly calling a thread-scheduling method such as `wait()` or `suspend()`:

- It can cease to be ready to execute (by calling a blocking I/O method, for example).
- It can get moved out of the CPU by a higher-priority thread that becomes ready to execute.

With time-slicing, a thread is allowed to execute only for a limited amount of time. It is then moved to the Ready state, where it must contend with all the other ready threads. Time-slicing ensures against the possibility of a single high-priority thread getting into the Running state and

never getting out, preventing all other threads from doing their jobs. Unfortunately, time-slicing creates a nondeterministic system; you can't be certain at any moment which thread is executing or for how long it will continue to execute.

It is natural to ask which implementation Java uses. The answer is that it depends on the platform; the Java specification gives implementations a lot of leeway.

Monitors, *wait()*, and *notify()*

A *monitor* is an object that can block and revive threads. The concept is simple, but it takes a bit of work to understand what monitors are good for and how to use them effectively.

The reason for having monitors is that sometimes a thread cannot perform its job until an object reaches a certain state. For example, consider a class that handles requests to write to standard output:

```
1. class Mailbox {
2.   public boolean    request;
3.   public String     message;
4. }
```

The intention of this class is that a client can set `message` to some value, and then set `request` to true:

```
1. myMailbox.message = "Hello everybody.";
2. myMailbox.request = true;
```

There must be a thread that checks `request`; on finding it true, the thread should write `message` to `System.out`, and then set `request` to false. (Setting `request` to false indicates that the mailbox object is ready to handle another request.) It is tempting to implement the thread like this:

```
 1. public class Consumer extends Thread {
 2.   private Mailbox myMailbox;
 3.
 4.   public Consumer(Mailbox box) {
 5.     this.myMailbox = box;
 6.   }
 7.
 8.   public void run() {
 9.     while (true) {
10.       if (myMailbox.request) {
```

```
11.         System.out.println(myMailbox.message);
12.         myMailbox.request = false;
13.       }
14.
15.       try {
16.         sleep(50);
17.       }
18.       catch (InterruptedException e) { }
19.     }
20.   }
```

The consumer thread loops forever, checking for requests every 50 milliseconds. If there is a request (line 10), the consumer writes the message to standard output (line 11) and then sets `request` to false to show that it is ready for more requests.

The `Consumer` class may look fine at first glance, but it has two serious problems:

- The `Consumer` class accesses data internal to the `Mailbox` class, introducing the possibility of corruption. On a time-sliced system, the consumer thread could just possibly be interrupted between lines 10 and 11. The interrupting thread could just possibly be a client that sets `message` to its own message (ignoring the convention of checking `request` to see if the handler is available). The consumer thread would send the wrong message.
- The choice of 50 milliseconds for the delay can never be ideal. Sometimes 50 milliseconds will be too long, and clients will receive slow service; sometimes 50 milliseconds will be too frequent, and cycles will be wasted. A thread that wants to send a message has a similar dilemma if it finds the `request` flag set: the thread should back off for a while, but for how long? There is no good answer to this question.

Ideally, these problems would be solved by making some modifications to the `Mailbox` class:

- The mailbox should be able to protect its data from irresponsible clients.
- If the mailbox is not available—that is, if the `request` flag is already set—then a client consumer should not have to guess how long to wait before checking the flag again. The handler should tell the client when the time is right.

Java's monitor support addresses these issues by providing the following resources:

- A lock for each object
- The `synchronized` keyword for accessing an object's lock
- The `wait()`, `notify()`, and `notifyAll()` methods, which allow the object to control client threads

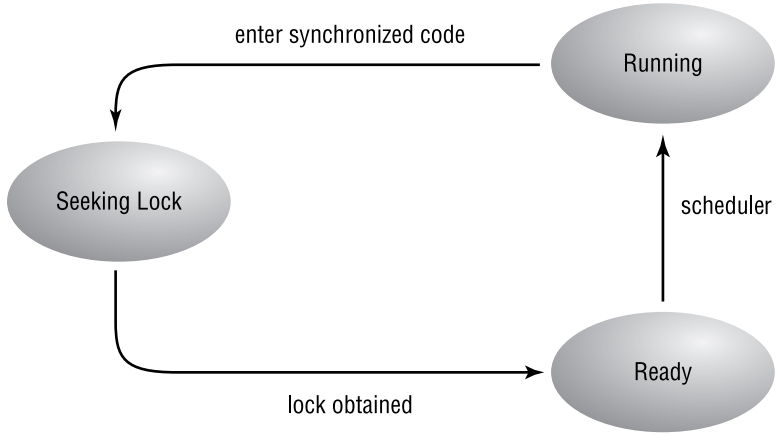
The following sections describe locks, synchronized code, and the `wait()`, `notify()`, and `notifyAll()` methods and show how these can be used to make thread code more robust.

The Object Lock and Synchronization

Every object has a *lock*. At any moment, that lock is controlled by, at most, one single thread. The lock controls access to the object's *synchronized code*. A thread that wants to execute an object's synchronized code must first attempt to acquire that object's lock. If the lock is available—that is, if it is not already controlled by another thread—then all is well. If the lock is under another thread's control, then the attempting thread goes into the Seeking Lock state and becomes ready only when the lock becomes available. When a thread that owns a lock passes out of the synchronized code, the thread automatically gives up the lock. All this lock-checking and state-changing is done behind the scenes; the only explicit programming you need to do is to declare code to be synchronized.

Figure 7.6 shows the Seeking Lock state. This figure is the first state in our expansion of the monitor states, as depicted in Figure 7.5.

FIGURE 7.6 The Seeking Lock state



You can mark code as synchronized two ways:

- Synchronize an entire method by putting the `synchronized` modifier in the method's declaration. To execute the method, a thread must acquire the lock of the object that owns the method.
- Synchronize a subset of a method by surrounding the desired lines of code with curly brackets ({}) and inserting the expression `synchronized(someObject)` before the opening curly. This technique allows you to synchronize the block on the lock of any object at all, not necessarily the object that owns the code.

The first technique is by far the more common; synchronizing on any object other than the object that owns the synchronized code can be extremely dangerous. The Certification Exam

requires you to know how to apply the second technique, but the exam does not make you think through complicated scenarios of synchronizing on external objects. The second technique is discussed at the end of this chapter.

Synchronization makes it easy to clean up some of the problems with the `Mailbox` class:

```
 1. class Mailbox {
 2.   private boolean   request;
 3.   private String    message;
 4.
 5.   public synchronized void
 6.   storeMessage(String message) {
 7.     request = true;
 8.     this.message = message;
 9.   }
10.
11.   public synchronized String retrieveMessage() {
12.     request = false;
13.     return message;
14.   }
15. }
```

Now the `request` flag and the message string are private, so they can be modified only via the public methods of the class. Because `storeMessage()` and `retrieveMessage()` are synchronized, there is no danger of a message-producing thread corrupting the flag and spoiling things for a message-consuming thread, or vice versa.

The `Mailbox` class is now safe from its clients, but the clients still have problems. A message-producing client should call `storeMessage()` only when the `request` flag is false; a message-consuming client should call `retrieveMessage()` only when the `request` flag is true. In the `Consumer` class of the previous section, the consuming thread's main loop polled the `request` flag every 50 milliseconds. (Presumably a message-producing thread would do something similar.) Now the `request` flag is private, so you must find another way.

It is possible to come up with any number of clever ways for the client threads to poll the mailbox, but the whole approach is backward. The mailbox becomes available or unavailable based on changes of its own state. The mailbox should be in charge of the progress of the clients. Java's `wait()` and `notify()` methods provide the necessary controls, as you will see in the next section.

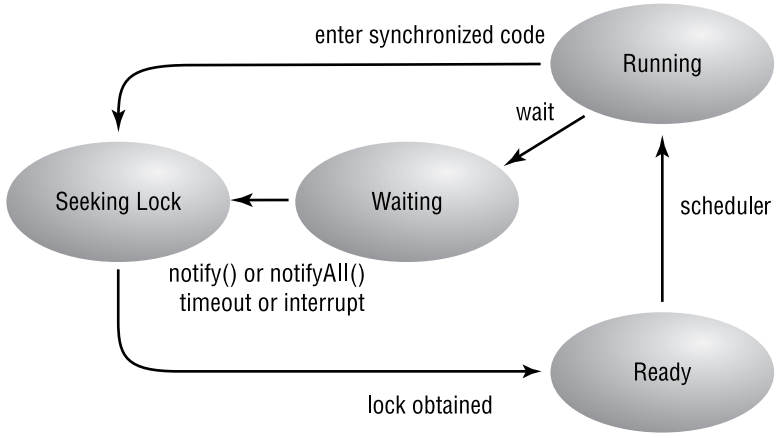
wait() and *notify()*

The `wait()` and `notify()` methods provide a way for a shared object to pause a thread when it becomes unavailable to that thread and to allow the thread to continue when appropriate. The threads themselves never have to check the state of the shared object.

An object that controls its client threads in this manner is known as a monitor. In strict Java terminology, a monitor is any object that has some synchronized code. To be really useful, most monitors make use of `wait()` and `notify()` methods. So, the `Mailbox` class is already a monitor; it just is not quite useful yet.

Figure 7.7 shows the state transitions of `wait()` and `notify()`.

FIGURE 7.7 The monitor states



Both `wait()` and `notify()` must be called in synchronized code. A thread that calls `wait()` releases the virtual CPU; at the same time, it releases the lock. It enters a pool of waiting threads, which is managed by the object whose `wait()` method got called. Every object has such a pool. The following code shows how the `Mailbox` class's `retrieveMessage()` method could be modified to begin taking advantage of calling `wait()`:

```
1. public synchronized String retrieveMessage() {
2.   while (request == false) {
3.     try {
4.       wait();
5.     } catch (InterruptedException e) { }
6.   }
7.   request = false;
8.   return message;
9. }
```

Now consider what happens when a message-consuming thread calls this method. The call might look like this:

```
myMailbox.retrieveMessage();
```

When a message-consuming thread calls this method, the thread must first acquire the lock for `myMailbox`. Acquiring the lock could happen immediately, or it could incur a delay if some other thread is executing any of the synchronized code of `myMailbox`. One way or another, eventually the consumer thread has the lock and begins to execute at line 2. The code first checks the `request` flag. If the flag is not set, then `myMailbox` has no message for the thread to retrieve. In this case the `wait()` method is called at line 4 (it can throw an `InterruptedException`, so the `try/catch` code is required, and the `while` will retest the condition). When line 4 executes, the consumer thread ceases execution; it also releases the lock for `myMailbox` and enters the pool of waiting threads managed by `myMailbox`.

The consumer thread has been successfully prevented from corrupting the `myMailbox` monitor. Unfortunately, it is stuck in the monitor's pool of waiting threads. When the monitor changes to a state where it can provide the consumer with something to do, then something will have to be done to get the consumer out of the Waiting state. This is done by calling `notify()` when the monitor's `request` flag becomes true, which happens only in the `storeMessage()` method. The revised `storeMessage()` looks like this:

```
1. public synchronized void
2. storeMessage(String message) {
3.   this.message = message;
4.   request = true;
5.   notify();
6. }
```

On line 5, the code calls `notify()` just after changing the monitor's state. The `notify()` method arbitrarily selects one of the threads in the monitor's waiting pool and moves it to the Seeking Lock state. Eventually that thread will acquire the mailbox's lock and can proceed with execution.

Now imagine a complete scenario. A consumer thread calls `retrieveMessage()` on a mailbox that has no message. It acquires the lock and begins executing the method. It sees that the `request` flag is false, so it calls `wait()` and joins the mailbox's waiting pool. (In this simple example, no other threads are in the pool.) Because the consumer has called `wait()`, it has given up the lock. Later, a message-producing thread calls `storeMessage()` on the same mailbox. It acquires the lock, stores its message in the monitor's instance variable, and sets the `request` flag to true. The producer then calls `notify()`. At this moment, only one thread is in the monitor's waiting pool: the consumer. So the consumer gets moved out of the waiting pool and into the Seeking Lock state. Now the producer returns from `storeMessage()`; because the producer has exited from synchronized code, it gives up the monitor's lock. Later the patient consumer reacquires the lock and gets to execute; once this happens, it checks the `request` flag and (finally!) sees that a message is available for consumption. The consumer returns the message; upon return it automatically releases the lock.

To briefly summarize this scenario: a consumer tried to consume something, but there was nothing to consume, so the consumer waited. Later a producer produced something. At that point there was something for the consumer to consume, so the consumer was notified; once the producer was done with the monitor, the consumer consumed a message.

As Figure 7.7 shows, a waiting thread has ways to get out of the Waiting state that do not require being notified. One version of the `wait()` call takes an argument that specifies a timeout in milliseconds; if the timeout expires, the thread moves to the Seeking Lock state, even if it has not been notified. No matter what version of `wait()` is invoked, if the waiting thread receives an `interrupt()` call, it moves immediately to the Seeking Lock state.

This example protected the consumer against the possibility that the monitor might be empty; the protection was implemented with a `wait()` call in `retrieveMessage()` and a `notify()` call in `storeMessage()`. A similar precaution must be taken in case a producer thread wants to produce into a monitor that already contains a message. To be robust, `storeMessage()` needs to call `wait()`, and `retrieveMessage()` needs to call `notify()`. The complete `Mailbox` class looks like this:

```
 1. class Mailbox {
 2.   private boolean   request;
 3.   private String    message;
 4.
 5.   public synchronized void
 6.   storeMessage(String message) {
 7.     while(request == true) {
 8.        // No room for another message
 9.       try {
10.         wait();
11.       } catch (InterruptedException e) { }
12.     }
13.     request = true;
14.     this.message = message;
15.     notify();
16.   }
17.
18.   public synchronized String retrieveMessage() {
19.     while(request == false) {
20.       // No message to retrieve
21.       try {
22.         wait();
23.       } catch (InterruptedException e) { }
24.     }
25.     request = false;
26.     notify();
27.     return message;
28.   }
29. }
```

By synchronizing code and judiciously calling `wait()` and `notify()`, monitors such as the `Mailbox` class can ensure the proper interaction of client threads and protect shared data from corruption.

Here are the main points to remember about `wait()`:

- The calling thread gives up the CPU.
- The calling thread gives up the lock.
- The calling thread goes into the monitor's waiting pool.

Here are the main points to remember about `notify()`:

- One arbitrarily chosen thread gets moved out of the monitor's waiting pool and into the Seeking Lock state.
- The thread that was notified must re-acquire the monitor's lock before it can proceed.

Real World Scenario

Order of Notification

If an object has multiple threads waiting for it, there is no guarantee about the order in which those threads will be revived when the object receives a `notifyAll()` call. In this exercise, you'll observe the order of notification.

Create a class called `NotifyLab`, with two inner classes. (You will probably want the inner classes to be static, because instances of them will be constructed in the `main()` method.) The first inner class, called `Rendezvous`, should provide a method called `hurryUpAndWait()`, which performs any necessary housekeeping and then calls `wait()`. The `main()` method will construct a single instance of this inner class. The second inner class, called `Waiter`, extends `Thread`; its `run()` method calls `hurryUpAndWait()` on the instance of `Rendezvous`, and then prints out a message to report that notification has happened. Each instance of `Waiter` should have a unique serial number, assigned at creation time and printed out after notification, so that you will be able to know the order in which threads were notified.

This exercise is mostly straightforward, but one part is tricky: you have to make sure that the threads call `wait()` in serial order. If you just create and start several waiter threads, you won't know the order in which they call `wait()`, so the output from your program won't be meaningful. One way to take care of this issue is to add a little housekeeping functionality to the `hurryUpAndWait()` method of `Rendezvous`. The rest is for you to work out!

The solution appears on the CD-ROM in the file `NotifyLab.java`. Here is a sample of its output:

```
>java NotifyLab 5
Thread #0 just got notified.
```

```
Thread #2 just got notified.
Thread #3 just got notified.
Thread #4 just got notified.
Thread #1 just got notified.
```

Be aware that the results you observe are anecdotal and should not be interpreted as predictors of general JVM behavior. Other JVMs may behave differently. The only way to write reliable code using `notifyAll()` is to be indifferent to order of notification.

The Class Lock

It is clear by now that every object (that is, every instance of every class) has a lock. Every class also has a lock. The class lock controls access to all synchronized static code in the class. Consider the following example:

```
class X {
  static int x, y;
  static synchronized void foo() {
    x++;
    y++;
  }
}
```

When the `foo()` method is called (for example, with the code `X.foo()`), the invoking thread must acquire the class lock for the `X` class. Ordinarily, when a thread attempts to call a nonstatic synchronized method, the thread must acquire the lock of the current object; the current object is referenced by `this` in the scope of the method. However, there is no `this` reference in a static method because there is no current object.

If Java did not provide class locks, there would be no built-in way to synchronize static code and no way to protect shared static data such as `x` and `y` in the previous example.

Beyond the Pure Model

The mailbox example of the previous few sections is a very simple example of a situation involving one producer and one consumer. In real life, things are not always so simple. You might have a monitor that has several methods that do not purely produce or purely consume. All you can say in general about such methods is that they cannot proceed unless the monitor is in a certain state, and they themselves can change the monitor's state in ways that could be of vital interest to the other methods.

The `notify()` method is not precise: You cannot specify which thread is to be notified. In a mixed-up scenario such as the one just described, a thread might alter the monitor's state in

a way that is useless to the particular thread that gets notified. In such a case, the monitor's methods should take two precautions:

- Always check the monitor's state in a `while` loop rather than an `if` statement.
- After changing the monitor's state, call `notifyAll()` rather than `notify()`.

The first precaution means that you should *not* do the following:

```
1. public synchronized void mixedUpMethod() {
2.   if (i<16 || f>4.3f || message.equals("UH-OH") {
3.     try { wait(); } catch (InterruptedException e) { }
4.   }
5.
6.   // Proceed in a way that changes state, and then...
7.   notify();
8. }
```

The danger is that sometimes a thread might execute the test on line 2 and then notice that `i` is (for example) 15, and have to wait. Later, another thread might change the monitor's state by setting `i` to –23444 and then call `notify()`. If the original thread is the one that gets notified, it will pick up where it left off, even though the monitor is not in a state where it is ready for `mixedUpMethod()`.

The solution is to change `mixedUpMethod()` as follows:

```
1. public synchronized void mixedUpMethod() {
2.   while (i<16 || f>4.3f || message.equals("UH-OH") {
3.     try { wait(); } catch (InterruptedException e) { }
4.   }
5.
6.   // Proceed in a way that changes state, and then...
7.   notifyAll();
8. }
```

The monitor's other synchronized methods should be modified in a similar manner. Now when a waiting thread gets notified, it does not assume that the monitor's state is acceptable. It checks again, in the `while`-loop check on line 2. If the state is still not conducive, the thread waits again.

On line 7, having made its own modifications to the monitor's state, the code calls `notifyAll()`; this call is like `notify()`, but it moves *every* thread in the monitor's waiting pool to the Seeking Lock state. Presumably every thread's `wait()` call happened in a loop like the one on lines 2–4, so every thread will once again check the monitor's state and either wait or proceed. Note that if a monitor has a large number of waiting threads, calling `notifyAll()` can cost a lot of time.

Using a `while` loop to check the monitor's state is a good idea even if you are coding a pure model of one producer and one consumer. After all, you can never be sure that somebody won't try to add an extra producer or an extra consumer.

Deadlock

The term *deadlock* describes another class of situations that might prevent a thread from executing. In general terms, if a thread blocks because it is waiting for a condition, and something else in the program makes it impossible for that condition to arise, then the thread is said to be deadlocked.

Deadlock conditions can arise for many reasons, but there is one classic example of the situation that is easy to understand. Because it is used as the standard example, this situation has a special name of its own: "deadly embrace."

Imagine a thread is trying to obtain exclusive use of two locks that are encapsulated in objects `a` and `b`. First the thread gets the lock on object `a`, and then it proceeds to try to get the lock on object `b`. This process sounds innocent enough, but now imagine that another thread already holds the lock on object `b`. Clearly, the first thread cannot proceed until the second thread releases the lock on object `b`.

Now for the nasty part: imagine that the other thread, while holding the lock on object `b`, is trying to get the lock on object `a`. This situation is now hopeless. The first thread holds the lock on object `a` and cannot proceed without the lock on object `b`. Further, the first thread cannot release the lock on object `a` until it has obtained the lock on object `b`. At the same time, the second thread holds the lock on object `b` and cannot release it until it obtains the lock on object `a`.

Let's have a look at code that could cause this situation:

```
 1. public class Deadlock implements Runnable {
 2.   public static void main(String [] args)  {
 3.     Object a = "Resource A";
 4.     Object b = "Resource B";
 5.     Thread t1 = new Thread(new Deadlock(a, b));
 6.     Thread t2 = new Thread(new Deadlock(b, a));
 7.     t1.start();
 8.     t2.start();
 9.   }
10.
11.   private Object firstResource;
12.   private Object secondResource;
13.
14.   public Deadlock(Object first, Object second) {
```

```
15.     firstResource = first;
16.     secondResource = second;
17.   }
18.
19.   public void run() {
20.     for (;;) {
21.       System.out.println(
22.         Thread.currentThread().getName() +
23.         " Looking for lock on " + firstResource);
24.
25.       synchronized (firstResource) {
26.         System.out.println(
27.           Thread.currentThread().getName() +
28.           " Obtained lock on " + firstResource);
29.
30.         System.out.println(
31.           Thread.currentThread().getName() +
32.           " Looking for lock on " + secondResource);
33.
34.         synchronized (secondResource) {
35.           System.out.println(
36.             Thread.currentThread().getName() +
37.             " Obtained lock on " + secondResource);
38.           // simulate some time consuming activity
39.           try { Thread.sleep(100); }
40.           catch (InterruptedException ex) {}
41.         }
42.       }
43.     }
44.   }
45. }
```

In this code, the resources are locked at lines 25 and 34. Notice that, although the same code executes in both threads, the references `firstResource` and `secondResource` actually refer to different objects in both threads. This is the case because of the way the two `Deadlock` instances are constructed on lines 5 and 6.

When you run the code, the exact behavior is nondeterministic, because of differences in thread scheduling between executions. Commonly, however, the output will look something like this:

```
Thread-1 Looking for lock on Resource A
Thread-1 Obtained lock on Resource A
```

```
Thread-2 Looking for lock on Resource B
Thread-1 Looking for lock on Resource B
Thread-2 Obtained lock on Resource B
Thread-2 Looking for lock on Resource A
```

If you study this output, you will see that the first thread (Thread-1) holds the lock on Resource A and is trying to obtain the lock on Resource B. Meanwhile, the second thread (Thread-2) holds the lock on Resource B—which prevents the first thread from ever executing. Further, the second thread is waiting for Resource A and can never proceed because that object will never be released by the first thread.

It is useful to realize that if both threads were looking for the locks in the same order, then the deadly embrace situation would never occur. However, it can be very difficult to arrange for this ordering solution in situations where the threads are disparate parts of the program. Indeed, looking at the variables used in this example, you will see that it can sometimes be difficult to recognize an ordering problem like this even if the code is all in one place.

Another Way to Synchronize

There is an additional way to synchronize code. It is hardly common and generally should not be used without a compelling reason. This approach is to synchronize on the lock of a different object.

We briefly mentioned in an earlier section ("The Object Lock and Synchronization") that you can synchronize on the lock of any object. Suppose, for example, that you have the following class, which is admittedly a bit contrived:

```
1. class StrangeSync {
2.   Rectangle rect = new Rectangle(11, 13, 1100, 1300);
3.   void doit() {
4.     int x = 504;
5.     int y = x / 3;
6.     rect.width -= x;
7.     rect.height -= y;
8.   }
9. }
```

If you add the `synchronized` keyword at line 3, then a thread that wants to execute the `doit()` method of some instance of `StrangeSync` must first acquire the lock for that instance. That may be exactly what you want. However, perhaps you only want to synchronize lines 6 and 7, and perhaps you want a thread attempting to execute those lines to synchronize on the lock of `rect`, rather than on the lock of the current executing object. The way to do this is shown here:

```
1. class StrangeSync {
2.   Rectangle rect = new Rectangle(11, 13, 1100, 1300);
3.   void doit() {
```

```
 4.     int x = 504;
 5.     int y = x / 3;
 6.     synchronized(rect) {
 7.       rect.width -= x;
 8.       rect.height -= y;
 9.     }
10.   }
11. }
```

This code synchronizes on the lock of some arbitrary object (specified in parentheses after the `synchronized` keyword on line 6), rather than synchronizing on the lock of the current object. Also, the code synchronizes just two lines, rather than an entire method.

It is difficult to find a good reason for synchronizing on an arbitrary object. However, synchronizing only a subset of a method can be useful; sometimes you want to hold the lock as briefly as possible, so that other threads can get their turn as soon as possible. The Java compiler insists that when you synchronize a portion of a method (rather than the entire method), you have to specify an object in parentheses after the `synchronized` keyword. If you put `this` in the parentheses, then the goal is achieved: you have synchronized a portion of a method, with the lock using the lock of the object that owns the method.

To summarize, your options are

- To synchronize an entire method, using the lock of the object that owns the method. To do this, put the `synchronized` keyword in the method's declaration.
- To synchronize part of a method, using the lock of an arbitrary object. Put curly brackets around the code to be synchronized, preceded by `synchronized(theArbitraryObject)`.
- To synchronize part of a method, using the lock of the object that owns the method. Put curly brackets around the code to be synchronized, preceded by `synchronized(this)`.

Summary

A Java thread scheduler can be preemptive or time-sliced, depending on the design of the JVM. No matter which design is used, a thread becomes eligible for execution (ready) when its `start()` method is invoked. When a thread begins execution, the scheduler calls the `run()` method of the thread's target (if there is a target) or the `run()` method of the thread itself (if there is no target). The target must be an instance of a class that implements the `Runnable` interface.

In the course of execution, a thread can become ineligible for execution for a number or reasons: A thread can suspend, sleep, block, or wait. In due time (we hope!), conditions will change so that the thread once more becomes eligible for execution; then the thread enters the Ready state and eventually can execute.

When a thread returns from its `run()` method, it enters the Dead state and cannot be restarted. You might find the following lists to be a useful summary of Java's threads.

- Scheduler implementations:
  - Preemptive
  - Time-sliced
- Constructing a thread:
  - `new Thread()`: no target; thread's own `run()` method is executed
  - `new Thread(Runnable target)`: target's `run()` method is executed
- Nonrunnable thread states:
  - Suspended: caused by `suspend()`, waits for `resume()`
  - Sleeping: caused by `sleep()`, waits for timeout
  - Blocked: caused by various I/O calls or by failing to get a monitor's lock, waits for I/O or for the monitor's lock
  - Waiting: caused by `wait()`, waits for `notify()` or `notifyAll()`
  - Dead: caused by `stop()` or returning from `run()`, no way out

Exam Essentials

Know how to write and run code for a thread by extending `java.lang.Thread`. Extend the `Thread` class, overriding the `run()` method. Create an instance of the subclass and call its `start()` method to launch the new thread.

Know how to write and run code for a thread by implementing the interface `java.lang.Runnable`. Create a class that implements `Runnable`. Construct the thread with the `Thread(Runnable)` constructor and call its `start()` method.

Know the mechanisms that suspend a thread's execution. These mechanisms include: entering any synchronized code, or calling `wait()`, `yield()`, or `sleep()`.

Recognize code that might cause deadly embrace. Deadlock conditions cause permanent suspension of threads, and deadly embrace is the classic example of this.

Understand the functionality of the `wait()`, `notify()`, and `notifyAll()` methods. These methods are explained in detail in the "`wait()` and `notify()`" section.

Know that the resumption order of threads that execute `wait()` on an object is not specified. The Java specification states that the resumption order for threads waiting on an object is unspecified.

Key Terms

Before you take the exam, be certain you are familiar with the following terms:

blocking
deadlock
lock
monitor
multithreaded
preemptive scheduling
priority
round-robin scheduling
single-threaded
sleeping thread
suspending
synchronized code
thread scheduler
time-sliced scheduling
waiting state
yielding

Review Questions

1. Which one statement is true concerning the following code?

```
 1. class Greebo extends java.util.Vector
 2.   implements Runnable {
 3.     public void run(String message) {
 4.       System.out.println("in run() method: " +
 5.       message);
 6.   }
 7. }
 8.
 9. class GreeboTest {
10.   public static void main(String args[]) {
12.     Greebo g = new Greebo();
13.     Thread t = new Thread(g);
14.     t.start();
15.   }
16. }
```

   A. There will be a compiler error, because class `Greebo` does not correctly implement the `Runnable` interface.

   B. There will be a compiler error at line 13, because you cannot pass a parameter to the constructor of a `Thread`.

   C. The code will compile correctly but will crash with an exception at line 13.

   D. The code will compile correctly but will crash with an exception at line 14.

   E. The code will compile correctly and will execute without throwing any exceptions.

2. Which one statement is always true about the following application?

```
 1. class HiPri extends Thread {
 2.   HiPri() {
 3.     setPriority(10);
 4.   }
 5.
 6.   public void run() {
 7.     System.out.println(
 8.       "Another thread starting up.");
 9.     while (true) { }
10.   }
11.
12.   public static void main(String args[]) {
```

```
13.     HiPri hp1 = new HiPri();
14.     HiPri hp2 = new HiPri();
15.     HiPri hp3 = new HiPri();
16.     hp1.start();
17.     hp2.start();
18.     hp3.start();
19.   }
20. }
```

A. When the application is run, thread `hp1` will execute; threads `hp2` and `hp3` will never get the CPU.

B. When the application is run, all three threads (`hp1`, `hp2`, and `hp3`) will get to execute, taking time-sliced turns in the CPU.

C. Either A or B will be true, depending on the underlying platform.

3. A thread wants to make a second thread ineligible for execution. To do this, the first thread can call the `yield()` method on the second thread.

A. True

B. False

4. A thread's `run()` method includes the following lines:

```
1. try {
2.   sleep(100);
3. } catch (InterruptedException e) { }
```

Assuming the thread is not interrupted, which one of the following statements is correct?

A. The code will not compile, because exceptions cannot be caught in a thread's `run()` method.

B. At line 2, the thread will stop running. Execution will resume in, at most, 100 milliseconds.

C. At line 2, the thread will stop running. It will resume running in exactly 100 milliseconds.

D. At line 2, the thread will stop running. It will resume running some time after 100 milliseconds have elapsed.

5. A monitor called `mon` has 10 threads in its waiting pool; all these waiting threads have the same priority. One of the threads is `thr1`. How can you notify `thr1` so that it alone moves from the Waiting state to the Ready state?

A. Execute `notify(thr1);` from within synchronized code of `mon`.

B. Execute `mon.notify(thr1);` from synchronized code of any object.

C. Execute `thr1.notify();` from synchronized code of any object.

D. Execute `thr1.notify();` from any code (synchronized or not) of any object.

E. You cannot specify which thread will get notified.

6. If you attempt to compile and execute the following application, will it ever print out the message `In xxx`?

```
1. class TestThread3 extends Thread {
2.   public void run() {
3.     System.out.println("Running");
4.     System.out.println("Done");
5.   }
6.
7.  private void xxx() {
8.    System.out.println("In xxx");
9.  }
10.
11.   public static void main(String args[]) {
12.     TestThread3 ttt = new TestThread3();
13.     ttt.xxx();
14.     ttt.start();
12.   }
13. }
```

   A. Yes

   B. No

7. A Java monitor must either extend `Thread` or implement `Runnable`.

   A. True

   B. False

8. Which of the following methods in the Thread class have been deprecated?

   A. `suspend()` and `resume()`

   B. `wait()` and `notify()`

   C. `start()` and `stop()`

   D. `sleep()` and `yield()`

9. Which of the following statements about threads is true?

   A. Every thread starts executing with a priority of 5.

   B. Threads inherit their priority from their parent thread.

   C. Threads are guaranteed to run with the priority that you set using the `setPriority()` method.

   D. Thread priority is an integer ranging from 1 to 100.

10. Which of the following statements about the `wait()` and `notify()` methods is true?

A. The `wait()` and `notify()` methods can be called outside of synchronized code.

B. The programmer can specify which thread should be notified in a `notify()` method call.

C. The thread that calls `wait()` goes into the monitor's pool of waiting threads.

D. The thread that calls `notify()` gives up the lock.

Answers to Review Questions

1. A. The `Runnable` interface defines a `run()` method with `void` return type and no parameters. The method given in the problem has a `String` parameter, so the compiler will complain that class `Greebo` does not define `void run()` from interface `Runnable`. B is wrong, because you can definitely pass a parameter to a thread's constructor; the parameter becomes the thread's target. C, D, and E are nonsense.

2. C. A is true on a preemptive platform, and B is true on a time-sliced platform. The moral is that such code should be avoided, because it gives such different results on different platforms.

3. B. The `yield()` method is static and always causes the current thread to yield. In this case, ironically, the first thread will yield.

4. D. The thread will sleep for 100 milliseconds (more or less, given the resolution of the JVM being used). Then the thread will enter the Ready state; it will not actually run until the scheduler permits it to run.

5. E. When you call `notify()` on a monitor, you have no control over which waiting thread gets notified.

6. A. The call to `xxx()` occurs before the thread is registered with the thread scheduler, so the question has nothing to do with threads.

7. B. A monitor is an instance of any class that has synchronized code.

8. A. The `suspend()` and `resume()` methods were deprecated in the Java 2 release.

9. B. A is not correct because, although the default priority for a thread is 5, it may be changed by the parent thread. C is not correct because Java does not make any promises about priority at runtime. Finally, D is incorrect because thread priorities range from 1 to 10.

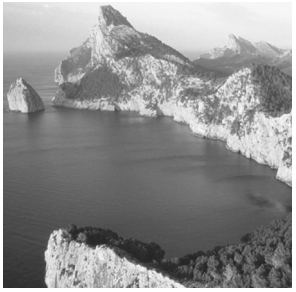
10. C. Option A is incorrect because `wait()` and `notify()` must be called from within synchronized code. Option B is incorrect because the `notify()` call arbitrarily selects a thread to notify from the pool of waiting threads. Option D is incorrect because the thread that calls `wait()` is the thread that gives up the lock.

Chapter 8

The *java.lang* and *java.util* Packages

JAVA CERTIFICATION EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- ✓ **8.1 Write code using the following methods of the `java.lang.Math` class: `abs`, `ceil`, `floor`, `max`, `min`, `random`, `round`, `sin`, `cos`, `tan`, `sqrt`.**
- ✓ **8.2 Describe the significance of the immutability of `String` objects.**
- ✓ **8.3 Describe the significance of wrapper classes, including making appropriate selections in the wrapper classes to suit specified behavior requirements, stating the result of executing a fragment of code that includes an instance of one of the wrapper classes, and writing code using the following methods of the wrapper classes (e.g., `Integer`, `Double`, etc.):**
  - `doubleValue()`
  - `floatValue()`
  - `intValue()`
  - `longValue()`
  - `parseXxx()`
  - `getXxx()`
  - `toString()`
  - `toHexString()`
- ✓ **9.1 Make appropriate selection of collection classes/ interfaces to suit specified behavior requirements.**
- ✓ **9.2 Distinguish between correct and incorrect implementations of the hashcode methods.**



The `java.lang` package contains classes that are central to the operation of the Java language and environment. Very little can be done without the `String` class, for example, and the `Object` class is completely indispensable. The Java compiler automatically imports all the classes in the `java.lang` package into every source file.

This chapter examines some of the most important classes of the `java.lang` package:

- `Object`
- `Math`
- The wrapper classes
- `String`
- `StringBuffer`

In addition, this chapter also covers the collection classes of the `java.util` package.

The *Object* Class

The `Object` class is the ultimate ancestor of all Java classes. If a class does not contain the `extends` keyword in its declaration, the compiler builds a class that extends directly from `Object`.

All the methods of `Object` are inherited by every class. Three of these methods (`wait()`, `notify()`, and `notifyAll()`) support thread control, and they are discussed in detail in Chapter 7, "Threads." Two other methods, `equals()` and `toString()`, provide little functionality on their own. The intention is that programmers who develop reusable classes can override `equals()` and `toString()` in order to provide useful class-specific functionality.

The signature of `equals()` is

```
public boolean equals(Object object)
```

The method is supposed to provide "deep" comparison, in contrast to the "shallow" comparison provided by the == operator. To see the difference between the two types of comparison, consider the `java.util.Date` class, which represents a moment in time. Suppose you have two references of type `Date`: `d1` and `d2`. One way to compare them is with the following line of code:

```
if (d1 == d2)
```

The comparison will be true if the *reference* in `d1` is equal to the *reference* in `d2`. Of course, this is the case only when both variables refer to the same object.

Sometimes you want a different kind of comparison. Sometimes you don't care whether `d1` and `d2` refer to the same `Date` object. Sometimes you *know* they are different objects. What you care about is whether the two objects, which encapsulate day and time information, represent the same moment in time. In this case, you don't want the shallow reference-level comparison of `==`; you need to look deeply into the objects themselves. The way to do that is with the `equals()` method:

```
if (d1.equals(d2))
```

The version of `equals()` provided by the `Object` class is not very useful because it just does an `==` comparison. All classes should override `equals()` so that it performs a useful comparison. That is just what most of the standard Java classes do: they compare the relevant instance variables of two objects.

The purpose of the `toString()` method is to provide a string representation of an object's state. This method is especially useful for debugging.

The `toString()` method is similar to `equals()` in that the version provided by the `Object` class is not especially useful—it just prints out the object's class name, followed by a hash code. Many JDK classes override `toString()` to provide more useful information. Java's string-concatenation facility makes use of this method, as you will see later in this chapter, in the "String Concatenation the Easy Way" section.

The *Math* Class

Java's `Math` class contains a collection of methods and two constants that support mathematical computation. The class is `final`, so you cannot extend it. The constructor is private, so you cannot create an instance. Fortunately, the methods and constants are static, so they can be accessed through the class name without having to construct a `Math` object. (See Chapter 3, "Modifiers," for an explanation of Java's modifiers, including `final`, `static`, and `private`.)

The two constants of the `Math` class are `Math.PI` and `Math.E`. They are declared to be public, static, final, and `double`.

The methods of the `Math` class cover a broad range of mathematical functionality, including trigonometry, logarithms and exponentiation, and rounding. The intensive number-crunching methods are often written as native methods to take advantage of any math-acceleration hardware that might be present on the underlying machine.

The Certification Exam requires you to know about the methods shown in Table 8.1.

TABLE 8.1 Methods of the *Math* Class

Method	Returns
`int abs(int i)`	Absolute value of `i`
`long abs(long l)`	Absolute value of `l`
`float abs(float f)`	Absolute value of `f`
`double abs(double d)`	Absolute value of `d`
`double ceil(double d)`	The smallest integer that is not less than `d` (returns as a `double`)
`double floor(double d)`	The largest integer that is not greater than `d` (returns as a `double`)
`int max(int i1, int i2)`	Greater of `i1` and `i2`
`long max(long l1, long l2)`	Greater of `l1` and `l2`
`float max(float f1, float f2)`	Greater of `f1` and `f2`
`double max(double d1, double d2)`	Greater of `d1` and `d2`
`int min(int i1, int i2)`	Smaller of `i1` and `i2`
`long min(long l1, long l2)`	Smaller of `l1` and `l2`
`float min(float f1, float f2)`	Smaller of `f1` and `f2`
`double min(double d1, double d2)`	Smaller of `d1` and `d2`
`double random()`	Random number >= 0.0and < 1.0
`int round(float f)`	Closest `int` to `f`
`long round(double d)`	Closest `long` to `d`
`double `(double d)`	Sine of `d`
`double cos(double d)`	Cosine of `d`
`double tan(double d)`	Tangent of `d`
`double sqrt(double d)`	Square root of `d`

The Wrapper Classes

Each Java primitive data type has a corresponding *wrapper class*. A wrapper class is simply a class that encapsulates a single, immutable value. For example, the `Integer` class wraps up an `int` value, and the `Float` class wraps up a `float` value. The wrapper class names do not perfectly match the corresponding primitive data type names. Table 8.2 lists the primitives and wrappers.

TABLE 8.2 Primitives and Wrappers

Primitive Data Type	Wrapper Class
`boolean`	`Boolean`
`byte`	`Byte`
`char`	`Character`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`

All the wrapper classes can be constructed by passing the value to be wrapped into the appropriate constructor. The following code fragment shows how to construct an instance of each wrapper type:

```
 1. boolean   primitiveBoolean = true;
 2. Boolean   wrappedBoolean =
 3.             new Boolean(primitiveBoolean);
 4.
 5. byte      primitiveByte = 41;
 6. Byte      wrappedByte = new Byte(primitiveByte);
 7.
 8. char      primitiveChar = 'M';
 9. Character wrappedChar = new Character(primitiveChar);
10.
11. short     primitiveShort = 31313;
```

```
12. Short     wrappedShort = new Short(primitiveShort);
13.
14. int       primitiveInt = 12345678;
15. Integer   wrappedInt = new Integer(primitiveInt);
16.
17. long      primitiveLong = 12345678987654321L;
18. Long      wrappedLong = new Long(primitiveLong);
19.
20. float     primitiveFloat = 1.11f;
21. Float     wrappedFloat = new Float(primitiveFloat);
22.
23. double    primitiveDouble = 1.11111111;
24. Double    wrappedDouble =
25.             new Double(primitiveDouble);
```

There is another way to construct any of these classes, with the exception of `Character`: you can pass into the constructor a `String` that represents the value to be wrapped. Most of these constructors throw `NumberFormatException`, because there is always the possibility that the string will not represent a valid value. Only `Boolean` does not throw this exception; the constructor accepts any `String` input and wraps a true value if the string (ignoring case) is "true." The following code fragment shows how to construct wrappers from strings:

```
 1. Boolean wrappedBoolean = new Boolean("True");
 2. try {
 3.   Byte wrappedByte = new Byte("41");
 4.   Short wrappedShort = new Short("31313");
 5.   Integer wrappedInt = new Integer("12345678");
 6.   Long wrappedLong = new Long("12345678987654321");
 7.   Float wrappedFloat = new Float("1.11f");
 8.   Double wrappedDouble = new Double("1.11111111");
 9. }
10. catch (NumberFormatException e) {
11.   System.out.println("Bad Number Format");
12. }
```

The values wrapped inside two wrappers of the same type can be checked for equality by using the `equals()` method discussed in the previous section. For example, the following code fragment checks two instances of `Double`:

```
1. Double d1 = new Double(1.01055);
2. Double d2 = new Double("1.11348");
3. if (d1.equals(d2)) {
4.   // Do something.
5. }
```

After a value has been wrapped, you may eventually need to extract it. For an instance of `Boolean`, you can call `booleanValue()`. For an instance of `Character`, you can call `charValue()`. The other six classes extend from the abstract superclass `Number`, which provides methods to retrieve the wrapped value as a `byte`, a `short`, an `int`, a `long`, a `float`, or a `double`. In other words, the value of any wrapped number can be retrieved as any numeric type. The retrieval methods are

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

The wrapper classes are useful whenever it would be convenient to treat a piece of primitive data as if it were an object. A good example is the `Vector` class, which is a dynamically growing collection of objects of arbitrary type. The method for adding an object to a vector is

```
public boolean add(Object ob)
```

Using this method, you can add any object of any type to a vector; you can even add an array (you saw why in Chapter 4, "Converting and Casting"). You cannot, however, add an `int`, a `long`, or any other primitive to a vector. No special methods exist for doing so, and `add(Object ob)` will not work because there is no automatic conversion from a primitive to an object. Thus, the following code will not compile:

```
1. Vector vec = new Vector();
2. boolean boo = false;
3. vec.add(boo);  // Illegal
```

The solution is to wrap the `boolean` primitive, as shown here:

```
1. Vector vec = new Vector();
2. boolean boo = false;
3. Boolean wrapper = new Boolean(boo);
4. vec.add(wrapper);  // Legal
```

The wrapper classes are useful in another way: They provide a variety of utility methods, most of which are static. For example, the static method `Character.isDigit(char ch)` returns a `boolean` that tells whether the character represents a base-10 digit. All the wrapper classes except `Character` have a static method called `valueOf(String s)`, which parses a string and constructs and returns a wrapper instance of the same type as the class whose method was called. So, for example, `Long.valueOf("23")` constructs and returns an instance of the `Long` class that wraps the value 23.

One set of static wrapper methods are the `parseXXX()` methods. These are `Byte.parseByte()`, `Short.parseShort()`, `Integer.parseInt()`, `Long.parseLong()`, `Float.parseFloat()`, and

`Double.parseDouble()`. Each of these takes a `String` argument and returns the corresponding primitive type. They all throw `NumberFormatException`.

Other static methods that are mentioned in the exam objectives are the `getXXX()`methods. These are `Boolean.getBoolean()`,`Integer.getInteger()`, and `Long.getLong()`. Each of these takes a `String` argument that is the name of a system property and returns the value of the property. The return value is a primitive `boolean` or a wrapper `Integer` or `Long` that encapsulates the property value, provided the property is defined, is not empty, and is compatible with the respective type. `Integer.getInteger()` and `Long.getLong()` have overloaded forms that take a second argument, which is a primitive of the respective type. The second argument is a default value that is wrapped and returned in case the property is not defined, is empty, or is not compatible with the respective type.

All the wrapper classes provide `toString()` methods. Additionally, the `Integer` and `Long` classes provide `toBinaryString()`, `toOctalString()`, and `toHexString()`, which return strings in base 2, 8, and 16.

All wrapper classes have an inconvenient feature: The values they wrap are immutable. After an instance is constructed, the encapsulated value cannot be changed. It is tempting to try to subclass the wrappers, so that the subclasses inherit all the useful functionality of the original classes while offering mutable contents. Unfortunately, this strategy doesn't work because the wrapper classes are final.

To summarize the major facts about the primitive wrapper classes:

- Every primitive type has a corresponding wrapper class type.
- All wrapper types can be constructed from primitives. All except `Character` can also be constructed from strings.
- Wrapped values can be tested for equality with the `equals()` method.
- Wrapped values can be extracted with various *XXX*`Value()` methods. All six numeric wrapper types support all six numeric *XXX*`Value()` methods.
- Wrapper classes provide various utility methods, including the static `valueOf()` methods, which parse an input string.
- Wrapped values cannot be modified.

Strings

Java uses the `String` and `StringBuffer` classes to encapsulate strings of characters. Java uses 16-bit Unicode characters to support a broader range of international alphabets than would be possible with traditional 8-bit characters. Both strings and string buffers contain sequences of 16-bit Unicode characters. The next several sections examine these two classes, as well as Java's string-concatenation feature.

The *String* Class

The `String` class contains an immutable string. Once an instance is created, the string it contains cannot be changed. Numerous forms of constructor allow you to build an instance out of an array of `byte`s or `char`s, a subset of an array of `byte`s or `char`s, another string, or a string buffer. Many of these constructors give you the option of specifying a character encoding, specified as a string. However, the Certification Exam does not require you to know the details of character encodings.

Probably the most common string constructor simply takes another string as its input. This constructor is useful when you want to specify a literal value for the new string:

```
String s1 = new String("immutable");
```

An even easier abbreviation could be:

```
String s1 = "immutable";
```

It is important to be aware of what happens when you use a string literal ("immutable" in both examples). Every string literal is represented internally by an instance of `String`. Java classes may have a pool of such strings. When a literal is compiled, the compiler adds an appropriate string to the pool. However, if the same literal already appeared as a literal elsewhere in the class, then it is already represented in the pool. The compiler does not create a new copy. Instead, it uses the existing one from the pool. This process saves on memory and can do no harm. Because strings are immutable, a piece of code can't harm another piece of code by modifying a shared string.

Earlier in this chapter, you saw how the `equals()` method can be used to provide a deep equality check of two objects. With strings, the `equals()` method does what you would expect: it checks the two contained collections of characters. The following code shows how this is done:

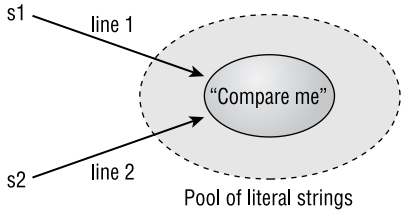
```
1. String s1 = "Compare me";
2. String s2 = "Compare me";
3. if (s1.equals(s2)) {
4.   // whatever
5. }
```

Not surprisingly, the test at line 3 succeeds. Given what you know about how string literals work, you can see that if line 3 is modified to use the == comparison, as shown here, the test still succeeds:

```
1. String s1 = "Compare me";
2. String s2 = "Compare me";
3. if (s1 == s2)) {
4.   // whatever
5. }
```

The == test is true because s2 refers to the String in the pool that was created in line 1. Figure 8.1 shows this graphically.

FIGURE 8.1 Identical literals



You can also construct a String by explicitly calling the constructor, as shown next; however, doing so causes extra memory allocation for no obvious advantage:

```
String s2 = new String("Constructed");
```

When this line is compiled, the string literal "Constructed" is placed into the pool. At runtime, the new String() statement is executed and a fresh instance of String is constructed, duplicating the String in the literal pool. Finally, a reference to the new String is assigned to s2. Figure 8.2 shows the chain of events.

FIGURE 8.2 Explicitly calling the String constructor

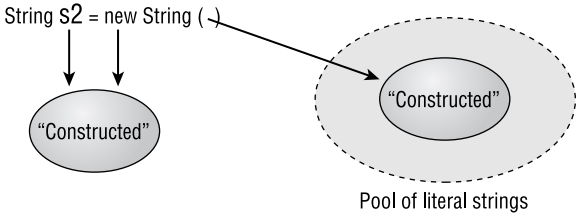


Figure 8.2 shows that explicitly calling new String() results in the existence of two objects, one in the literal pool and the other in the program's space.

You just saw that if you create a new String instance at runtime, it will not be in the pool but will be a new and distinct object. You can arrange for your new String to be placed into the pool for possible reuse, or to reuse an existing identical String from the pool, by using the intern() method of the String class. In programs that use a great many similar strings, this approach can reduce memory requirements. More importantly, in programs that make a lot of String equality comparisons, ensuring that all strings are in the pool allows you to use the == reference comparison in place of the equals() method. The equals() method runs slower because it must do a character-by-character comparison of the two strings, whereas the == operator only compares the two memory addresses.

The `String` class includes several convenient methods, some of which transform a string. For example, `toUpperCase()` converts all the characters of a string to uppercase. It is important to remember that the original string is not modified. That would be impossible, because strings are immutable. What really happens is that a new string is constructed and returned. Generally, this new string will not be in the pool unless you explicitly call `intern()` to put it there.

The methods in the following list are just some of the most useful methods of the `String` class. There are more methods than those listed here, and some of those listed have overloaded forms that take different inputs. This list includes all the methods that you are required to know for the Certification Exam, plus a few additional useful ones:

`char charAt(int index)` Returns the indexed character of a string, where the index of the initial character is 0.

`String concat(String addThis)` Returns a new string consisting of the old string followed by `addThis`.

`int compareTo(String otherString)` Performs a lexical comparison; returns an `int` that is less than 0 if the current string is less than `otherString`, equal to 0 if the strings are identical, and greater than 0 if the current string is greater than `otherString`.

`boolean endsWith(String suffix)` Returns true if the current string ends with `suffix`; otherwise returns false.

`boolean equals(Object ob)` Returns true if `ob instanceof String` and the string encapsulated by `ob` matches the string encapsulated by the executing object.

`boolean equalsIgnoreCase(String s)` Returns true if `s` matches the current string, ignoring upper- and lowercase considerations.

`int indexOf(int ch)` Returns the index within the current string of the first occurrence of `ch`. Alternative forms return the index of a string and begin searching from a specified offset.

`int lastIndexOf(int ch)` Returns the index within the current string of the last occurrence of `ch`.

`int length()` Returns the number of characters in the current string.

`String replace(char oldChar, char newChar)` Returns a new string, generated by replacing every occurrence of `oldChar` with `newChar`.

`boolean startsWith(String prefix)` Returns true if the current string begins with `prefix`; otherwise returns false. Alternate forms begin searching from a specified offset.

`String substring(int startIndex)` Returns the substring, beginning at `startIndex` of the current string and extending to the end of the current string. An alternate form specifies starting and ending offsets.

`String toLowerCase()` Converts the executing object to lowercase and returns a new string.

`String toString()` Returns the executing object.

`String toUpperCase()` Converts the executing object to uppercase and returns a new string.

`String trim()` Returns the string that results from removing whitespace characters from the beginning and ending of the current string.

The following code shows how to use two of these methods to modify a string. The original string is " 5 + 4 = 20". The code first strips off the leading blank space and then converts the addition sign to a multiplication sign:

```
1. String s = " 5 + 4 = 20";
2. s = s.trim();                  // "5 + 4 = 20"
3. s = s.replace('+', 'x');       // "5 x 4 = 20"
```

After line 3, `s` refers to a string whose appearance is shown in the line 3 comment. Of course, the modification has not taken place within the original string. Both the `trim()` call in line 2 and the `replace()` call of line 3 construct and return new strings; the address of each new string in turn gets assigned to the reference variable `s`. Figure 8.3 shows this sequence graphically.

FIGURE 8.3 Trimming and replacing

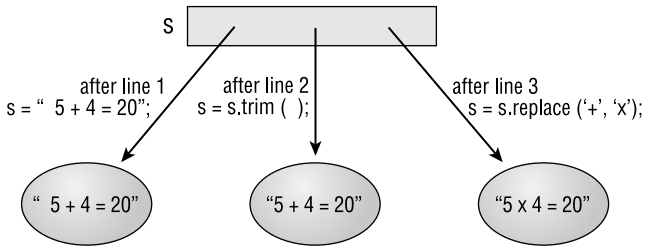


Figure 8.3 shows that the original string seems to be only modified, but it is actually replaced, because strings are immutable. If much modification is required, then this process becomes very inefficient—it stresses the garbage collector cleaning up all the old strings, and it takes time to copy the contents of the old strings into the new ones. The next section discusses a class that helps alleviate these problems because it represents a mutable string: the `StringBuffer` class.

The *StringBuffer* Class

An instance of Java's `StringBuffer` class represents a string that can be dynamically modified.

The most commonly used constructor takes a `String` instance as input. You can also construct an empty string buffer (probably with the intention of adding characters to it later). An empty string buffer can have its initial capacity specified at construction time. The three constructors are

`StringBuffer()` Constructs an empty string buffer

`StringBuffer(int capacity)` Constructs an empty string buffer with the specified initial capacity

`StringBuffer(String initialString)` Constructs a string buffer that initially contains the specified string

A string buffer has a `capacity`, which is the longest string it can represent without needing to allocate more memory. A string buffer can grow beyond this capacity as necessary, so usually

you do not have to worry about it. However, it is more efficient to declare a large initial capacity when instantiating a string buffer to avoid the system calls required to allocate more memory.

The following list presents some of the methods that modify the contents of a string buffer. All of them return the original string buffer:

`StringBuffer append(String str)` Appends `str` to the current string buffer. Alternative forms support appending primitives and character arrays which are converted to strings before appending.

`StringBuffer append(Object obj)` Calls `toString()` on `obj` and appends the result to the current string buffer.

`StringBuffer insert(int offset, String str)` Inserts `str` into the current string buffer at position `offset`. There are numerous alternative forms.

`StringBuffer reverse()` Reverses the characters of the current string buffer.

`StringBuffer setCharAt(int offset, char newChar)` Replaces the character at position `offset` with `newChar`.

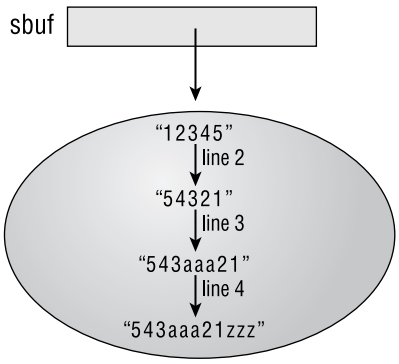
`StringBuffer setLength(int newLength)` Sets the length of the string buffer to `newLength`. If `newLength` is less than the current length, the string is truncated. If `newLength` is greater than the current length, the string is padded with null characters.

The following code shows the effect of using several of these methods in combination:

```
1. StringBuffer sbuf = new StringBuffer("12345");
2. sbuf.reverse();            // "54321"
3. sbuf.insert(3, "aaa");     // "543aaa21"
4. sbuf.append("zzz");        // "543aaa21zzz"
```

The method calls actually modify the string buffer they operate on (unlike the `String` class example of the previous section). Figure 8.4 graphically shows what this code does.

FIGURE 8.4 Modifying a string buffer



The `StringBuffer` class does not override the version of `equals()` that it inherits from `Object`. Thus the method returns `true` only when comparing references to the same single object. If two distinct instances encapsulate identical strings, `equals()` will return false.

One last string buffer method that bears mentioning is `toString()`. You saw earlier in this chapter that every class has one of these methods. Not surprisingly, the string buffer's version just returns the encapsulated string as an instance of class `String`. You will see in the next section that this method plays a crucial role in string concatenation.

String Concatenation the Easy Way

The `concat()` method of the `String` class and the `append()` method of the `StringBuffer` class glue two strings together. Another way to concatenate strings is to use Java's overloaded + operator. Similarly, another way to append a string is to use Java's overloaded += operator. However, don't forget that you, the programmer, cannot define additional operator overloads.

String concatenation is useful in many situations—for example, in debugging print statements. So, to print the value of a `double` called `radius`, all you have to do is this:

```
System.out.println("radius = " + radius);
```

This technique also works for object data types. To print the value of a `Dimension` called `dimension`, all you need is

```
System.out.println("dimension = " + dimension);
```

It is important to understand how the technique works. At compile time, if either operand of a + operator (that is, what appears on either side of a + sign) is a `String` object, then the compiler recognizes that it is in a *string context*. In a string context, the + sign is interpreted as calling for string concatenation rather than arithmetic addition.

A string context is simply a run of additions, where one of the operands is a string. For example, if variable `a` is a string, then the following partial line of code is a string context, regardless of the types of the other operands:

```
a + b + c
```

The Java compiler treats the previous code as if it were the following:

```
new
StringBuffer().append(a).append(b).append(c).toString();
```

If any of the variables (`a`, `b`, or `c`) is a primitive, the `append()` method computes an appropriate string representation. For an object variable, the `append()` method uses the string returned from calling `toString()` on the object. The conversion begins with an empty string buffer, then appends each element in turn to the string buffer, and finally calls `toString()` to convert the string buffer to a string.

The following code implements a class with its own `toString()` method:

```
 1. class Abc {
 2.   private int a;
 3.   private int b;
 4.   private int c;
 5.
 6.   Abc(int a, int b, int c) {
 7.     this.a = a;
 8.     this.b = b;
 9.     this.c = c;
10.   }
11.
12.   public String toString() {
13.     return "a = " + a + ", b = " + b + ", c = " + c;
14.   }
15. }
```

Now the `toString()` method (lines 12–14) can be used by any code that wants to take advantage of string concatenation. For example:

```
Abc theAbc = new Abc(11, 13, 48);
System.out.println("Here it is: " + theAbc);
```

The output is

```
Here it is: a = 11, b = 13, c = 48
```

To summarize the sequence of events for a string context:

1. An empty string buffer is constructed.
2. Each argument in turn is concatenated to the string buffer, using the `append()` method.
3. The string buffer is converted to a string with a call to `toString()`.

That is all you need to know about string manipulation for the Certification Exam, and it's probably all you need to know to write effective and efficient code, too. Next, we're going to look at collections.

The Collections API

Many programs need to keep track of groups of related data items. The most basic mechanism for doing this is the array. Although they are extremely useful for many purposes, arrays have

some inherent limitations. They provide only a very simple mechanism for storing and accessing data. Moreover, their capacity must be known at construction time because there is no way to make an array bigger or smaller. Java has always had arrays and also some additional classes, such as the `Vector` and `Hashtable` classes, to allow you to manipulate groups of objects. Since JDK 1.2, however, a significant feature supports much more generalized collection management: the Collections API. The Certification Exam objectives now require that you have a grasp of the concepts of this new functionality.

The Collections API is a mechanism for manipulating object references. Although arrays are capable of storing primitives or references, collections are not. If you need to take advantage of collection functionality to manipulate primitives, you have to wrap the primitives in the wrapper classes that were presented earlier in this chapter.

The Collections API is often referred to as a *framework*. That is, the classes have been designed with a common abstraction of data container behavior in mind, ensuring uniform semantics wherever possible. At the same time, each implemented collection type is free to optimize its own operations. The factory class `java.util.Collections` supplements support for these types, which are discussed next, with a variety of static helper methods. These methods support operations such as synchronizing the container, establishing immutability, and executing binary searches. With these classes in place, programmers are no longer required to build their own basic data structures from scratch.

A class or group of classes is considered *threadsafe* if any thread can call any method of any instance at any time. The collections framework as a whole is not threadsafe. If you use collections in a multithreaded environment, you are generally responsible for protecting the integrity of the encapsulated data. One way to accomplish this is to use certain static factory methods of the `java.util.Collections` class. The names of these methods begin with "synchronized" (such as `synchronizedList()` and `synchronizedMap()`); they are well documented in the API.

Collection Types

There are several different collections. They vary, for example, in the storage mechanisms used, in the way they can access data, and in the rules about what data can be stored. The Collections API provides a variety of interfaces and some concrete implementation classes covering these variations.

A general interface, `java.util.Collection`, defines the basic framework for collections. This interface stipulates the methods that allow you to add items, remove items, determine if items are in the collection, and count the number of items in the collection. A collection is sometimes known as a *bag* or a *multiset*. A simple collection places no constraints on the type, order, or repetition of elements within the collection.

Some collections are ordered; that is, there is a clear notion of one item following another. A collection of this kind is commonly known as a *list* or a *sequence*. In some lists, the order is the order in which items are added to the collection; in others, the elements themselves are

assumed to have a natural order, and that order is understood by the list. In the Java Collections API, the interface `java.util.List` defines a basic framework for collections of this sort.

If a collection imposes the specific condition that it cannot contain the same value more than once, then it is known as a *set*. The interface `java.util.Set` defines the basic framework for this type of collection. In some sets, the `null` value is a legitimate entry; but if it is allowed, `null` can occur only once in a set.

The final type of specialized behavior directly supported by the Collections API is known as a *map*. A map uses a set of key values to look up, or index, the stored data. For example, if you store an object representing a person, then as the key value you could either use that person's name or some other unique identifier such as a social security number or employee ID number. Maps are particularly appropriate for implementing small online databases, especially if the data being stored will usually be accessed via the unique identifier. It is a requirement for a map that the key be unique, and for this reason if you were storing data about a person in a map, the name would not make a very good key—it is quite possible for two people to have the same name.

Let's take a moment to recap these points:

- A collection has no special order and does not reject duplicates.
- A list is ordered and does not reject duplicates.
- A set has no special order but rejects duplicates.
- A map supports searching on a key field, values of which must be unique.

Of course, it is possible for combinations of these behaviors to be meaningful. For example, a map might also be ordered. However, the Certification Exam only requires you to understand these four fundamental types of collection.

The storage associated with any one collection can be implemented in many ways, but the Collections API implements the four methods that are most widely used: an array, a linked list, a tree, or a hash table. Each of these techniques has benefits and constraints. Let's consider these benefits and constraints for each storage technique in turn.

Array *Array* storage tends to be fast to access, but it is relatively inefficient as the number of elements in the collection grows or if elements need to be inserted or deleted in the middle of a list. These limitations occur because the array itself is a fixed sequence. Adding or removing elements in the middle requires that all the elements from that point onward must be moved up or down by one position. Adding more data once the array is full requires a whole new array to be allocated and the entire contents to be copied over to the new array. Another limitation of an array is that it provides no special search mechanism. Despite these weaknesses, an array can still be an appropriate choice for data that is ordered, does not change often, and does not need to be searched much.

Linked list A *linked list* allows elements to be added to or removed from the collection at any location in the container, and it allows the size of the collection to grow arbitrarily without the penalties associated with array copying. This improvement occurs because each element is an individual object that refers to the next (and sometimes previous, in a double-linked list) element in the list. However, it is significantly slower to access by index than an array, and it still provides no special search mechanism. Because linked lists can insert new elements at arbitrary

locations, however, they can apply ordering very easily, making it a simple (if not always efficient) matter to search a subset, or range, of data.

Tree A *tree*, like a linked list, allows easy addition and deletion of elements and arbitrary growth of the collection. Unlike lists, trees insist on a means of ordering. In fact, constructing a tree requires that there be some comparison mechanism to the data being stored—although this can be created artificially in some cases. A tree will usually provide more efficient searching than either an array or a linked list, but this benefit may be obscured if unevenly distributed data is being stored.

Hash table A *hash table* requires that some unique identifying key can be associated with each data item, which in turn provides efficient searching. Hashes still allow a reasonably efficient access mechanism and arbitrary collection growth. Hashing may be inappropriate for small data sets, however, because some overhead typically is associated with calculating the hash values and maintaining the more complex data structure associated with this type of storage. Without a sufficiently large number of elements that would justify the operational costs, the overhead of a hashing scheme may cancel out or outweigh the benefits of indexed access.

Collections, Equality, and Sorting

Sets maintain uniqueness of their elements. Maps maintain uniqueness of their keys. It is important to understand how uniqueness is defined. Set elements and map keys are considered unique if they are unequal, and equality is tested with the `equals()` method, not with the == operator. (The difference between these two concepts of equality is discussed in Chapter 1, "Language Fundamentals.")

Consider the following code fragment (assume that the set is initially empty):

```
1.  Integer aSix = new Integer(6);
2.  Integer anotherSix = new Integer(6);
3.  mySet.add(aSix);
4.  mySet.add(anotherSix);
```

When line 3 executes, the instance of `Integer` referenced by `aSix` is added to the set. When line 4 executes, the instance of `Integer` referenced by `anotherSix` is compared to the set's contents using the `equals()` method. Although the two instances of `Integer` are distinct objects, the `equals()` call returns true and so the two objects are considered equal. Thus line 4 has no effect on the set.

Certain collection implementations are ordered. To work properly, the elements of these collections must implement the interface `java.lang.Comparable`, which defines `compareTo()`, a method for determining the inherent order of two items of the same type. Most implementations of `Map` will also require a correct implementation of the `hashCode()` method.

It is advisable to keep these three methods in mind whenever you define a new class, even if you do not anticipate storing instances of this class in collections.

The *hashCode()* Method

Many maps rely on hashing algorithms to provide efficient implementations. The Programmer's Exam does not require you to know the details of how this is accomplished. However, maps can only hash effectively if the objects they manage implement reasonable `hashCode()` methods. The exam tests your ability to write correct `hashCode()` methods.

The purpose of `hashCode()` is to provide a single integer to represent an object. Two objects that are equal (as determined by the `equals()` method) should return the same hashcode; otherwise hashing maps will not be able to reliably store and retrieve those objects. Note that the reverse is not required; that is, it is not necessary that two unequal objects must return different hashcodes. However, a `hashcode()` method that really does return different values for unequal objects is likely to result in better hashing map performance. This requirement and its corollaries are know as the hashcode "contract." The contract is spelled out formally on the API page for `java.lang.Object` under the `hashcode()` method's entry.

Any time you create a class that might be inserted into a hashing map, you should give your class a `hashCode()` method. A reasonable approach is to return a value that is a function of those variables that are scrutinized by the `equals()` method. Consider the following simple class:

```
class HashMe {
  private int a, b, c;
  public boolean equals(Object that) {
    if (!(that instanceof HashMe))
      return false;
    HashMe h = (HashMe)that;
    return (a == h.a  &&  b == h.b);
  }
}
```

Note that the `equals()` method considers the values of `a` and `b` but ignores `c`. We will look at three alternative ways to provide this class with a `hashcode()` method.

A very simple approach is

```
public int hashCode() { return 49; }
```

This method returns a constant value. Certainly any two instances of `HashMe` will generate identical hashcodes, so the contract is obeyed, but a map that contains many instances of this class could perform very slowly.

A more reasonable version would be

```
public int hashCode() { return a+b; }
```

This version returns a value that is a function of the two variables that are scrutinized by the `equals()` method. There will be times when unequal instances will return equal hashcodes. For example, one instance might have `a=1` and `b=2`, while another instance has `a=-5` and `b=8`. Both

objects will return a hashcode of 3. Presumably this will be rare enough that maps will be able to function efficiently.

A third approach is the following:

```
public int hashCode() { return (int)(Math.pow(2,a) + Math.pow(3,b)); }
```

This version will only return equal hashcodes on equal objects. Thus the hashing inefficiency of the first example is completely ignored. However, this `hashcode()` method is itself not very efficient because it involves three double-precision operations and a number of type conversions.

Collection Implementations in the API

A variety of concrete implementation classes are supplied in the Collections API to implement the interfaces `Collection`, `List`, `Set`, and `Map`, using different storage types. Some of them are listed here:

`HashMap/Hashtable` These two classes are very similar; both use hash-based storage to implement a map. `Hashtable` has been in the Java API since the earliest releases, and `HashMap` was added at JDK 1.2. The main difference between the two is that `Hashtable` does not allow the `null` value to be stored, although it makes some efforts to support multithreaded use.

The `List` and `Set` interfaces extend the `Collection` interface. The `Map` interface does not extend `Collection`.

`HashSet` This is a set, so it does not permit duplicates and it uses hashing for storage.

`LinkedList` This is an implementation of a list, based on a linked list storage.

`TreeMap` This class provides an ordered map. The elements must be orderable, either by implementing the `Comparable` interface or by providing a `Comparator` class to perform the comparisons.

`TreeSet` This class provides an ordered set, using a tree for storage. As with the `TreeMap`, the elements must have an order associated with them.

`Vector` This class, which has been in the Java API since the first release, implements a list using an array internally for storage. The array is dynamically reallocated as necessary, as the number of items in the vector grows.

`Stack` This class implements a last-in, first-out (LIFO) collection of objects.

Real World Scenario

A Sortable, Reversible Stack

Here's a scenario in which you will write a class that implements a stack with additional properties.

The class should have a constructor that populates the stack with a specified number of instances of the `Character` class. These `Characters` should be random and unique. Provide methods called `sort()` and `reverse()`, which should respectively sort and reverse the contents of the stack. Provide a `main()` method that demonstrates the functionality of the constructor and of the `sort()` and `reverse()` methods. Here is a sample:

```
>java CleverStack
Initial State: CZqnWSNMyu
       Sorted: CMNSWZnquy
     Reversed: yuqnZWSNMC
```

A second run produces different output because the stack is initialized with different random `Characters`:

```
>java CleverStack
Initial State: DZBpUlPMbI
       Sorted: BDIMPUZblp
     Reversed: plbZUPMIDB
```

Without collections, initializing, sorting, and reversing would require a lot of original code. With the proper use of the classes in the `java.util` package, these operations require only a few lines each.

As an additional exercise, consider what would happen if the `Characters` in the stack did not have to be unique. (Thus, for example, it would be legal to have an initial state of AAAqwertyu.) Construction would become easier because the constructor would no longer need to prevent duplication, and reversing would not be affected. However, sorting would be profoundly affected. Think about why this is the case, and think about ways to implement sorting of nonunique elements. The solution is beyond the scope of this chapter.

Collections and Code Maintenance

There is no such thing as the "best implementation" of a collection. Using any kind of collection involves several kinds of overhead penalty: memory usage, storage time, and retrieval time. No implementation can optimize all three of these features. So, instead of looking for the best list or the best hash table or the best set, it is more reasonable to look for the most appropriate list, set, or hash table implementation for a particular programming situation.

As a program evolves, its data collections tend to grow. A collection that was created to hold a little bit of data may later be required to hold a large amount of data, while still providing reasonable response time. It is prudent from the outset to design code in such a way that it is easy to substitute one collection implementation type for another. Java's collections framework makes this easy because of its emphasis on interfaces. This section presents a typical scenario.

Imagine a program that maintains data about shoppers who are uniquely identified by their e-mail addresses. Such a program might use a `Shopper` class, with instances of this class stored in some kind of hash table, keyed by e-mail address. Suppose that when the program is first written, it is known that there are and always will be only three shoppers. The following code fragment constructs one instance for each shopper and stores the data in a hash map; then the map is passed to various methods for processing:

```
 1. private void getShoppers() {
 2.   Shopper sh1 = getNextShopper();
 3.   String email1 = getNextEmail();
 4.   Shopper sh2 = getNextShopper();
 5.   String email2 = getNextEmail();
 6.   Shopper sh3 = getNextShopper();
 7.   String email3 = getNextEmail();
 8.
 9.   Map map = new HashMap();  // Very important!
10.   map.put(email1, sh1);
11.   map.put(email2, sh2);
12.   map.put(email3, sh3);
13.
14.   findDesiredProducts(map);
15.   shipProducts(map);
16.   printInvoices(map);
17.   collectMoney(map);
18. }
```

Note the declaration of `map` on line 9. The reference type is `Map`, not `HashMap` (the interface, rather than the class). This is a very important difference whose value will become clear later on. The four processing methods do not much concern us here. Just consider their declarations:

```
private void findDesiredProducts(Map map) { ... }
private void shipProducts (Map map) { ... }
private void printInvoices (Map map) { ... }
private void collectMoney (Map map) { ... }
```

Imagine that each of these methods passes the hash map to other subordinate methods, which pass it to still other methods; our program has a large number of processing methods. Note that the argument types will be `Map`, not `HashMap` (again, the interface, rather than the class).

As development proceeds, suppose it becomes clear that the `getShoppers()` method should return the map's keys (which are the shoppers' e-mail addresses) in a sorted array. Because there are and always will be only three shoppers, there are and always will be only three keys to sort; the easiest implementation is therefore as follows:

```
 1. private String[] getShoppers() {  // New return type
 2.   Shopper sh1 = getNextShopper();
 3.   String email1 = getNextEmail();
 4.   Shopper sh2 = getNextShopper();
 5.   String email2 = getNextEmail();
 6.   Shopper sh3 = getNextShopper();
 7.   String email3 = getNextEmail();
 8.
 9.   Map map = new HashMap();
10.   map.put(email1, sh1);
11.   map.put(email2, sh2);
12.   map.put(email3, sh3);
13.
14.   findDesiredProducts(map);
15.   shipProducts(map);
16.   printInvoices(map);
17.   collectMoney(map);
18.
19.   // New sorting code.
20.   String[] sortedKeys = new String[3];
21.   if (email1.compareTo(email2) < 0  &&
22.       email1.compareTo(email3) < 0) {
23.     sortedKeys[0] = email1;
24.     if (email2.compareTo(email3) < 0)
25.       sortedKeys[1] = email2;
26.     else
27.       sortedKeys[2] = email3;
28.    }
29.    else if (email2.compareTo(email3) < 0) {
30.     sortedKeys[0] = email2;
31.     if (email1.compareTo(email3) < 0)
32.       sortedKeys[1] = email1;
33.     else
34.       sortedKeys[2] = email3;
35.    }
36.    else {
```

```
37.       sortedKeys[0] = email3;
38.       if (email1.compareTo(email2) < 0)
39.         sortedKeys[1] = email1;
40.       else
41.      sortedKeys[2] = email2;
42.    }
43.    return sortedKeys;
44. }
```

The added code is fairly lengthy: 26 lines.

Beware of specs claiming that the size of anything is and always will be small.

Predictably, as soon as the code is developed and debugged, someone will decide that the program needs to be expanded to accommodate 20 shoppers instead of the original 3. The new requirement suggests the need for a separate sorting algorithm, in its own separate method. The new method will be called `sortStringArray()`. The next evolution of `getShoppers()` looks like this:

```
 1. private String[] getShoppers() {
 2.    String[] keys = new String[20];
 3.    Map map = new HashMap()
 4.    for (int i=0; i<20; i++) {
 5.      Shopper s = getNextShopper();
 6.      keys[i] = getNextEmail();
 7.      map.put(keys[i], s);
 8.    }
 9.
10.    findDesiredProducts(map);
11.    shipProducts(map);
12.    printInvoices(map);
13.    collectMoney(map);
14.
15.    sortStringArray(keys);
16.    return keys;
17. }
```

This code is much more modular and compact. However, it is still not mature. The next requirement is that it has to be able to handle any number of shoppers, even a very large number.

At first glance, the solution seems very simple: just pass the number of shoppers in to the method, as shown here:

```
 1. private String[] getShoppers(int nShoppers) {
 2.   String[] keys = new String[nShoppers];
 3.   Map map = new HashMap()
 4.   for (int i = 0; i < nShoppers; i++) {
 5.     Shopper s = getNextShopper();
 6.     keys[i] = getNextEmail();
 7.     map.put(keys[i], s);
 8.   }
 9.
10.   findDesiredProducts(map);
11.   shipProducts(map);
12.   printInvoices(map);
13.   collectMoney(map);
14.
15.   sortStringArray(keys);
16.   return keys;
17. }
```

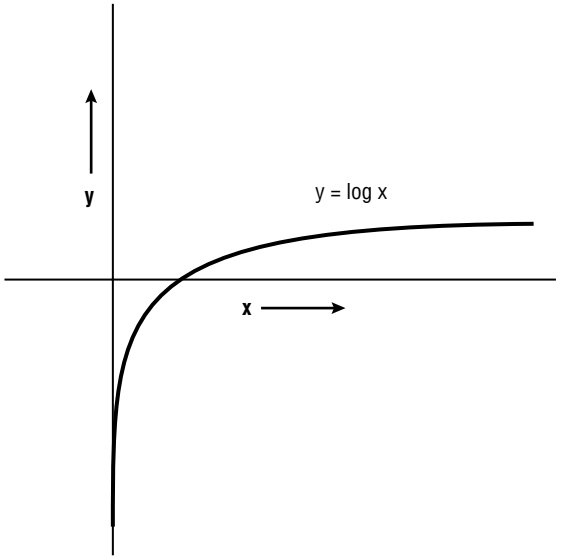
This code seems fine until the number of shoppers crosses some threshold. Then the amount of time spent sorting the keys (in method `sortStringArray()`, called on line 15) becomes prohibitive. Now is the time when the `Collections` framework shows its true value. In particular, you are about to see the value of referencing the map with variables of type `Map`, rather than `HashMap` (the interface, rather than the class).

Because the sorting method is now the bottleneck, it is reasonable to wonder whether a different kind of map can solve the performance problem. It is time for a quick look at the API pages for the classes that implement the `Map` interface. You'll find a suitable alternative: the `TreeMap` class. This implementation maintains its keys in sorted order and has a method for returning them in sorted order. Because the keys are always sorted, there seems to be zero overhead for sorting. Actually, the situation is not quite so good—there must be some extra overhead (which you can hope will be slight) in the `put()` method, when the tree map stores a new key. Before deciding that `TreeMap` is the right class to use, it is important to ascertain that storing and retrieving data in the new collection will not cost an unreasonable amount of time, even if the map is very large.

First, what is the current cost of storing and retrieving in a hash map? The API page for `HashMap` says that storage and retrieval take constant time, no matter what the size of the map might be. This is ideal; let's hope the performance of a tree map will also be constant. If it is not constant, it must still be acceptable when the data collection is large.

The API page for `TreeMap` says that the class "provides guaranteed log(n) time cost" for various operations, including storage and retrieval. This means that the time to store and retrieve data grows with the logarithm of the size of the data set. Figure 8.5 shows a graph of the logarithm function.

FIGURE 8.5 The logarithm function



The graph in the figure rises steadily, but at an ever-decreasing rate. The cost for accessing a large tree map is only slightly greater than the cost for accessing a small one. Logarithmic overhead is almost as good as constant overhead; it is certainly acceptable for the current application.

Apparently, the `TreeMap` class is a very good substitute for the original `HashMap` class. Now you see how easy it is to replace one collection implementation with another. Because all references to the hash table are of type `Map` (the interface) rather than type `HashMap` (the class), only one line of code needs to be modified: the line in which the hash table is constructed. That line originally was

```
Map map = new HashMap();
```

All that is required is to call a different constructor:

```
Map map = new TreeMap();
```

Many data-processing methods pass references to the hash table back and forth among themselves. Not one of these methods needs to be modified at all. In fact, the only major change that needs to be made is to dispense with the `sortStringArray()` method and the call to it, substituting the tree map's intrinsic functionality. This modification is not directly relevant to the main point of this example, which is how easy it is to replace one collection type with another. However, it is instructive to see how the modification is accomplished. The final code looks like this:

```
1. private String[] getShoppers(int nShoppers) {
2.   Map map = new TreeMap();
3.   for (int i=0; i< nShoppers; i++) {
```

```
 4.      map.put(getNextEmail(), getNextShopper());
 5.    }
 6.
 7.    findDesiredProducts(map);
 8.    shipProducts(map);
 9.    printInvoices(map);
10.    collectMoney(map);
11.
12.    String[] keys = new String[nShoppers];
13.    Iterator iter = map.keySet().iterator();
14.    int i = 0;
15.    while (iter.hasNext())
16.      keys[i++] = (String)iter.next();
17.    return keys;
18. }
```

An *iterator* is an object that returns the elements of a collection one by one. Here the iterator on line 13 returns the elements of the hash table's key set. Because the hash table is an instance of `TreeMap`, the key set is guaranteed to be sorted.

This example shows the importance of referencing collections with variables of interface rather than class type. If you do this, replacing one collection type with another type becomes trivially easy.

Summary

The `java.lang` package contains classes that are indispensable to Java's operation, so all the classes of the package are automatically imported into all source files. Some of the most important classes in the package are

- `Object`
- `Math`
- The wrapper classes
- `String`
- `StringBuffer`

In a string context, addition operands are appended in turn to a string buffer, which is then converted to a string; primitive operands are converted to strings, and objects are converted by having their `toString()` methods invoked.

The `java.util` package contains many utilities, but for the Certification Exam, the Collections API is of primary interest. Collections provide ways to store and retrieve data in a program. Different types of collection provide different rules for storage, and different collection implementations optimize different access and update behaviors.

Summary of Collections

The essential points we've covered about collections are

- Collections impose no order nor restrictions on content duplication.
- Lists maintain an order (possibly inherent in the data, possibly externally imposed).
- Sets reject duplicate entries.
- Maps use unique keys to facilitate lookup of their contents.

For storage:

- Using arrays makes insertion, deletion, and growing the store more difficult.
- Using a linked list supports insertion, deletion, and growing the store, but makes indexed access slower.
- Using a tree supports insertion, deletion, and growing the list. Indexed access is slow, but searching is faster.
- Using hashing supports insertion, deletion, and growing the store. Indexed access is slow, but searching is particularly fast. However, hashing requires the use of unique keys for storing data elements.

Exam Essentials

Understand the common methods of the `Math` class. These methods are summarized in Table 8.1.

Understand the functionality of the wrapper classes. Each of the eight primitive types has a corresponding wrapper class.

Understand the functionality of the `String` class. The encapsulated text is immutable. Strings are supported by the string literal pool.

Understand the functionality of the `StringBuffer` class. The encapsulated text is mutable. String concatenation via the + operator is implemented with behind-the-scenes string buffers.

Know the main characteristics of each kind of Collections API: `List`, `Set`, and `Map`. Be aware that `List` maintains order, `Set` prohibits duplicate members, and `Map` associates keys with values.

Understand how collections test for duplication and equality. Collections use the `equals()` method rather than the == operator.

Understand that collection classes are not threadsafe. Most implementation classes are not threadsafe. You should assume that a collection class is not threadsafe unless its API documentation explicitly states otherwise.

Understand why it is preferable for references to collections to have interface type rather than class type. Be aware of the maintenance benefits when substituting one implementing class for another.

Key Terms

Before you take the exam, be certain you are familiar with the following terms:

array
framework
hash table
iterator
linked list
list
map
sequence
set
string context
threadsafe
tree
wrapper class

Review Questions

1. Given a string constructed by calling `s = new String("xyzzy")`, which of the calls modify the string? (Choose all that apply.)

   A. `s.append("aaa");`

   B. `s.trim();`

   C. `s.substring(3);`

   D. `s.replace('z', 'a');`

   E. `s.concat(s);`

   F. None of the above

2. Which one statement is true about the following code?

```
1. String s1 = "abc" + "def";
2. String s2 = new String(s1);
3. if (s1 == s2)
4.   System.out.println("== succeeded");
5. if (s1.equals(s2))
6.   System.out.println(".equals() succeeded");
```

   A. Lines 4 and 6 both execute.

   B. Line 4 executes and line 6 does not.

   C. Line 6 executes and line 4 does not.

   D. Neither line 4 nor line 6 executes.

3. Suppose you want to write a class that offers static methods to compute hyperbolic trigonometric functions. You decide to subclass `java.lang.Math` and provide the new functionality as a set of static methods. Which one statement is true about this strategy?

   A. The strategy works.

   B. The strategy works, provided the new methods are public.

   C. The strategy works, provided the new methods are not private.

   D. The strategy fails because you cannot subclass `java.lang.Math`.

   E. The strategy fails because you cannot add static methods to a subclass.

4. Which one statement is true about the following code fragment?

```
1. import java.lang.Math;
2. Math myMath = new Math();
3. System.out.println("cosine of 0.123 = " +
4.   myMath.cos(0.123));
```

   A. Compilation fails at line 2.

   B. Compilation fails at line 3 or 4.

   C. Compilation succeeds, although the import on line 1 is not necessary. During execution, an exception is thrown at line 3 or 4.

   D. Compilation succeeds. The import on line 1 is necessary. During execution, an exception is thrown at line 3 or 4.

   E. Compilation succeeds and no exception is thrown during execution.

5. Which one statement is true about the following code fragment?

```
1. String s = "abcde";
2. StringBuffer s1 = new StringBuffer("abcde");
3. if (s.equals(s1))
4.   s1 = null;
5. if (s1.equals(s))
6.   s = null;
```

   A. Compilation fails at line 1 because the `String` constructor must be called explicitly.

   B. Compilation fails at line 3 because `s` and *`s1`* have different types.

   C. Compilation succeeds. During execution, an exception is thrown at line 3.

   D. Compilation succeeds. During execution, an exception is thrown at line 5.

   E. Compilation succeeds. No exception is thrown during execution.

6. In the following code fragment, after execution of line 1, `sbuf` references an instance of the `StringBuffer` class. After execution of line 2, `sbuf` still references the same instance.

```
1. StringBuffer sbuf = new StringBuffer("abcde");
2. sbuf.insert(3, "xyz");
```

   A. True

   B. False

7. In the following code fragment, after execution of line 1, `sbuf` references an instance of the `StringBuffer` class. After execution of line 2, `sbuf` still references the same instance.

```
1. StringBuffer sbuf = new StringBuffer("abcde");
2. sbuf.append("xyz");
```

A. True

B. False

8. In the following code fragment, line 4 is executed.

```
1. String s1 = "xyz";
2. String s2 = "xyz";
3. if (s1 == s2)
4.   System.out.println("Line 4");
```

A. True

B. False

9. In the following code fragment, line 4 is executed.

```
1. String s1 = "xyz";
2. String s2 = new String(s1);
3. if (s1 == s2)
4.   System.out.println("Line 4");
```

A. True

B. False

10. Which would be most suitable for storing data elements that must not appear in the store more than once, if searching is not a priority?

A. `Collection`

B. `List`

C. `Set`

D. `Map`

E. `Vector`

Answers to Review Questions

1. F. Strings are immutable.

2. C. Because `s1` and `s2` are references to two different objects, the == test fails. However, the strings contained within the two `String` objects are identical, so the `equals()` test passes.

3. D. The `java.lang.Math` class is final, so it cannot be subclassed.

4. A. The constructor for the `Math` class is private, so it cannot be called. The `Math` class methods are static, so it is never necessary to construct an instance. The import at line 1 is not required, because all classes of the `java.lang` package are automatically imported.

5. E. A is wrong because line 1 is a perfectly acceptable way to create a `String` and is actually more efficient than explicitly calling the constructor. B is wrong because the argument to the `equals()` method is of type `Object`; thus any object reference or array variable may be passed. The calls on lines 3 and 5 return false without throwing exceptions because s and s1 are objects of different types.

6. A. The `StringBuffer` class is mutable. After execution of line 2, `sbuf` refers to the same object, although the object has been modified.

7. A. The `StringBuffer` class is mutable. After execution of line 2, `sbuf` refers to the same object, although the object has been modified.

8. A. Line 1 constructs a new instance of `String` and stores it in the string pool. In line 2, `"xyz"` is already represented in the pool, so no new instance is constructed.

9. B. Line 1 constructs a new instance of `String` and stores it in the string pool. Line 2 explicitly constructs another instance.

10. C. A set prohibits duplication, whereas a list or collection does not. A map also prohibits duplication of the key entries, but maps are primarily for looking up data based on the unique key. So, in this case, you could use a map, storing the data as the key and leaving the data part of the map empty. However, you are told that searching is not a priority, so the proper answer is a set.

The Developer's Exam

PART II