Chapter 1

# Fundamentals

## SCJP EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- ✓ **Given a code example and a scenario, write code that uses the appropriate access modifiers, package declarations, and import statements to interact with (through access or inheritance) the code in the example.**
- ✓ **Given an example of a class and a command line, determine the expected runtime behavior.**
- ✓ **Determine the effect upon object references and primitive values when they are passed into methods that perform assignments or other modifying operations on the parameters.**
- ✓ **Given a code example, recognize the point at which an object becomes eligible for garbage collection, determine what is and is not guaranteed by the garbage collection system, and recognize the behaviors of the `Object.finalize()` method.**
- ✓ **Given the fully-qualified name of a class that is deployed inside and/or outside a JAR file, construct the appropriate directory structure for that class. Given a code example and a classpath, determine whether the classpath will allow the code to compile successfully.**
- ✓ **Write code that correctly applies the appropriate operators including assignment operators (limited to: `=`, `+=`, `-=`), arithmetic operators (limited to: `+`, `-`, `*`, `/`, `%`, `++`, `--`), relational operators (limited to: `<`, `<=`, `>`, `>=`, `==`, `!=`), the `instanceof` operator, logical operators (limited to: `&`, `|`, `^`, `!`, `&&`, `||`), and the conditional operator ( `? :` ), to produce a desired result. Write code that determines the equality of two objects or two primitives.**

Java is an interpretive, object-oriented programming language that Sun Microsystems developed. A considerable benefit of writing Java applications is that they run in a Java Runtime Environment (JRE) that is well defined. As a Java programmer, you know your Java program is going to run on a Java Virtual Machine (JVM), regardless of the device or operating system. Consequently, you know an `int` is 32 bits and signed, a `boolean` is `true` or `false`, method arguments are passed by value, and the garbage collector cleans up your unreachable objects whenever it feels like it. (Okay, not every aspect of Java is an exact science!) The point is that Java runs in a precise environment, and passing the SCJP exam requires a strong knowledge of these well-defined Java fundamentals.

This chapter covers the fundamentals of Java programming, including writing Java classes, running Java applications, creating packages, defining classpath, and using the Java operators. We will also discuss the details of garbage collection and call by value.

# Writing Java Classes

The exam objectives state that you need to be able to "write code that uses the appropriate access modifiers, package declarations, and imports statements." In other words, you need to be able to write Java classes, which makes sense because Java is an object-oriented programming (OOP) language and writing classes is an essential aspect of OOP. Your executable Java code will appear within the definition of a class. A *class* describes an object, which is a noun in your program. The object can either represent something tangible, like a television or an employee, or it can represent something less obvious but just as useful in your program, like an event handler or a stream of data being read from a file.

An *object* is an instance of a class. Think of a class as a blueprint for a house, and the object as the house. Another common analogy is to think of a class as a recipe for cookies, and the objects are the cookies. (We will discuss the details of instantiating objects in Chapter 2, "Declarations, Initialization, and Scoping.") Because classes are a fundamental aspect of Java programming, the certification exam assumes you are familiar with the rules for writing them, and in this section we cover these details.

For starters, a Java class must be defined in a text file with a `.java` extension. In addition, if the class is declared `public`, then the name of the file must match the name of the class. Consequently, a `.java` file can only contain at most one top-level `public` class. For example, the following class definition must appear in a file named `Cat.java`:

```
public class Cat {
  public String name;
  public int weight;
}
```

Compiled Java code is referred to as *bytecode*, and the name of the bytecode file matches the name of the class. Compiling the `Cat.java` source file creates a bytecode file named `Cat.class`.

### Line Numbers

Java source files do not contain line numbers. However, the classes on the exam display line numbers. If the numbering starts with a 1, then the entire definition of a source file is being displayed. If the numbering starts with some other value, then only a portion of a source file is being displayed. You will see this explanation in the instructions at the beginning of the SCJP exam.

Java allows multiple classes in a single `.java` file as long as no more than one of the top-level classes is declared `public`. The compiler still generates a separate `.class` file for each class defined in the `.java` file. For example, suppose a file named `Customer.java` contains the following two class definitions:

```
1.  public class Customer {
2.    public String name;
3.    public String address;
4.  }
5.
6.  class Order {
7.    public int partNumber;
8.    public int quantity;
9.    public boolean shipped;
10. }
```

Compiling `Customer.java` generates two files: `Customer.class` and `Order.class`. Note that the `Order` class cannot be `public` because `Customer` is already `public`, nor can `Order` be `protected` or `private` because Java does not allow top-level classes to be `protected` or `private`. Therefore, `Order` must have the default access, often referred to as friendly or package-level access, meaning only classes within the same package can use the `Order` class. (We discuss packages in the next section.)

**Access Specifiers for Top-Level Classes**

A top-level class has two options for an access modifier: `public` or package-level access (often called the default access). Keep an eye out for exam questions that declare a top-level class as `private` or `protected`. For example, the following code will not compile:

```
//Generates a compiler error: "modifier private not allowed here"
private class HelloWorld {
    public static void main(String [] args) {
        System.out.println(args[1] + args[2]);
    }
}
```

**Multiple Classes in a Single File**

Java allows multiple top-level classes to be defined in a single file, but in the real world this is rarely done. We typically want our classes to be public, and only top-level classes can be public. That being said, the exam might contain questions that define multiple classes in a single source file because it is convenient and many questions on the exam involve more than one class.

# Packages

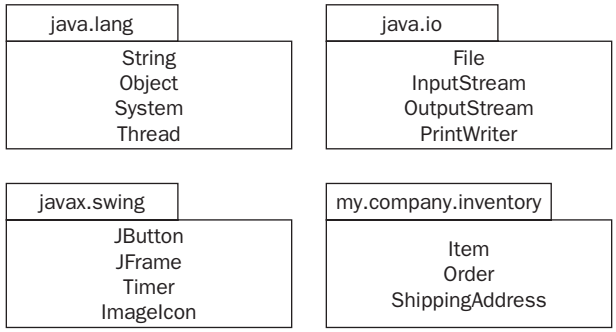
The exam objectives state that you need to be able to "write code that uses the appropriate package declarations and import statements," and I can assure you there will be more than one question on the exam testing your knowledge of the `package` keyword and its effect on a Java class. This section discusses the details you need to know about Java packages. A *package* is a grouping of classes and interfaces. It can also contain enumerations and annotated types, but because these are special types of classes and interfaces, I will refer to items in a package as simply classes and interfaces for brevity. This grouping of classes and interfaces is typically based on their relationship and usage. For example, the `java.io` package contains classes and interfaces related to input and output. The `java.net` package contains the classes and interfaces related to networking. There are two key benefits of using packages in Java:

- Packages organize Java programs by grouping together related classes and interfaces.
- Packages create a namespace for your classes and interfaces.

The Application Programming Interface (API) for the Java Platform, Standard Edition (Java SE) contains hundreds of packages that you can use in any Java SE application. As

a Java programmer, you will create your own packages for the classes that you develop. Packages are often drawn as tabbed folders, as shown in Figure 1.1.

**FIGURE 1.1** When designing a Java application, packages are drawn as tabbed folders.




To view all of the packages in the Java SE API, visit the API documentation at `http://java.sun.com/javase/6/docs/api/`. This web page contains three frames. The upper-left frame is a list of all the packages. Clicking a package displays its classes and interfaces in the lower-left frame. Clicking a class or interface in the lower-left frame displays its documentation page in the main frame. You should spend time browsing the Java API documentation! I find it extremely useful, especially when using a Java class or interface for the first time.

If you are developing a Java program with hundreds of classes and interfaces, grouping related types into packages provides a much-needed organization to the project. In addition, the namespace provided by a package is useful for avoiding naming conflicts.

This section discusses these two benefits of packages in detail. I will start with a discussion on the `package` keyword and then cover the details of imports, the `CLASSPATH` environment variable, and the directory structure required for packages.

## The *package* Keyword

The `package` keyword puts a class or interface in a package, and it must be the first line of code in your source file (aside from comments, which can appear anywhere within a source file). For example, the following `Employee` class is declared in the `com.sybex.payroll` package:

```
package com.sybex.payroll;

public class Employee {
  public Employee() {
      System.out.println(
          "Constructing a com.sybex.payroll.Employee");
  }
}
```

Putting a class in a package has two important side effects that you need to know for the exam:

1. The fully qualified name of a class or interface changes when it is in a package. The package name becomes a prefix for the class name. For example, the fully qualified name of the `Employee` class shown earlier is `com.sybex.payroll.Employee`.
2. The compiled bytecode file must appear in a directory structure on your file system that matches the package name. For example, a `.class` file for any class or interface in the `com.sybex.payroll` package must appear in a directory structure matching `\com\sybex\payroll\`. You can either create this directory structure yourself or use the `-d` flag during compilation and the compiler will create the necessary directory structure for you. We discuss the `-d` flag in detail later in this section.

The fully qualified name of the `Employee` class is `com.sybex.payroll.Employee`. Other classes that want to use the `Employee` class need to refer to it by its fully qualified name. For example, the following program creates an instance of the `Employee` class:

```
public class CreateEmployee {
  public static void main(String [] args) {
      com.sybex.payroll.Employee e =
          new com.sybex.payroll.Employee();
  }
}
```

Here's the output of the `CreateEmployee` program:

```
Constructing a com.sybex.payroll.Employee
```

**The Unnamed Package**

If a class is not specifically declared in a package, then that class belongs to the *unnamed package*. Classes and interfaces in the unnamed package cannot be imported into a source file. You should only use the unnamed package when writing simple classes and interfaces that are not being used in a production application. In the real world, you will rarely write a Java class or interface that is not declared in a package. Your classes will appear in a package name that contains your company's Internet domain name, which the next section discusses.

## The *import* Keyword

As you can see by the `CreateEmployee` program, using the fully qualified name of a class can be tedious and makes for a lot of typing! The `import` keyword makes your life as a coder easier by allowing you to refer to a class in a source file without using its fully qualified name.

The `import` keyword is used to import a single class or, when used with the wildcard (*), an entire package. A source file can have any number of `import` statements, and they must appear after the `package` declaration and before the `class` declaration. Importing classes and packages tells the compiler that you are not going to use fully qualified names for classes. The compiler searches your list of imports to determine the fully qualified names of the classes referenced in the source file.

Here is the `CreateEmployee` program again, except this time the `com.sybex.payroll.Employee` class is imported, allowing the `Employee` class to be referred to without using its fully qualified name:

```
import com.sybex.payroll.Employee;

public class CreateEmployee2 {
   public static void main(String [] args) {
       Employee e = new Employee();
   }
}
```

The output is the same as before:

```
Constructing a com.sybex.payroll.Employee
```

In fact, the compiled bytecode files `CreateEmployee.class` and `CreateEmployee2.class` are completely identical (except for the number 2 that appears in `CreateEmployee2.class`). The `import` statement does not affect the compiled code. Behind the scenes, the compiler removes the `import` statement and replaces each occurrence of `Employee` with `com.sybex.payroll.Employee`.

### What Does Import Mean?

The term *import* sounds like something is being brought into your source file, but nothing is physically added to your source code by importing a class or package. An `import` statement is strictly to make your life as a programmer easier. The Java compiler removes all `import` statements and replaces all the class names in your source code with their fully qualified names. For this reason, you never need to use `import` statements. Instead, you can use fully qualified names throughout your source files. However, you will quickly discover the benefit of `import` statements, especially when you work with long package names.

The `CreateEmployee` and `CreateEmployee2` programs both refer to the `String` class. `String` is defined in the `java.lang` package, but this package was not imported. The `java.lang` package is unique in that the compiler automatically imports all the public classes and

interfaces of `java.lang` into every source file, so there is never any need to import types from `java.lang` (although it is perfectly valid to do so).

The following program demonstrates an `import` statement that uses the wildcard to `import` an entire package. The program uses the `File`, `FileReader`, `BufferedReader`, and `IOException` classes, all found in the `java.io` package. The program reads a line of text from a file named `mydata.txt`.

```
1.  import java.io.*;
2.
3.  public class ReadFromFile {
4.    public static void main(String [] args) {
5.        File file = new File("mydata.txt");
6.        FileReader fileReader = null;
7.        try {
8.            fileReader = new FileReader(file);
9.            BufferedReader in = new BufferedReader(fileReader);
10.            System.out.println(in.readLine());
11.        }catch(IOException e) {
12.            e.printStackTrace();
13.        }
14.    }
15. }
```

Because nothing is actually included into your source file by the `import` keyword, using the wildcard does not impact the size of your bytecode files. However, common practice in Java is to avoid using the wildcard because it may lead to ambiguity when two packages are imported that share a common class name. For example, the following code does not compile because there is a class called `AttributeList` in both the `javax.swing.text.html.parser` package and the `javax.management` package:

```
1. import javax.swing.text.html.parser.*;
2. import javax.management.*;
3.
4. public class ImportDemo {
5.    public AttributeList a;
6. }
```

The `ImportDemo` class generates the following compiler error:

```
reference to AttributeList is ambiguous, both class
 javax.management.AttributeList in javax.management and class
 javax.swing.text.html.parser.AttributeList in
 javax.swing.text.html.parser match
    public AttributeList a;
```

If you ever are in a situation where you need to use two classes with the same name but in different packages, then using imports does not work. You will need to refer to each class by their fully qualified name in your source file. The following code compiles successfully:

```
1. public class FullyQualifiedDemo {
2.     public javax.management.AttributeList a1;
3.     public javax.swing.text.html.parser.AttributeList a2;
4. }
```

The `FullyQualifiedDemo` program demonstrates why packages are often referred to as namespaces because package names are used to avoid naming conflicts. Without packages, there is no way for the compiler or the JVM to distinguish between the two `AttributeList` classes. However, because the two `AttributeList` classes are declared in different packages, they can be referred to by their fully qualified names to avoid any ambiguity.

**Naming Convention for Packages**

The namespace ambiguity situation can still occur if programmers happen to use the same package names in different programs. If you and I both write a class called `Dog` and we both define `Dog` in a package named `pets`, then a naming conflict still occurs. However, the standard Java naming convention for a package name is to use your company's domain name (in reverse) as a prefix to your package names. For example, a class written by an employee of Sybex uses a package name that starts with `com.sybex`.

Subsequent components of the package name may include your department and project name, followed by a descriptive name for the package. For example, `com.sybex.scjpbook.pets` is a good package name for a class named `Dog` that appears in this book. It is extremely unlikely that someone else would use this package name, although I am sure there are other `Dog` classes in the world.

If everyone who writes Java code follows this naming convention for package names, then naming conflicts can only occur within a single company or project, making it easier to resolve the naming conflict.

## Package Directory Structure

The exam objectives state that "given the fully-qualified name of a class that is deployed inside and/or outside a JAR file," you need to be able to "construct the appropriate directory structure for that class." This objective refers to the required directory structure that results from using packages. In addition to creating a namespace, packages organize your programs by grouping related classes and interfaces together. One result of using packages is that the bytecode of a class or interface must appear in a directory structure that matches its package name. If you do not put your bytecode in the proper directory structure, the compiler or the JVM will be unable to find your classes.

Suppose we have the following class definition:

```
package com.sybex.payroll;

public class Employee {
   public Employee() {
       System.out.println(
           "Constructing a com.sybex.payroll.Employee");
   }
}
```

This `Employee` class is in the `com.sybex.payroll` package, so its compiled file `Employee.class` must be in a directory with a pathname `\com\sybex\payroll`. This requires a directory named `\com`, which can appear anywhere on your file system. Inside `\com` you must have a `\sybex` subdirectory, which must contain a `\payroll` subdirectory.

The `\com` directory can appear anywhere on your file system. A common technique is to put your source files in a directory named `\src` and your bytecode files in a directory named `\build`. For example, suppose the `Employee` source file is in the following directory:

```
c:\myproject\src\com\sybex\payroll\Employee.java
```

Suppose you want the compiled code to be in the `c:\myproject\build` directory. You can use the `-d` flag of the compiler to achieve this. The `-d` flag has two effects:

- The compiled code will be output in the directory specified by the `-d` flag.
- The appropriate directory structure that matches the package names of the classes is created automatically in the output directory.

Consider the following compiler command, executed from the `c:\myproject\src` directory:

```
javac -d c:\myproject\build .\com\sybex\payroll\Employee.java
```

The `-d` flag specifies the output directory as `c:\myproject\build`. Assuming the class compiles successfully, the compiler creates the file `Employee.class` in the following directory:

```
c:\myproject\build\com\sybex\payroll\Employee.class
```

Keep in mind the directory `c:\myproject\build` is arbitrary; we could have output the bytecode into the directory of our choosing. After you start putting bytecode in arbitrary directories on your file system, the compiler and the JVM need to know where to look to find it. They look for the bytecode files in your classpath, an important concept that the next section discusses in detail.

## The *CLASSPATH* Environment Variable

The exam objectives state that "given a code example and a classpath," you need to be able to "determine whether the classpath will allow the code to compile successfully." The *classpath* refers to the path on your file system where your `.class` files are saved, and the classpath is defined by the `CLASSPATH` environment variable. The `CLASSPATH` environment variable specifies the directories and JAR files where you want the compiler and the JVM to search for bytecode. Using `CLASSPATH` allows your bytecode to be stored in the directory of your choosing, as well as in multiple directories or Java archive (JAR) files.

For example, suppose you have a class named `com.sybex.payroll.Employee`. The compiler and the JVM look for the `\com\sybex\payroll` directory structure by searching your `CLASSPATH` environment variable. For example, if `Employee.class` is in the following directory:

```
c:\Documents and Settings\Rich\workspaces\build\com\sybex\payroll
```

then your `CLASSPATH` needs to include the directory:

```
c:\Documents and Settings\Rich\workspaces\build
```

The `CLASSPATH` environment variable can contain any number of directories and JAR files. Setting `CLASSPATH` on Windows can be done from a command prompt using a semicolon to separate multiple values:

```
set CLASSPATH="c:\Documents and Settings\Rich\workspaces\build";
 c:\myproject\build;c:\tomcat\lib\servlet.jar;.;
```

In this example, the compiler and the JVM look for bytecode files in the two `\build` directories specified, the `servlet.jar` file in `c:\tomcat\lib`, and the current working directory (represented by the dot). The double quotes are necessary in the first directory because of the spaces in the pathname.

On Unix, you use the `setenv` command and colons to separate multiple values. For example:

```
setenv CLASSPATH /usr/build:/myproject/build:/tomcat/lib/servlet.jar
```

A common mistake new Java programmers make is to include part of the package pathname in the `CLASSPATH`. If you are struggling with classes not being found, you might be tempted to try the following command line:

```
set CLASSPATH=c:\myproject\build\com\sybex\payroll;
```

Including `\com\sybex\payroll` in your `CLASSPATH` does not work! Do not add any of the package directories to your `CLASSPATH`, only the parent directory. The compiler and the JRE will look for the appropriate subdirectories.

CLASSPATH plays a key role in compiling and running your Java applications, which we discuss in the next section.

# Running Java Applications

The SCJP certification exam tests your knowledge of running a Java program from the command line using an appropriate CLASSPATH. If you are using Sun's Java Development Kit (JDK), then `java.exe` in the `\bin` folder of the JDK directory is the executable used to run your Java applications. The sample commands in this book assume `java.exe` is in your path.

The entry point of a Java program is `main`, which you can define in any class. The signature of `main` must look like this:

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

The only changes you can make to this signature are the name of the parameter `args`, which can be arbitrary, and the order of `public` and `static`. For example, the following declaration is a valid signature of `main`:

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

In addition, you can specify the array of `String` objects using the syntax for variable-length arguments:

```
public static void main(String... args)
```

**Variable-Length Arguments**

As of Java 5.0, a method in Java can declare a variable-length argument list denoted by the ellipsis (. . .). Variable-length arguments are discussed in detail in Chapter 2.

The `args` array contains the command-line arguments, discussed in detail later in this section. The `main` method has to be `public` so that the JVM has access to it, and making it `static` allows the JVM to invoke this method without having to instantiate an instance of the containing class.

Let's start with a simple example. Suppose the following class is saved in the `c:\myproject` directory. First, does the following `SaySomething` class compile, and does it successfully declare the `main` method?

```
1. public class SaySomething {
2.     private static String message = "Hello!";
3.
4.     public static void main() {
5.         System.out.println(message);
6.     }
7. }
```

The answers are yes and no. Yes, this class compiles, but no, it does not define `main` properly. A `static` method can access a `static` field in the same class, so there is no problem with the `message` field. Also, you can write a method called `main` that does not have an array of `String` objects, so the compiler will not complain about the `main` method defined on line 4. However, this class cannot be executed as a Java application because it does not successfully declare the proper `main` method for a Java application.

Let's try it again, this time with the following `SayHello` class. Does this class compile and successfully declare the `main` method?

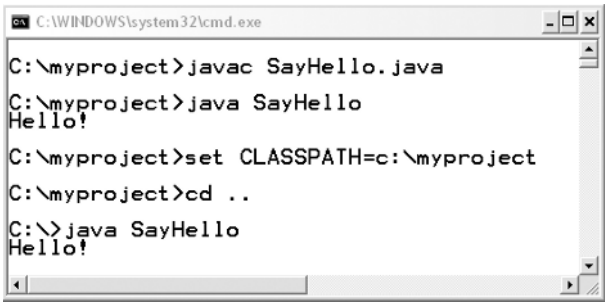
```
1. public class SayHello {
2.     private static String message = "Hello!";
3.
4.     public static void main(String [] args) {
5.         System.out.println(message);
6.     }
7. }
```

The answer is yes to both: `SayHello` compiles and declares the proper version of `main` so that it can be executed as a stand-alone Java application. The following command line runs the `SayHello` application:

```
java SayHello
```

This command line assumes that you run the command from the directory that contains the file `SayHello.class`, which in our case is `c:\myproject`. If you want to run this Java application from any directory (instead of just `c:\myproject`), you need to include `c:\myproject` in your `CLASSPATH`. Figure 1.2 shows `SayHello` being executed from `c:\myproject`, and then being executed from `c:\` after the `CLASSPATH` is correctly set.

**FIGURE 1.2** Compiling and running the `SayHello` program from a command prompt



### Specifying the Class Name

The command line for `java.exe` requires the name of the class that contains `main`. Notice that the name of the class is not the same as the name of the bytecode file, which in the `SayHello` example is `SayHello.class`. The following command line does not work:

```
java SayHello.class
```

The JVM looks for a class named `class` in the `SayHello` package (which it will not find) and throws a `NoClassDefFoundError`. The JVM only needs the name of the class; it will find the corresponding bytecode file by scanning all the directories and JAR files set in your CLASSPATH environment variable. If you do not set a CLASSPATH, the JVM looks in the current working directory.

The exam will likely test your knowledge with a more complex example where the class containing `main` is in a package. Let's look at another example, starting with a class called `ColorChanger` in the `com.sybex.events` package:

```
1.  package com.sybex.events;
2.
3.  import java.awt.Component;
4.  import java.awt.Color;
5.  import java.awt.event.*;
6.
7.  public class ColorChanger implements ActionListener {
8.    private Component container;
9.
10.    public ColorChanger(Component c) {
11.        container = c;
12.    }
13.
14.    public void actionPerformed(ActionEvent e) {
15.        String color = e.getActionCommand();
16.        if(color.equals("red")) {
17.            container.setBackground(Color.RED);
18.        } else if(color.equals("blue")) {
19.            container.setBackground(Color.BLUE);
20.        } else {
21.            container.setBackground(Color.WHITE);
22.        }
23.    }
24. }
```

The source file ColorChanger.java is saved in c:\myproject\src\com\sybex\events and the class is compiled using the following command executed from c:\myproject\src:

```
javac -d c:\myproject\build .\com\sybex\events\ColorChanger.java
```

This command line creates ColorChanger.class in the c:\myproject\build\com\sybex\events directory. The following program contains main and tests the ColorChanger class:

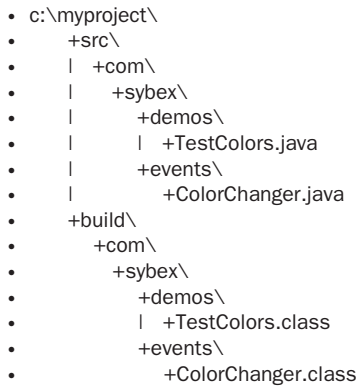
```
1.  package com.sybex.demos;
2.
3.  import com.sybex.events.ColorChanger;
4.  import java.awt.Button;
5.  import java.awt.Color;
6.  import java.awt.event.ActionEvent;
7.
8.  public class TestColors {
9.
10.    public static void main(String [] args) {
11.        Button b = new Button("Testing...");
12.        b.setBackground(Color.GREEN);
13.        System.out.println("Color is " + b.getBackground());
14.
15.        ColorChanger cc = new ColorChanger(b);
16.        ActionEvent action = new ActionEvent(b,
17.                ActionEvent.ACTION_PERFORMED,
18.                "blue");
19.        cc.actionPerformed(action);
20.        System.out.println("Now the color is "
21.                + b.getBackground());
22.    }
23. }
```

TestColors.java is saved in the c:\myproject\src\com\sybex\demos directory. Because TestColors is not in the same package as ColorChanger, it imports the ColorChanger class. TestColors is compiled using the following command executed from the c:\myproject\src directory:

```
javac -d c:\myproject\build .\com\sybex\demos\TestColors.java
```

This command line creates TestColors.class in the directory c:\myproject\build\com\sybex\demos. Figure 1.3 shows the directory structure after compiling the source files with -d.

**FIGURE 1.3** The source code and bytecode are typically stored in separate folders.




A typical exam question at this point is to ask what the `CLASSPATH` needs to be for you to run the `TestColors` program at the command prompt from any working directory. Do you know the answer? I will reveal it in a moment, but first here is the command prompt that runs the `TestColors` application if you execute it from the `c:\myproject\build` directory:

```
java com.sybex.demos.TestColors
```

Notice the fully qualified class name of `TestColors` must be specified to execute properly. Using the fully qualified name has nothing to do with `CLASSPATH` or the current working directory. The following command does not work and results in a `java.lang.NoClassDefFoundError`, no matter what directory you run it from or what your `CLASSPATH` is set to:

```
java TestColors
```

Why will this never work? Because there is no class called `TestColors`. Remember, putting a class in a package changes the name of the class. Because `TestColors` is in the `com.sybex.demos` package, the name of the class is `com.sybex.demos.TestColors`, and that name must be used on the command line.

By the way, the answer to the question earlier about `CLASSPATH` is it needs to contain `c:\myproject\build`:

```
set CLASSPATH=c:\myproject\build;
```

With this `CLASSPATH`, the command to run the `TestColors` program can be executed from any directory.

**Don't Panic During the Exam!**

The purpose of the `ColorChanger` and `TestColors` example is to demonstrate running a Java application from a command line, so what the code does is not relevant in this situation. If you are not familiar with the `Container` and `ActionListener` classes, a `ColorChanger` can listen to action events of a GUI component in Java because it implements `ActionListener`. When an action event occurs, the `actionPerformed` method is invoked, which changes the background color of the given GUI component.

You might encounter a situation on the exam where you are not familiar with some of the classes in the given code. Don't panic! Focus on what the exam question is asking before trying to figure out what the code is doing. You might discover that the behavior of the code is irrelevant because the question is testing you on a different facet of the language.

You can also set the classpath for the JVM on the command line using the `-classpath` flag, which is discussed in the next section, followed by a discussion on running Java code stored in JAR files.

## The *-classpath* Flag

The `java` command that starts the JVM has a `-classpath` flag that allows the classpath to be specified from the command line. This is a common technique for ensuring the classpath is pointing to the right directories and JAR files. Using the `-classpath` flag overrides the `CLASSPATH` environment variable.

For example, we could run the `TestColors` program using the following command prompt executed from any directory:

```
java -classpath c:\myproject\build com.sybex.demos.TestColors
```

If you have multiple directories or JAR files, use a semicolon on a Windows machine to separate them on the `-classpath` flag. For example, the following command line adds the current directory to the classpath:

```
java -classpath c:\myproject\build;. com.sybex.demos.TestColors
```

On a Unix machine, use a colon to separate multiple directories and JAR files:

```
java -classpath /myproject/build:. com.sybex.demos.TestColors
```

The `java` command can also define the classpath using the `-cp` flag, which is just a shortcut for the `-classpath` flag.

## JAR Files

Bytecode can be stored in archived, compressed files known as *JAR files*. JAR is short for Java archive. The compiler and the JVM can find bytecode files in JAR files without needing to uncompress the files onto your file system. JAR files are the most common way

to distribute Java code, and the exam tests your understanding of JAR files and how they relate to CLASSPATH.

The JDK comes with the tool jar.exe for creating and extracting JAR files. The following command adds the bytecode files of the c:\myproject\build directory to a new JAR file named myproject.jar:

```
C:\myproject\build>jar -cvf myproject.jar .
added manifest
adding: com/(in = 0) (out= 0)(stored 0%)
adding: com/sybex/(in = 0) (out= 0)(stored 0%)
adding: com/sybex/demos/(in = 0) (out= 0)(stored 0%)
adding: com/sybex/demos/TestColors.class(in = 1209) (out= 671)(deflated 44%)
adding: com/sybex/events/(in = 0) (out= 0)(stored 0%)
adding: com/sybex/events/ColorChanger.class(in = 883) (out= 545)(deflated 38%)
adding: com/sybex/payroll/(in = 0) (out= 0)(stored 0%)
adding: com/sybex/payroll/Employee.class(in = 402) (out= 292)(deflated 27%)
```

The -c flag is for creating a new JAR file. The -v flag tells the jar command to be verbose while it is processing files. The -f flag is for denoting the filename of the new JAR file, which in this example is myproject.jar. After the filename, you specify the files or directories to include in the JAR. In our example, because all of our bytecode was conveniently located in the \build directory, we simply added the entire contents of c:\myproject\build, using the dot to represent the current directory.

### JAR Files and Package Names

If a class is in a package, then the JAR file must contain the appropriate directory structure when the .class file is included in the JAR. Notice in the verbose output of the jar command shown earlier, the necessary \com directory and subdirectories matching our package names are added to the JAR.

You can add a JAR file to your CLASSPATH. In fact, it is common to have lots of JAR files in your CLASSPATH. The following example demonstrates adding myproject.jar to the CLASSPATH of a Windows machine, then running the TestColors program (which is in myproject.jar):

```
C:\>set CLASSPATH=c:\myproject\build\myproject.jar;

C:\>java com.sybex.demos.TestColors
Color is java.awt.Color[r=0,g=255,b=0]
Now the color is java.awt.Color[r=0,g=0,b=255]
```

**Real World Scenario**

**Separating Source Code and Bytecode Files**

You might have been wondering why the examples in this chapter separated the source files from the bytecode files. In general, when you distribute your code you do not want the JAR files to include your source code. Having the bytecode separate makes it much easier to create JAR files that only contain your bytecode.

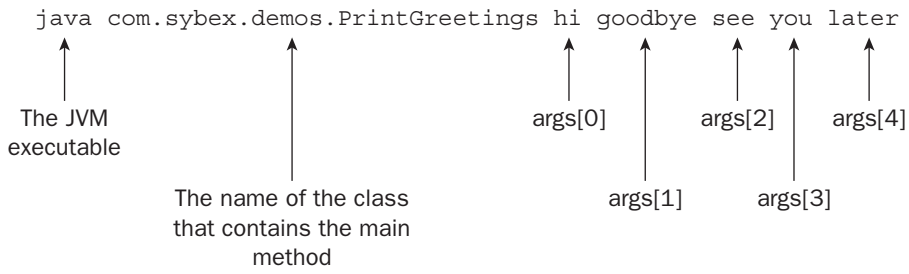
You might have also noticed that the source code files in `\src` use the same directory structure as their package names. This is not a requirement for your `.java` files; they can be stored in any directory. In most development teams, you will be required to run the `javadoc` tool on your source files to generate the HTML documentation for your classes and interfaces. The `javadoc` tool requires that your source file directories match the package names. The exam does not contain any questions that involve the `javadoc` tool, but in the real world you will quickly learn to appreciate the benefits of `javadoc` documentation!

In projects I work on, we put source code in the `\src` directory, using the package name subdirectory structure. Bytecode goes in a subdirectory of `\build` depending on whether or not the bytecode is in a JAR. JAR files appear in the `\build\lib` directory, and .class files appear in the `\build\classes` subdirectory that matches the package name structure.

## Command-Line Arguments

The `java.exe` executable starts the JVM, and on the command line you provide the name of the class that contains the `main` method. The command-line arguments are passed into the `main` method as a single array of `String` objects. For example, suppose `PrintGreetings` is a class that contains `main` and it is executed with the command line in Figure 1.4.

**FIGURE 1.4** This command line starts the JVM and invokes the `main` method in the `PrintGreetings` class.



This command has five command-line arguments, so the first element in the `String` array is `"hi"`, the second element in the array is `"goodbye"`, and so on. The following `PrintGreetings` class contains a `for` loop that iterates through the command-line arguments and outputs them to the console:

```
1.  package com.sybex.demos;
2.  public class PrintGreetings {
3.          public static void main(String [] args) {
4.             for(int i = 0; i < args.length; i++) {
5.                          System.out.println(args[i]);
6.                 }
7.         }
8.  }
```

If `PrintGreetings` is executed with the command line in Figure 1.4, then the output looks like this:

```
hi
goodbye
see
you
later
```

### Command-Line Arguments on the Exam

Notice that the first command-line argument in the array is `args[0]` because Java uses zero-based indexes for arrays. The exam creators seem to like questions about arrays and command-line arguments, so don't be surprised if you see a question that tests both topics at the same time. For example, what is the output of the `DoSomething` class when executed with the following command?

```
java DoSomething one two
```

```
1. public class DoSomething {
2.     public static void main(String args []) {
3.         System.out.print(args[1]);
4.         System.out.print(args[2]);
5.     }
6. }
```

> By the way, the square brackets following `args` instead of preceding `args` are perfectly valid in Java, although not common practice. The output of this program is the string `"two"` followed by an `ArrayIndexOutOfBoundsException` on line 4, as shown here:
>
> ```
> twoException in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2
>         at DoSomething.main(DoSomething.java:4)
> ```
>
> The length of `args` is two, so `args[2]` is beyond the end of the array.

All command-line arguments are treated as `String` objects, even if they represent another data type. The wrapper classes in `java.lang` contain helpful methods for parsing strings into other data types. Consider the following `ParseDemo` program:

```
1.  public class ParseDemo {
2.    public static void main(String [] args) {
3.        System.out.println("Processing " + args.length +
4.                           " arguments");
5.         int x = Integer.parseInt(args[0]);
6.        System.out.println(x);
7.        boolean b = Boolean.parseBoolean(args[1]);
8.        System.out.println(b);
9.        float f = Float.parseFloat(args[2]);
10.        System.out.println(f);
11.        char c = args[3].charAt(0);
12.        System.out.println(c);
13.    }

14. }
```

Here is a command line that runs the `ParseDemo` program, followed by its output:

```
c:\myproject>java ParseDemo 34567 false 3.14159 R
Processing 4 arguments
34567
false
3.14159
R
```

There is no need to parse a `String` into a `char` because the `String` already is an array of characters. The `ParseDemo` program simply selects the first character in the `String` to "convert" it to a `char`.

We now turn our attention to a discussion on garbage collection, which first requires an understanding of the differences between reference types and primitive types.

# Reference vs. Primitive Types

Java applications contain two types of data: *primitive types* and *reference types*. In this section, we will discuss the differences between a primitive type and a reference type. The differences are important when we discuss garbage collection later in this chapter.

## Primitive Types

Java has eight built-in data types, referred to as the *Java primitive types*. These eight data types represent the building blocks for Java objects, because all Java objects are just a complex collection in memory of these primitive data types. The SCJP exam assumes you are well versed in the eight primitive data types, their size, and what can be stored in them. Table 1.1 shows the Java primitive types together with their size in bytes and the range of values that each holds.

**TABLE 1.1** The Java Primitive Data Types

| Primitive Type | Size | Range of Values (inclusive) |
|---|---|---|
| `byte` | 8 bits | –128 to 127 |
| `short` | 16 bits | –32768 to 32767 |
| `int` | 32 bits | –2147483648 to 2147483647 |
| `long` | 64 bits | –9223372036854775808 to 9223372036854775807 |
| `float` | 32 bits | $2^{-149}$ to $(2 - 2^{-23}) \cdot 2^{127}$ |
| `double` | 64 bits | $2^{-1074}$ to $(2 - 2^{-52}) \cdot 2^{1023}$ |
| `char` | 16 bits | '\u0000' to '\uffff' (0 to 65535) |
| `boolean` | unspecified | true or false |

**Do I Need to Memorize These Sizes?**

Not all of them. Don't try to memorize the range of values in a `long`, `float`, or `double`, but it is important to know their size in bits. However, you should be able to state the range of a `byte` exactly and recognize when a `short` or `int` has likely gone beyond its range. Expect a question involving the size of a `char`, especially because a `char` in C/C++ is only 8 bits and uses the ASCII format, while a Java `char` is 16 bits and uses the UNICODE format.

Primitive types are allocated in memory by declaring them in your code. For example, the following lines of code declare an `int` and a `double`:

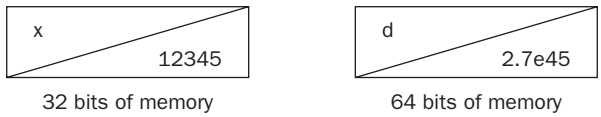
```
int x;
double d;
```

In memory, the compiler allocates 32 bits for the variable `x` and 64 bits for the variable `d`. A primitive type can only store a value of that same type. For example, the variable `x` can only hold an `int` and `d` can only hold a `double`. Suppose we assign values to `x` and `d`:

```
x = 12345;
d = 2.7e45;
```

Figure 1.5 shows how these primitive types look in memory. The value 12345 is stored in the memory where `x` is allocated. Similarly, the value 2.7e45 is stored in the memory where `d` is allocated.

**FIGURE 1.5** An `int` is 32 bits and a `double` is 64 bits.




## Reference Types

Reference types are variables that are class types, interface types, and array types. A reference refers to an object (an instance of a class). Unlike primitive types that hold their values in the memory where the variable is allocated, references do not hold the value of the object they refer to. Instead, a reference "points" to an object by storing the memory address where the object is located, a concept referred to as a *pointer*. However, the Java language does not allow a programmer to access a physical memory address in any way, so even though a reference is similar to a pointer, you can only use a reference to gain access to the fields and methods of the object it refers to. It is impossible to determine the actual address stored in the memory of the reference variable.

Let's take a look at some examples that declare and initialize reference types. Suppose we declare a reference of type `java.util.Date` and a reference of type `String`:

```
java.util.Date today;
String greeting;
```

The `today` variable is a reference of type `Date` and can only point to a `Date` object. The `greeting` variable is a reference that can only point to a `String` object. A value is assigned to a reference in one of two ways:

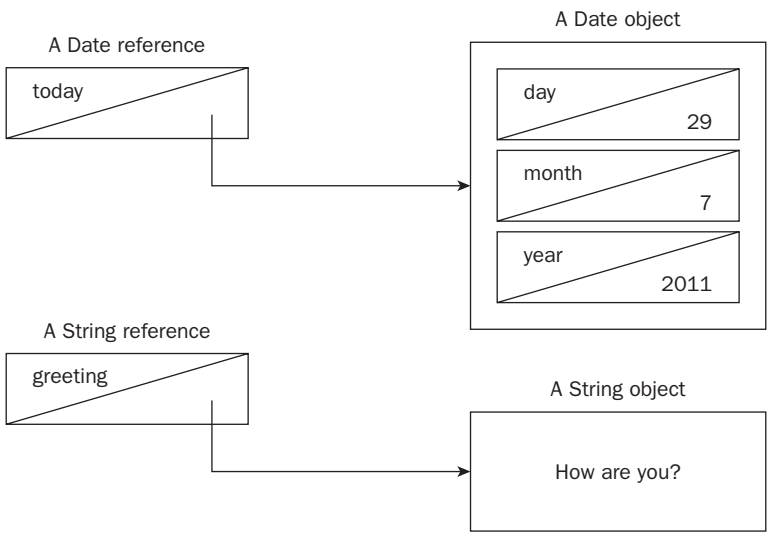
- A reference can be assigned to another reference of the same type.
- A reference can be assigned to a new object using the `new` keyword.

For example, the following statements assign these references to new objects:

```
today = new java.util.Date();
greeting = "How are you?";
```

The `today` reference now points to a new `Date` object in memory, and `today` can be used to access the various fields and methods of this `Date` object. Similarly, the `greeting` reference points to a new `String` object, `"How are you?"` The `String` and `Date` objects do not have names and can only be accessed via their corresponding reference. Figure 1.6 shows how the reference types appear in memory.

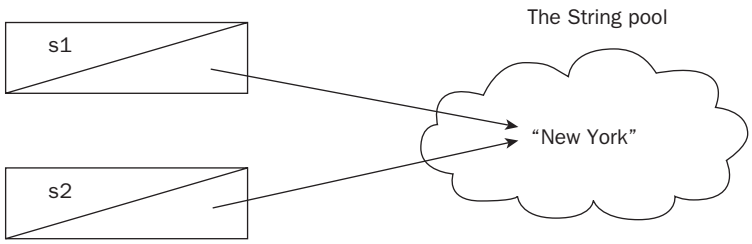
**FIGURE 1.6** An object in memory can only be accessed via a reference.



### String Literals and the String Pool

The new keyword is not required for creating the `String` object `"How are you?"` because it is a string literal. String literals get special treatment by the JVM. Behind the scenes, the JVM instantiates a `String` object for `"How are you?"` and stores it in the *string pool*. The `greeting` reference refers to this `String` object in the pool. Because `String` objects in Java are *immutable* (which means they cannot be changed), the JVM can optimize the use of string literals by allowing only one instance of a string in the pool. For example, the following two `String` references actually point to the same string in the pool, as shown in the following diagram:

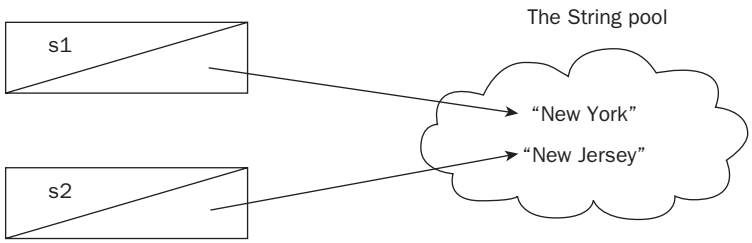
```
String s1 = "New York";
String s2 = "New York";
```




You might think if the two references point to the same object, then changing one object would inadvertently change the value of the other. But `String` objects are immutable, so the following statement only changes `s2`:

```
s2 = "New Jersey";
```

The reference `s2` now points to `"New Jersey"`, but `s1` still points to `"New York"`, as shown in the following diagram:



In addition, arrays in Java are objects and therefore have a reference type. The Java language implicitly defines a reference type for each possible array type: one for each of the eight primitive types and also an `Object` array. This allows for references of the following type:

```
int [] grades;
String [] args;
Runnable [] targets;
```

### The `null` Type

There is a special data type in Java for `null`. The `null` type does not have a name, so it is not possible to declare a variable to be the `null` type. However, you can assign any reference to the `null` type:

```
String firstName = null;
Runnable [] targets = null;
```

Primitive types cannot be assigned to `null`, only references. The following statement is not valid:

```
int x= null;    //does not compile
```

We can also assign a reference to another reference as long as their data types are compatible. For example, the following code assigns two `ArrayList` references to each other:

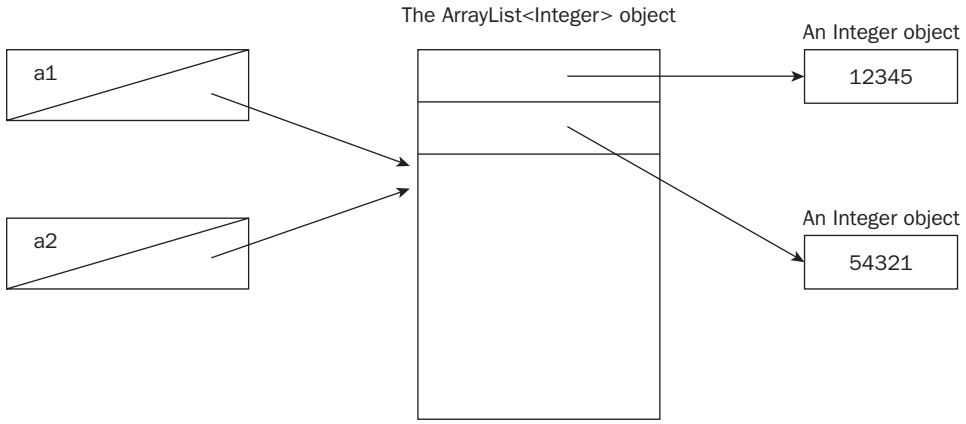
```
java.util.ArrayList<Integer> a1 =
       new java.util.ArrayList<Integer>();
java.util.ArrayList<Integer> a2 = a1;
```

The references `a1` and `a2` both point to the same object, an `ArrayList` that contains `Integer` objects. (Two references pointing to the same object is a common occurrence in Java.) The `ArrayList` object can be accessed using either reference. Examine the following code and determine if it compiles successfully and, if so, what its output is:

```
a1.add(new Integer(12345));
a2.add(new Integer(54321));
for(int i = 0; i < a1.size(); i++) {
  System.out.println(a2.get(i));
}
```

The code adds an `Integer` to the `ArrayList` using `a1`, and then adds another `Integer` using `a2`. Because they point to the same `ArrayList`, the list now has two `Integer` objects in it, as shown in Figure 1.7.

**FIGURE 1.7** The `ArrayList` object can be accessed using either `a1` or `a2`.




The `for` loop compiles successfully and the output looks like this:

```
12345
54321
```

Let's look at a different example. Examine the following code that assigns two references to each other and determine if it compiles successfully:

```
java.math.BigDecimal bd = new java.math.BigDecimal(2.75);
String s = bd;
```

The reference `bd` is of type `BigDecimal`, and `s` is of type `String`. These two classes are not compatible, so assigning `s` to `bd` generates the following compiler error:

```
incompatible types
found   : java.math.BigDecimal
required: java.lang.String
        String s = bd;
```

Even using the cast operator does not fix the problem. The following code generates a similar compiler error, except this time the compiler complains the types are inconvertible:

```
java.math.BigDecimal bd = new java.math.BigDecimal(2.75);
String s = (String) bd;
```

The compiler error looks like this:

```
inconvertible types
found   : java.math.BigDecimal
required: java.lang.String
        String s = (String) bd;
```

Even though `s` and `bd` are both references that behind the scenes are identical in terms of memory consumption (most likely they are 32-bit unsigned integers, but this is JVM-dependent), it is not possible to assign them to each other because there is no relationship between a `String` object and a `BigDecimal` object. Two references are compatible only when either the objects they point to are the same type or one of the objects is a child class of the other. `String` and `BigDecimal` have no inheritance relationship.

Hopefully you have a better understanding of the differences between references and primitive types. References play a key role in understanding garbage collection, our next topic.

# Garbage Collection

All Java objects are stored in your program memory's *heap*. The heap, which is also referred to as the *free store,* represents a large pool of unused memory allocated to your Java application. The heap may be quite large, depending on your environment, but there is always a limit to its size. If your program keeps instantiating objects and leaving them on the heap, eventually it will run out of memory.

Garbage collection refers to the process of automatically freeing memory on the heap by deleting objects that are no longer reachable in your program. Every JVM has a garbage collector, and many different algorithms are used to determine the efficiency and timing of garbage collection. The SCJP exam does not test your knowledge of any individual garbage collection algorithm. However, you do need to know "what is and is not guaranteed by the garbage collection system," as well as "recognize the point when an object becomes eligible for garbage collection." This section discusses both of these objectives in detail.

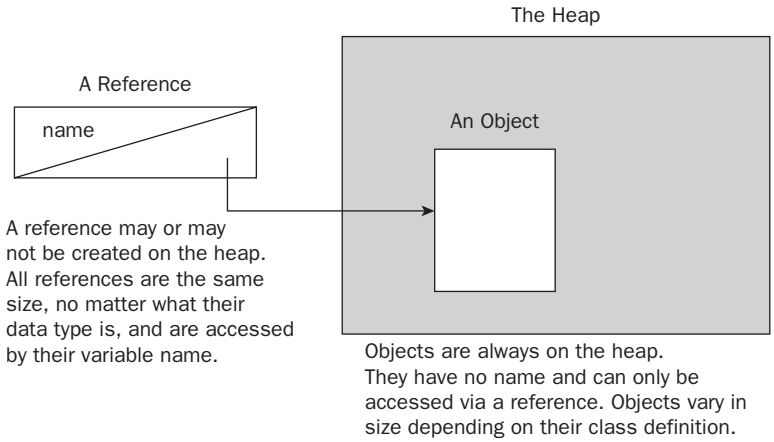
The `new` keyword instantiates a new object on the heap and returns a reference to the object. Typically you will save that object's reference in a variable. An object will remain on the heap until it is no longer reachable. An object is no longer reachable when one of two situations occurs:

- The object no longer has any references pointing to it.
- All references to the object have gone out of scope.

### Objects vs. References

Do not confuse a reference with the object that it refers to. They are two different entities. The reference is a variable that has a name and can be used to access the contents of an object. A reference can be assigned to another reference, passed to a method, or returned from a method. All references are the same size, no matter what their type is. A reference is most likely 32 bits, but their actual size depends on your JVM.

An object sits on the heap and does not have a name. Therefore, you have no way to access an object except through a reference. Objects come in all different shapes and sizes and consume varying amounts of memory. An object cannot be assigned to another object, nor can an object be passed to a method or returned from a method. It is the object that gets garbage collected, not its reference.




Realizing the difference between a reference and an object goes a long way toward understanding garbage collection, call by value, the new operator, and many other facets of the Java language.

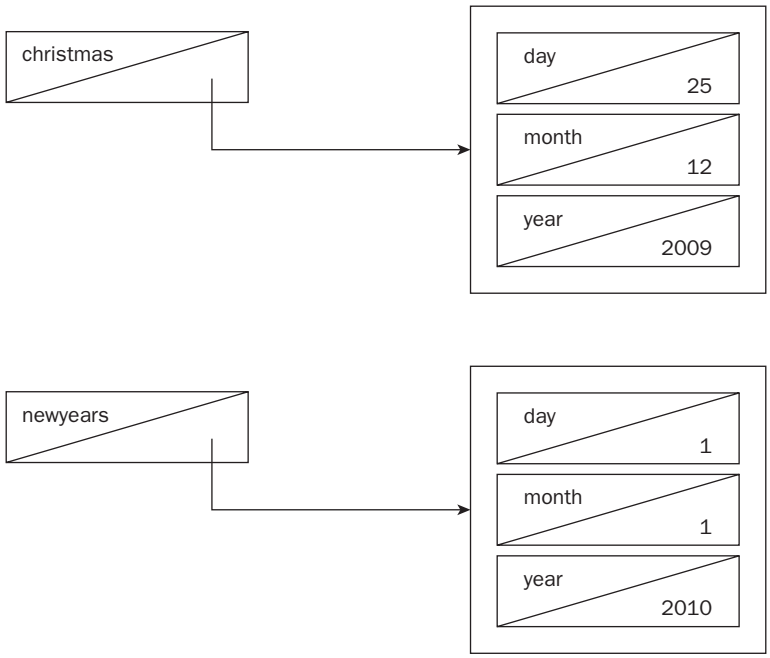
Consider the following program that instantiates two `GregorianCalendar` objects and assigns them to various references. Study the code and see if you can determine when each of the two objects either goes out of scope or all references to it are lost.

```
1.  import java.util.GregorianCalendar;
2.
3.  public class GCDemo {
4.    public static void main(String [] args) {
5.        GregorianCalendar christmas, newyears;
```

```
6.         christmas = new GregorianCalendar(2009,12,25);
7.         newyears = new GregorianCalendar(2010,1,1);
8.
9.         christmas = newyears;
10.         GregorianCalendar d = christmas;
11.         christmas = null;
12.     }
13. }
```

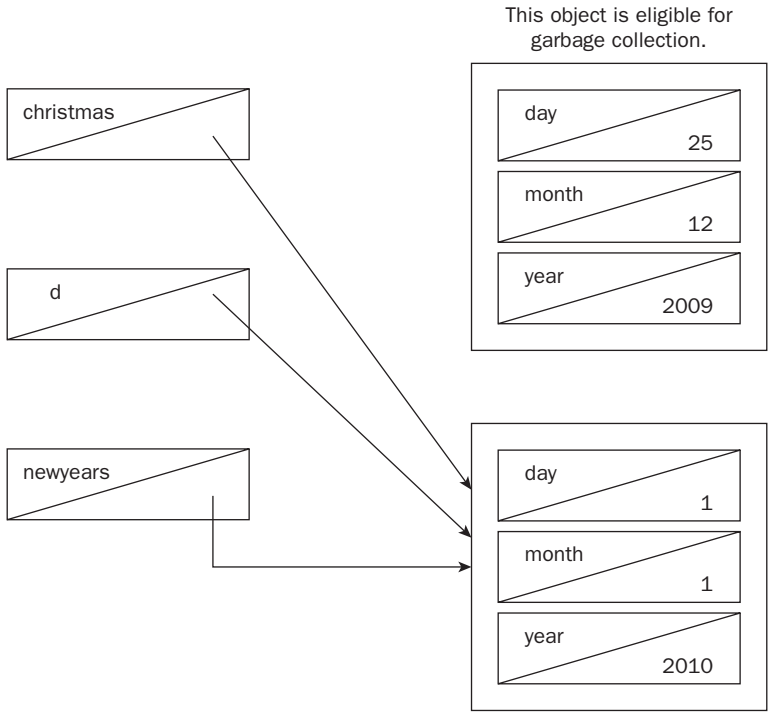
The two `GregorianCalendar` objects are created on lines 6 and 7, resulting in the references and objects that Figure 1.8 shows.

**FIGURE 1.8** Each `GregorianCalendar` object has a reference pointing to it.




On line 9, the `christmas` reference is assigned to `newyears`, which results in no more references pointing to the object from line 6, so this object immediately becomes available for garbage collection after line 9. There is now only one `GregorianCalendar` object (from line 7) reachable in memory, and after line 10 there are three references pointing to it, as Figure 1.9 shows.

**FIGURE 1.9** One `GregorianCalendar` object has no references to it and the other now has three.




Setting `christmas` to `null` on line 11 does not cause the object from line 7 to become eligible for garbage collection because there are still two references pointing to it: `d` and `newyears`. However, after line 12 the `main` method ends and both `d` and `newyears` go out of scope. Therefore, the object instantiated on line 7 becomes eligible for garbage collection after line 12.

### Know When an Object Is Eligible for Garbage Collection

The `GCDemo` program is typical of a question that you will encounter on the certification exam. Make sure you understand exactly when each of the two `GregorianCalendar` objects becomes eligible for garbage collection.

What does it mean to become eligible for garbage collection? Why not simply have the garbage collector immediately free the memory instead? The answer is that there is no guarantee in Java as to exactly when an object is actually garbage collected. The JVM

specification does not define how a garbage collector accomplishes the task of freeing memory. The specification only states that when an object is eligible for garbage collection, the garbage collector must eventually free the memory.

As a Java coder, you cannot specifically free memory on the heap. You can only ensure that your objects that you no longer want in memory are no longer reachable. In other words, make sure you don't have any references to the object that are still in scope.

The following section discusses the `System.gc` method, which provides a small amount of control over freeing memory on the heap.

## The *System.gc* Method

The `java.lang.System` class has a static method called `gc` that attempts to run the garbage collector. `System.gc` is the only method in the Java API that communicates with the garbage collector. Here is what the Java SE API documentation says about the `System.gc` method:

> Calling the `gc` method suggests that the Java Virtual Machine expend effort toward recycling unused objects in order to make the memory they currently occupy available for quick reuse. When control returns from the method call, the Java Virtual Machine has made a best effort to reclaim space from all discarded objects.

In other words, the `gc` method does not guarantee anything! The method might be useful if you are familiar with the intricate details of your JVM and how it implements this method. But the end result is that as a Java programmer you cannot free memory specifically in your code. You can only ensure that your objects are eligible for garbage collection, and then assume the garbage collector will do its job!

Let's look at another example typical of a question found on the exam. Examine the following code and determine when the `String` objects become eligible for garbage collection and when they actually get garbage collected:

```
1.  public class GCDemo2 {
2.    public static void main(String [] args) {
3.        String one = "Hello";
4.        String two = one;
5.        String three = "Goodbye";
6.
7.        three = null;
8.        System.gc();
9.        one = null;
10.       System.gc();
11.       two = null;
12.   }
13. }
```

The `"Goodbye"` object is created on line 5 and assigned to the reference `three`. Then `three` is set to `null` and the `gc` method is invoked. After line 7 the `"Goodbye"` object is definitely eligible for garbage collection, but if the exam question asks you when the object is garbage collected, the answer can only be "unknown." The call to `gc` on line 8 might have caused `"Goodbye"` to get garbage collected, but that is not guaranteed at all.

Line 9 does not cause `"Hello"` to become eligible because the reference `two` points to `"Hello"` also. Only after line 11 does `"Hello"` become eligible for garbage collection, and as already discussed we cannot know when the objects are actually garbage collected.

## The *finalize* Method

According to the exam objectives, you need to be able to "recognize the behaviors of the `Object.finalize()` method." The garbage collector invokes the `finalize` method of an object right before the object is actually garbage collected. The `finalize` method is declared in `Object`, and any subclass can override `finalize` to perform any necessary cleanup or dispose of system resources. The `finalize` method is only invoked on an object once by the garbage collector.

There won't be any trick questions about `finalize`. Just remember it gets invoked once and only when the object is in the process of being removed from memory. Be sure not to do anything in the `finalize` method that might somehow cause the object's reference to come back into scope. It is also a good idea to call `super.finalize` because you are overriding the behavior of `finalize` in the parent classes.

### Calling `super.finalize`

If you do call `super.finalize`, which is recommended, you need to declare the `java.lang.Throwable` exception thrown by the parent class's `finalize` method:

```
public class A extends Object {
    public void finalize() throws Throwable {
        System.out.println("Finalizing A");
    }
}
```

Let's look at an example. It is difficult to simulate garbage collection because you have little control over the garbage collector, but I came up with an example that demonstrates when the `finalize` method is called and also provides an extra level of complexity in

determining when an object is eligible for garbage collection. Consider the following class named `Dog` that contains a `finalize` method that prints out a simple message:

```
1.  public class Dog {
2.    private String name;
3.    private int age;
4.
5.    public Dog(String name, int age) {
6.        this.name = name;
7.        this.age = age;
8.    }
9.
10.    public void finalize() {
11.        System.out.println(name + " is being garbage collected");
12.    }
13. }
```

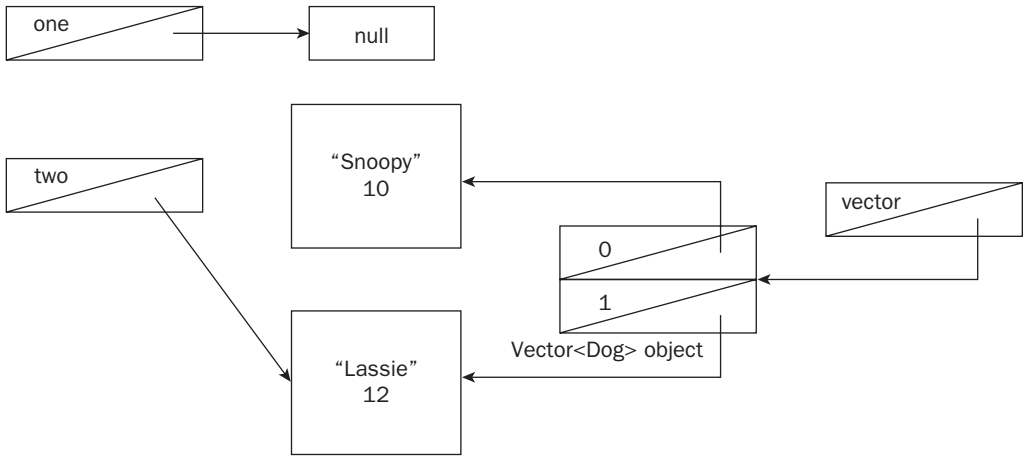
The following program instantiates two `Dog` objects and stores them in a `java.util.Vector`. Examine this program and see if you can determine when the two `Dog` objects become eligible for garbage collection:

```
1.  import java.util.Vector;
2.  public class GCDemo3 {
3.    public static void main(String [] args) {
4.        Vector<Dog> vector = new Vector<Dog>();
5.        Dog one = new Dog("Snoopy", 10);
6.        Dog two = new Dog("Lassie", 12);
7.
8.        vector.add(one);
9.        vector.add(two);
10.
11.       one = null;
12.       System.out.println("Calling gc once...");
13.       System.gc();
14.
15.       vector = null;
16.       System.out.println("Calling gc twice...");
17.       System.gc();
18.
19.       two = null;
20.       System.out.println("Calling gc again...");
```

```
21.        System.gc();
22.        System.out.println("End of main...");
23.
24.    }
25. }
```

The calls to `gc` are an attempt to force garbage collection so we can see when `finalize` is invoked on the `Dog` objects. The first step is determining when the objects are eligible for garbage collection. Adding the two `Dog` objects to the `Vector` creates additional references to the objects. On line 11 the reference `one` is set to `null`, but `Snoopy` is not eligible yet for garbage collection because of line 8. The `Vector` still has a reference to the Snoopy object, as shown in Figure 1.10.

**FIGURE 1.10** The `Vector` still has a reference to the Snoopy object.




However, when you set `vector` to `null` on line 15, it causes the `Snoopy` object to immediately become eligible for garbage collection. The `Lassie` object still has the reference `two` pointing to it, so it does not become eligible until after line 19. Here is a sample output of the `GCDemo3` program. (I use the term "sample output" because the output can change each time the program is executed depending on when the garbage collector actually invokes the `finalize` method.)

```
Calling gc once...
Calling gc twice...
Snoopy is being garbage collected
Calling gc again...
Lassie is being garbage collected
End of main...
```

No objects are freed after the first call to `gc` because no objects are eligible at that time. After the second call to `gc`, the `Snoopy` object is eligible, but the call to `finalize` happens in the thread of the garbage collector, so the output of `Snoopy`'s `finalize` method may or may not appear before the third call to `gc`. The exact output of running the `GCDemo3` program is indeterminate. The previous output is just one possible result.

**The `finalize` Method Is Only Invoked Once**

Expect at least one question on the exam about the `finalize` method. Keep in mind that it can only be called once on an object, and it only gets called by the garbage collector after an object is eligible for garbage collection but before the object is actually garbage collected.

This ends our discussion on garbage collection, an important topic not just for the SCJP exam but in our everyday programming of Java. Now we discuss another important topic in Java: the concept of call by value.

# Call by Value

The exam objectives state that you need to know "the effect upon object references and primitive values when they are passed into methods that perform assignments or other modifying operations on the parameters." A variable that is passed into a method is called an *argument*. Java simplifies the concept of passing arguments into methods by providing only one way to pass arguments: by value. Passing arguments *by value* means that a copy of the argument is passed to the method. Method return values are also returned by value, meaning a copy of the variable is returned. The SCJP exam requires an understanding of what call by value means, and we will discuss the details now.

An argument is passed into a corresponding method parameter. A *parameter* is the name of the variable in the method signature that gets assigned the value of the argument. Let's look at an example. Suppose we have the following method definition:

```
21. public long cubic(int y) {
22.     long longValue = (long) y;
23.     y = -1;
24.     return longValue * longValue * longValue;
25. }
```

To invoke this method, you must pass in an `int` argument. For example, the following code invokes the `cubic` method:

```
31. int x = 11;
32. long result = cubic(x);
```

The value of `x` is copied into the parameter `y`. We now have two `int`s in memory that have the value 11: `x` and `y`. Changing `y` to –1 in `cubic` has no effect on `x`. In fact, it is impossible to change `x` within the `cubic` function.

### Passing Primitives vs. Passing References

Sun seems to enjoy questions on the exam regarding call by value and methods that attempt to change the value of the argument. If the argument passed into a method parameter is a primitive type, it is impossible in Java for the method to alter the value of the original primitive.

If the argument passed into a method parameter is a reference type, the same rule applies: it is impossible for a method to alter the original *reference*. However, because the method now has a reference to the same object that the argument points to, the method *can* change the object. This is an important difference to understand. Study the upcoming `StackDemo` program for an example of this situation.

The following example of call by value uses references. Suppose we have the following method signature:

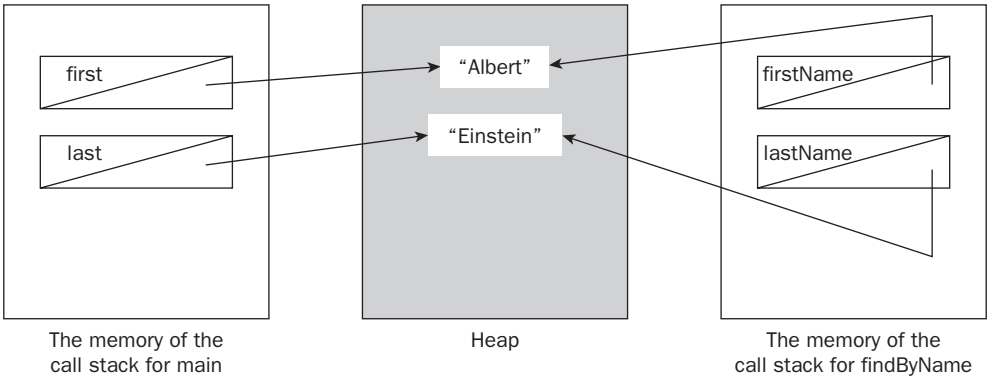
```
5. public int findByName(String lastName, String firstName) {
6.     lastName = "Doe";
7.     firstName = "Jane";
8.     return -1;
9. }
```

This method has two parameters, `lastName` and `firstName`. To invoke this method, two `String` objects must be passed in as arguments. For example, the following code invokes the `findByName` method. What is the output of this code?

```
14. String first = "Albert";
15. String last = "Einstein";
16. int result = findByName(last, first);
17. System.out.println(first + " " + last);
```

The argument `last` is copied into the parameter `lastName`. The argument `first` is copied into the parameter `firstName`. What gets copied? Well, because `last` and `first` are references, they contain memory addresses, and that is what gets copied. The result is that `lastName` points to the same `String` object as `last`, which in this example is `"Einstein"`. Similarly, `firstName` points to `"Albert"`, as shown in Figure 1.11. The objects did not get copied! There is still only one `String` object with the value `"Einstein"` in memory and only one `String` object with the value `"Albert"` in memory.

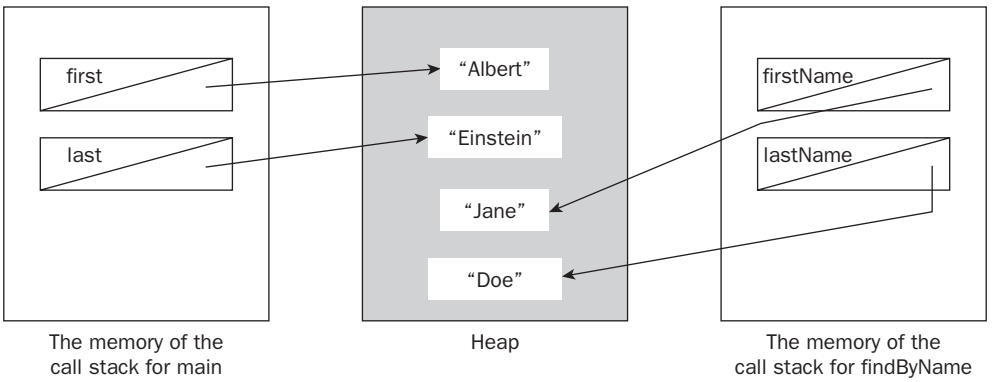
**FIGURE 1.11** The arguments from `main` are copied into the parameters of `findByName`.




Because `String` objects are immutable, the parameters `lastName` and `firstName` cannot change the objects `"Albert"` or `"Einstein"`. Setting the parameters equals to `"Jane"` and `"Doe"` has no effect on `first` and `last`, as Figure 1.12 shows. Therefore, the output of that code is

```
Albert Einstein
```

**FIGURE 1.12** `String` objects are immutable, so `findByName` cannot change `first` and `last`.



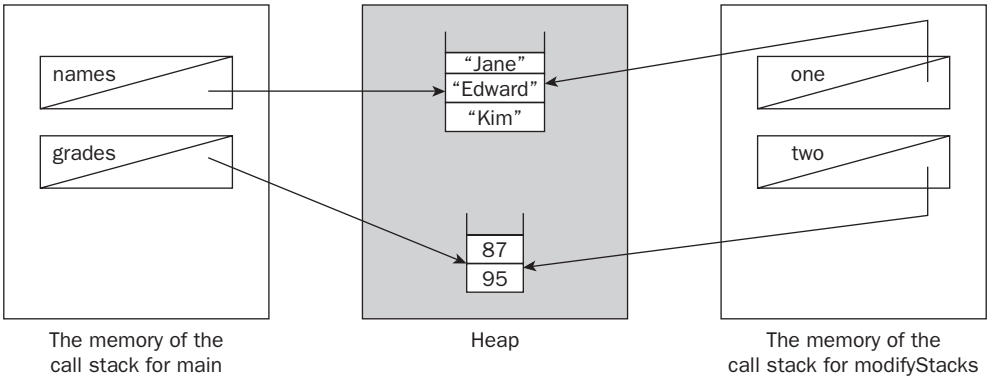
The only reason `firstName` and `lastName` could not change the objects is because the example uses `String` types and `String` objects are immutable. Let's look at an example where the arguments passed in refer to objects that *can* be altered by the method. Examine the following program and try to determine its output. If you are not familiar with the `java.util.Stack` class, the `push` method adds an element to the top of the stack and the `pop` method removes the top element from the stack.

```
1.  import java.util.Stack;
2.
3.  public class StackDemo {
4.
5.    public static void modifyStacks(Stack<String> one,
6.                            Stack<Integer> two) {
7.        two.push(50);
8.        one.pop();
9.        one = new Stack<String>();
10.    }
11.
12.    public static void main(String [] args) {
13.        Stack<String> names = new Stack<String>();
14.        names.push("Kim");
15.        names.push("Edward");
16.        names.push("Jane");
17.
18.        Stack<Integer> grades = new Stack<Integer>();
19.        grades.push(95);
20.        grades.push(87);
21.
22.        modifyStacks(names, grades);
23.
24.        for(String name : names) {
25.            System.out.println(name);
26.        }
27.
28.        for(int grade : grades) {
29.            System.out.println(grade);
30.        }
31.    }
32. }
```

Within `main`, two `Stack` objects are instantiated. The reference `names` refers to a `Stack` that contains `String` objects, and the reference `grades` refers to a `Stack` containing `Integer`

objects. Three strings are pushed onto the `names` stack, and two `int`s are pushed onto `grades`. Then `names` and `grades` are passed into `modifyStacks`. The parameter `one` points to the stack of `String`s and `two` points to the stack of `grades`, as Figure 1.13 shows.

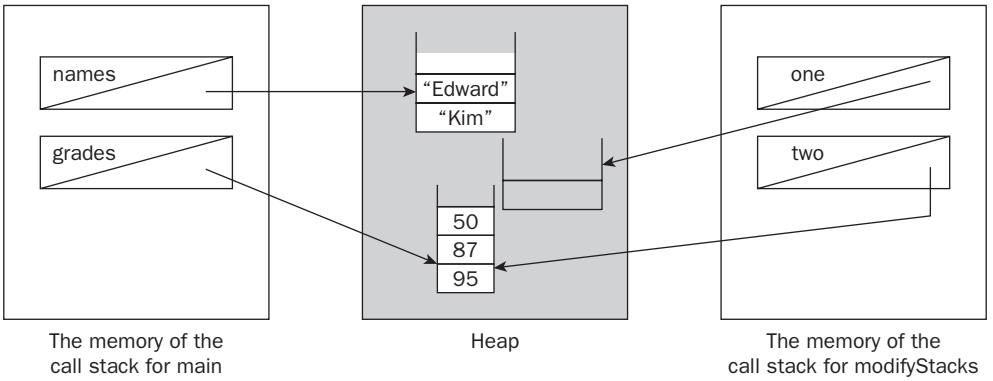
**FIGURE 1.13** The references from `main` are copied into the parameters of `modifyStacks`.




Pushing 50 onto `two` is the same as pushing it onto `grades` because the two references point to the same stack. Similarly, popping a value off `one` removes `"Jane"` from the `names` stack.

Note that setting `one` equal to a new `Stack` does not affect the `Stack` that `names` points to. We cannot modify the reference `names` within `modifyStacks`. Figure 1.14 shows the references and objects just before the `modifyStacks` method returns.

**FIGURE 1.14** Assigning the one reference to a new `Stack` does not affect the `names` stack.



When `modifyStacks` returns, `names` still points to the `Stack` containing `"Kim"` and `"Edward"`, and `grades` now points to a `Stack` containing 95, 87, and 50. The output of `StackDemo` is

```
Kim
Edward
95
87
50
```

Changing the reference `one` does not change the reference `names`. Although it is impossible for the `modifyStacks` method to alter the `names` reference, it was quite easy for the method to modify the object that `names` points to.

The concept of call by value also applies to values returned by a method, as we will see in the next section. As discussed earlier in this chapter, you need to be able to view code and determine when an object becomes eligible for garbage collection. When does the object on line 9 of `StackDemo` become eligible for garbage collection? Because the variable `one` is a parameter, it goes out of scope when the `modifyStacks` method returns on line 10. Because `one` is the only reference pointing to the `Stack` object from line 9, the object is eligible for garbage collection after line 10.

### Passing References vs. Passing Objects

You need to be able to distinguish the difference between a reference and an object. When passing arguments to a method, it is the reference that gets passed, not the object. It is impossible to pass an object to a method. In fact, the largest amount of data that can be copied into a parameter (or returned from a method) is a `long` or a `double`, both of which are 64 bits.

Return values are also passed by value, meaning a copy of the data is sent to the calling method. A method can return `void`, one of the eight primitive types, or a reference: there are no other possibilities. (Of course, the reference can be of any class or interface type, so the possible values you can return are actually endless, as long as you realize that a reference is getting returned, never an actual object!)

Let's look at an example using primitive types. Suppose we have the following method definition:

```
31. public int max(int a, int b) {
32.     int response;
33.     if( a < b) {
34.         response = b;
35.     } else {
```

```
36.         response = a;
37.     }
38.     return response;
39. }
```

The `max` method returns a local variable named `response`. A copy of `response` is returned to the calling method. Consider the following invocation of `max`:

```
45. public void go() {
46.     int x = 20, y = 30;
47.     int biggest = max(20, 30);
48.     System.out.println(biggest);
49. }
```

In this case, the parameter `a` is 20 and `b` is 30, resulting in a `response` of 30. A copy of 30 is passed back to the `go` method and stored in `biggest`. Because `max` is done executing, its call-stack memory is freed and `a`, `b`, and `response` all get destroyed. It doesn't make sense to try to modify `response` in the `go` method because `response` no longer exists in memory.

### The Call Stack

Every method that gets invoked in a Java thread is pushed onto the thread's method *call stack*. The method at the top of the call stack is the currently executing method. Each method on the call stack gets its own small amount of memory. When a method finishes executing (by running to completion, returning a value, or throwing an exception), the method gets popped off the call stack and its memory is freed. Any parameters and local variables are destroyed and no longer exist in the program's memory.

The next example shows a method that returns a reference to an object. Examine the code and see if you can determine when the `File` object instantiated on line 6 is eligible for garbage collection:

```
1.  import java.io.File;
2.
3.  public class ReturnDemo {
4.
5.    public File getFile(String fileName) {
6.        File f = new File(fileName);
7.        return f;
```

```
8.    }
9.
10.    public static void main(String [] args) {
11.        ReturnDemo demo = new ReturnDemo();
12.        File file = demo.getFile(args[0]);
13.
14.        if(file.exists()) {
15.            System.out.println(file.getName() + " file exists");
16.        } else {
17.            System.out.println(file.getName() + " doesn't exist");
18.        }
19.
20.        file = null;
21.    }
22. }
```

The `getFile` method returns the reference `f`, which points to a new `File` object. Keep in mind that this `File` object is on the heap, not in the method's call stack memory, so the `File` object is not destroyed when `getFile` returns. The local variable `file` in `main` gets a copy of `f` when `getFile` returns. The `File` object from line 6 does not become eligible for garbage collection until after line 20.

The `ReturnDemo` program demonstrates a method that instantiates an object and returns a reference to that object. This is a common occurrence in Java. Just remember that the object is on the heap (all objects are instantiated on the heap!) and a *copy* of the reference is returned to the calling method. As with method arguments, the largest piece of data that can be returned from any Java method is 64 bits (a `long` or `double`). The fact that Java only allows call by value is an attempt to simplify the language. There is never any confusion with arguments and parameters: the parameter is always a copy of the argument.

Now that we have discussed the details of call by value, we turn our attention to another objective in the "Fundamentals" section: the Java operators.

# Java Operators

You need to be able to "write code that correctly applies the appropriate operators." This section discusses the Java operators that appear on the exam. Table 1.2 lists all of the 41 operators in Java 6.0, listed in their *order of precedence*. Order of operations in Java is well defined, and the operators are guaranteed to be evaluated in the order shown. If operators have the same level of precedence, Java guarantees evaluation in left-to-right order.

**TABLE 1.2** The Java Operators

| Operator | Symbol and Precedence |
|---|---|
| Post-increment/post-decrement | *expression*++, *expression*-- |
| Pre-increment/pre-decrement | ++*expression*, --*expression* |
| Unary operators | +, -, ~, ! |
| Multiplication/division/modulus | *, /, % |
| Addition/subtraction | +, - |
| Shift operators | <<, >>, >>> |
| Relational operators | <, >, <=, >=, instanceof |
| Equal to/not equal to | ==, != |
| Bitwise AND, exclusive OR, inclusive OR | &, ^, \| |
| Logical AND, OR | &&, \|\| |
| Ternary operator | ? : |
| Assignment operators | = += -= *= /= %= &= ^= \|= <<= >>= >>>= |

The SCJP exam objectives specifically mention the following operators:

- Assignment operators: =, += and -=
- Arithmetic operators: +, -, *, /, %, ++, and --
- Relational operators: <, <=, >, >=, ==, and !=
- The `instanceof` operator
- Bitwise and logical operators: &, |, ^, !, &&, and ||
- The conditional operator (?:)

The upcoming sections discuss each of these categories of operators and the details that you need to know about the operators for the SCJP exam.

## The Assignment Operators

Java has 12 *assignment operators*: the *simple assignment* = and 11 *compound assignment operators*: +=, -=, *=, and so on. An assignment stores the result of the right-hand side of

the expression into the variable on the left-hand side. Here is an example using a simple assignment:

```
4. byte b = 120;
5. int x = b;
```

The `byte b` is assigned the literal value 120, and the `int x` is assigned the value of `b`, which is also 120. An assignment will not compile if the right-hand operand cannot be converted to the data type of the left-hand variable. For example, the following line of code does not compile:

```
7. int y = 12.5;    //does not compile
```

The literal 12.5 is a `double`, and a `double` cannot implicitly be converted to an `int` without loss of data. For this code to compile, you would need to cast the right-side to an `int`:

```
8. int y = (int) 12.5;    //compiles fine
```

The value of `y` is 12 after this line of code executes. The decimal value is simply truncated.

The compound assignment operators perform the given operator first between the left and right sides of the operand, and then the result is stored in the left-hand variable. What is the value of `z` after this line of code?

```
10. int x = 5;
11. int z = 10;
12. z *= x;
```

The compound assignment operator is multiplication, so `z` is multiplied by `x`, which evaluates to 50, and then `z` is assigned 50. The same result could have been evaluated using a simple assignment:

```
13. z = z * x;
```

However, sometimes the compound operator can save us from needing to cast a value before the assignment. For example, the following statements generate a compiler error. Do you see why?

```
15. long m = 1000;
16. int n = 5;
17. n = n * m; //compiler error here
```

The expression `n * m` is an `int` times a `long`. Before the multiplication can be evaluated, the `int` is promoted to a `long`. The result is therefore a `long`, so we need a cast to make the compiler happy:

```
18. n = (int) (n * m);
```

The result is n equal to 5000. However, using the compound operator avoids the cast. The following statements compile successfully and assign n to 5000:

```
19. long m = 1000;
20. int n = 5;
21. n *= m;
```

In this case, the value of m is implicitly cast to an int before the multiplication occurs. An int times an int results in an int, so no cast is needed.

**The Assignment Operators**

According to the SCJP exam objectives, knowledge of the assignment operators is limited to =, += and –=. Of course, if you understand how += and –= work, you understand how the other compound assignment operators work!

# The Arithmetic Operators

The exam objectives specifically mention having working knowledge of the following *arithmetic operators*:

- + – : addition and subtraction
- * / : multiplication and division
- % : modulus
- ++ –– : increment and decrement

We will now discuss each of these operators in detail.

## The Additive Operators

The operators + and – are referred to as *additive operators*. They can be evaluated on any of the primitive types except boolean. Additionally, the + operator can be applied to String objects, which results in string concatenation.

If the operands are of different types, the smaller operand is promoted to the larger. At a minimum, the operands are promoted to ints. For example, the following innocent-looking code does not compile. Can you see why?

```
short s1 = 10, s2 = 12;
short sum = s1 + s2;    //does not compile!
```

Because a short is smaller than an int, both s1 and s2 are promoted to ints before the addition. The result of s1 + s2 is an int, so you can only store the result in a short if you

use the cast operator. The compiler complains about a possible loss of precision, but casting fixes the problem:

```
short s1 = 10, s2 = 12;
short sum = (short) (s1 + s2);
```

The value of `sum` is 22 after this code executes.

### A Note about Casting

I want to take a moment to point out something subtle but important about the cast operator. The sole purpose of casting primitive types is to make the compiler happy. When you assign a larger data type to a smaller one, the compiler complains about a possible loss of precision.

However, if you are aware and comfortable with the possible loss of precision at runtime, then you simply cast the result, which tells the compiler you know what you are doing. At runtime, the data may very well be invalid. For example, the following code compiles and runs, but you might be surprised by the output:

```
byte b1 = 60, b2 = 60;
byte product = (byte) (b1 * b2);
System.out.println(product);
```

This code outputs the number 16, clearly not the result of 60 times 60. The mistake lies in the limitations of a `byte`, which can only store values up to 127. Because 60 * 60 = 3600, the value of 16 is the lower 8 bits of the binary representation of 3600. The significant bits were lost in the runtime assignment of 3600 to the byte product.

We will revisit this discussion of casting again when we talk about inheritance and casting references in Chapter 6, "OO Concepts," because casting reference types is a different story altogether!

The JVM ensures order of operations is evaluated left-to-right when operators share the same precedence. For example, what is the value of `x` after this line of code executes?

```
String x = 12 - 6 + "Hello" + 7 + 5;
```

Following the order from left to right, 12 – 6 is evaluated first and results in 6. The next + operator is not addition but string concatenation, so the 6 is promoted to a `String` and the result is `"6Hello"`. Following left to right, the next + is also string concatenation, resulting in `"6Hello7"`, and finally the value of `x` after the last string concatenation is `"6Hello75"`.

## The Multiplicative Operators

The operators *, /, and % are referred to as the *multiplicative operators*. They have a higher precedence of operation than additive operators. The multiplicative operators can only be performed on the numeric primitive types; otherwise, a compiler error occurs.

As with + and –, the multiplicative operators promote both operands to the data type of the larger operand. If both operands are smaller than an `int`, both operands are converted to `int`s before the multiplication occurs. For example, what is the result of the following statements?

```
4. int a = 26, b = 5;
5. double d = a / b;
```

The expression `a / b` is integer division, which results in the `int` 5. Therefore, the value of `d` is 5.0. The fact that we store the result in a `double` does not affect the arithmetic because the assignment takes place after the arithmetic is already performed.

If one of the operands is a `float` or `double`, the expression is evaluated using floating-point arithmetic and the result will be a `float` or `double` depending on the operand types. For example, what is the result of the following statements?

```
8. int a = 26;
9. float f = a / 5.0F;
```

Because 5.0 is a `float` (by virtue of the `"F"` appended to it), the `int a` is promoted to a `float` and floating-point division is performed. The value of `f` is 5.2 after this code executes.

## The MODULUS Operator

The modulus operator, also known as the remainder operator, evaluates the remainder of two numbers when they are divided. For example, what is the result of the following expression?

```
int x = 12 % 5;
```

The remainder of 12 divided by 5 is 2, so `x` is 2.

If the first operand is negative, so is the result of the modulus. The value of `y` after the following statement is –1:

```
int y = -17 % 4;
```

In Java you can evaluate the remainder of floating-point numbers as well. While not as intuitive as integer modulus, there is still a remainder in floating-point division. For example, what is the output of the following code?

```
System.out.println(12.4 % 3.2);
```

The answer is 2.8. A calculator won't help you on this one. You need to perform the division longhand to see where the remainder of 2.8 comes from.

The multiplication operators are evaluated left-to-right if the expression does not contain parentheses. What is the value of `result` after this statement?

```
int result = 12 + 2 * 5 % 3 - 15 / 4;
```

The expression evaluates to an `int` because all the literal values are `int`s. Here is how the expression is evaluated one level of precedence at a time. The parentheses are added for clarification.

```
12 + (2 * 5) % 3 - (15 / 4)
    12 + (10 % 3) - 3
      (12 + 1) - 3
         13 - 3
           10
```

Therefore the value of `result` is 10 after the statement executes.

## The Increment and Decrement Operators

The operators ++ and – – are referred to as the increment and decrement operators because they increment and decrement (respectively) a numeric type by 1. The operators can be applied to an expression either prefix or postfix. These operators have the highest level of precedence of all the Java operators. They can only be applied to numeric operands, and the result is the same data type as the operand.

For example, the following statements create an `int` and increment it using ++. What is the output of this code?

```
3. int x = 6;
4. System.out.println(x++);
5. System.out.println(x);
```

Adding or subtracting 1 seems simple enough, but these operators can be confusing because of *when* they are evaluated! The output of the previous statements is

```
6
7
```

When the operator appears after the operand, the increment or decrement does not occur until after the operand is used in the current expression. On line 3, `x` is printed out as 6, then incremented to 7, which is demonstrated by the output of line 5.

When the increment operator appears before the operand, the operand is incremented first, and then the result is used in the current expression. The same is true for the decrement operator.

Examine the following code and try to determine its output:

```
10. char c = 'A';
11. for(int i = 1; i <= 10; i++) {
12.    System.out.print(c++ + " ");
13. }
14. System.out.print(c);
```

The first value printed is `'A'`, then `c` is incremented, which results in `'B'` printed on the second iteration of the loop. In total, 11 `char`s are printed and the output is

```
A B C D E F G H I J K
```

The following code demonstrates use of the decrement operator. Examine the code and try to determine its output:

```
16. int y = 5;
17. int result = y-- * 3 / --y;
18. System.out.println("y = " + y);
19. System.out.println("result = " + result);
```

I have to admit this is a tricky question! (I hope you never see code like this in the real world.) Notice `y` is decremented twice, so the output of `y` is 3. The value of `result` is not as obvious. Order of operations dictates that the multiplication is evaluated first. The value of `y` is 5, so `5 * 3` is 15. The multiplication is done, so the post-decrement occurs and `y` becomes 4. Now the division is evaluated and `y` is pre-decremented to 3 before the division, resulting in `15 / 3`, which is 5. The output of this code is

```
y = 3
result = 5
```

**Make Sure You Understand the Increment and Decrement Operators**

The exam has plenty of questions that use the prefix and postfix increment and decrement operators. In many situations, the exam question is testing a different Java concept, not the incrementing or decrementing of variables. Make sure you have a good understanding of these fundamental (and sometimes tricky) operators.

## The Relational Operators

The following operators are referred to as the *relational operators*:

- `<` : less than
- `<=` : less than or equal

- `>` : greater than
- `>=` : greater than or equal

The relational operators can only be performed on numeric primitive types, and the result of each relational operator is always a `boolean`. If the operands are not the same primitive type, the smaller operand is promoted to the larger operand's type before the comparison is made.

To demonstrate the relational operators, let's take a look at some examples. What is the result of the following statements?

```
5. int x = 10, y = 20, z = 10;
6. System.out.println(x < y);
7. System.out.println(x <= y);
8. System.out.println(x > z);
9. System.out.println(x >= z);
```

Because `x` and `z` are the same value, `x > z` is `false`. The other statements evaluate to `true`. Therefore, the output of this code is

```
true
true
false
true
```

**The `boolean` Primitive Type**

The result of a relational operator is a `boolean`, which can only be the values `true` or `false`. The following line of code does not compile:

```
int result = x < y;
```

The `boolean` primitive type in Java is not compatible with the `int` type. In other languages like C and C++, numeric types are often used for Boolean expressions, where 0 is false and non-zero is true. In Java, a `boolean` can never be treated as a numeric type, nor can a numeric type ever be treated as a `true` or `false` value.

## The *instanceof* Operator

The `instanceof` operator compares a reference to a class or interface data type. The result is `true` if the reference is an instance of the data type; otherwise, the result is `false`. The syntax for the `instanceof` operator looks like this:

```
reference instanceof ClassOrInterfaceName
```

Let's take a look at an example. See if you can determine the output of the following statements:

```
3.  String s = "Hello, World";
4.  if(s instanceof String) {
5.      System.out.print("one");
6.  }
7.  if(s instanceof Object) {
8.      System.out.print("two");
9.  }
10. if(s instanceof java.io.Serializable) {
11.     System.out.print("three");
12. }
```

The reference `s` points to a `String` object, so line 4 is `true` and `"one"` is printed on line 5. Every object in Java is of type `Object`, so line 7 is `true` for any reference; therefore, `"two"` is printed. The `String` class implements the `Serializable` interface, which makes `String` objects `Serializable` objects as well. Therefore, line 10 is also `true` and the output of the previous code is

```
onetwothree
```

One of the main usages of the `instanceof` operator is when you cast a reference to a subclass type. If you cast a reference to an invalid data type, a `ClassCastException` is thrown by the JVM. For example, the following statements compile, but at runtime an exception is thrown:

```
Object x = new String("a String object");
Date d = (Date) x;
```

The output of this code is

```
Exception in thread "main" java.lang.ClassCastException:
 java.lang.String cannot be cast to java.util.Date
```

Using the `instanceof` operator, you can avoid this situation:

```
17. Object x = new String("a String object");
18. if(x instanceof Date) {
19.     Date d = (Date) x;
20. }
```

Because `x` points to a `String` object and not a `Date` object, line 18 is `false` and the invalid cast does not occur, avoiding the uncaught `ClassCastException`. We will see the `instanceof` operator again in Chapter 6.

## The Bitwise and Logical Operators

The following operators are referred to as the *bitwise and logical operators*:

- `&` : the AND operator
- `^` : the exclusive OR operator
- `|` : the inclusive OR operator
- `&&` : the conditional AND operator
- `||` : the conditional OR operator

The `&`, `^`, and `|` operate on expressions where both operands are either primitive numeric types or both are `boolean` expressions. When operating on numeric types, they are bitwise operators. When operating on `boolean` types, they are logical operators. The `&&` and `||` operators require both operands to be `boolean` expressions, so they are strictly logical operators.

The term *bitwise* refers to the `&`, `^`, and `|` operators performing a bitwise AND or OR of the two operands. Table 1.3 shows the result of the possible outcomes for each of these three operators.

**TABLE 1.3** The Bitwise Operators

| & (AND) | ^ (exclusive OR) | \| (inclusive OR) |
|---|---|---|
| 0 & 0 is 0 | 0 ^ 0 is 0 | 0 \| 0 is 0 |
| 0 & 1 is 0 | 0 ^ 1 is 1 | 0 \| 1 is 1 |
| 1 & 0 is 0 | 1 ^ 0 is 1 | 1 \| 0 is 1 |
| 1 & 1 is 1 | 1 ^ 1 is 0 | 1 \| 1 is 1 |

Notice the `&` operator results in 1 only when both operands are 1, while the `|` operator results in 0 only when both operators are 0. The exclusive OR `^` is 1 when the two operands are different; otherwise it is 0.

The bitwise operators are evaluated on integer types. To compute the result, you need to know the binary representation of the values. For example, what is the result of the following expression?

```
int result = 12 ^ 45;
```

Begin by converting the 12 and 45 to binary numbers and align them vertically. Then perform the exclusive OR on each column, as Figure 1.15 shows.

**FIGURE 1.15** Computing the exclusive or expression12^45

```
  12   =   0000 1100
  45   =   0010 1101
-------------------
12^45  =   0010 0001
```

The result is 00100001 in binary, which is 33 in decimal. Therefore, the value of `result` is 33.

The `&`, `^`, and `|` are also logical operators, meaning they can operate on `boolean` types. The result of each operator is identical to Table 1.2 if you were to replace each 0 with `false` and each 1 with `true`. For example, the AND operator `&` is only `true` when both operands are `true`. The inclusive OR operator `|` is only `false` when both operands are `false`. The exclusive OR is only `true` when the two operands are different.

What is the output of the following logical statements?

```
3. int a = 5, b = 10, c = 0;
4. boolean one = a < b & c != 0;
5. System.out.println(one);
6. boolean two = true | true & false;
7. System.out.println(two);
8. boolean three = (c != 0) & (a / c > 1);
9. System.out.println(three);
```

The variable `one` on line 4 is the result of `true & false`, which is `false`. The result of `two` on line 6 might surprise you. The `&` operator has a higher precedence than `|`, so the `true & false` is evaluated first, which results in `false`. Then `true | false` is evaluated, which is `true`, so `two` evaluates to `true`.

You might think that the Boolean on line 8 evaluates to `false`, but that line of code actually throws an `ArithmeticException` when attempting to compute `a / c`. The value of `c` is 0 and integer division by 0 is undefined in Java. Therefore, the last `println` never executes.

The example of `a / c` is a typical situation where a conditional operator comes in handy. The conditional operators `&&` and `||` short-circuit, meaning the right operand may not get evaluated if the left hand operand can determine the result.

For example, when using `&&`, if the left operand is `false`, there is no need to check the right operand. False AND anything is `false`. In this case, the right-hand expression is not evaluated. Similarly, when using `||`, if the left operand is `true`, there is no need to check the right operand because true OR anything is `true`.

The following statements are a modification of the previous example, except this time the logical expression short-circuits. What is the value of `three` after the following statements?

```
21. int a = 5, b = 10, c = 0;
22. boolean three = (c != 0) && (a / c > 1);
```

Because c is 0, the expression `c != 0` is `false` and evaluation stops. The variable `three` is `false` and this code does not throw an exception at runtime.

**Short-Circuit Behavior**

Watch for the short-circuit behavior on the exam. The exam question might alter a variable in the right operand. For example, what is the output of the following code?

```
int x = 6;
boolean answer = (x >= 6) || (++x <= 7);
System.out.println(x);
```

Because `x >= 6` is true, the incrementing of x does not occur in the right operand, so the output of this code is 6.

## The Conditional Operator

Java contains a *conditional operator* `? :`, often referred to as the *ternary operator* because it is the only operator in Java that has three operands. The syntax for the conditional operator is

```
boolean_expression ? true_expression : false_expression
```

The first operand must be a `boolean` expression. If this `boolean` expression is `true`, then the second operand is chosen; otherwise, the third operand is chosen. The second and third operands can be any expressions that evaluate to a value, or any method calls that return a value.

The conditional operator is a condensed version of an if/else statement that can be handy in a lot of different situations, especially when outputting or displaying data. For example, what is the output of the following statements?

```
int x = 6;
System.out.println( x != 0 ? 10/x : 0);
```

Because x is not 0, the output is the result of `10 / 6`, which is 1.

Let's look at another example. What is the output of the following statements?

```
double d = 0.36;
System.out.println( d > 0 && d < 1 ? d *= 100 : "not a percent");
```

Because d is between 0 and 1, the output is 36.0. There is no requirement that the second and third operands be the same data types (or even compatible types).

## The Equality Operators

The == (equal to) and != (not equal to) operators are referred to as the equality operators. The equality operators can be used in the following three situations, all of which return a `boolean`:

- The two operands are numerical primitive types.
- The two operands are `boolean` types.
- The two operands are references types or null types.

This implies that you cannot compare a byte to a `boolean`, or an `int` to a reference type. The two operands must be compatible. If one operand is a larger type, then the smaller type is promoted before the comparison. For example, you can compare an `int` to a `float`; the `int` is promoted to a `float` and a floating-point comparison is made. You can compare a `char` to an `int`: the `char` is promoted to an `int` and integer equality is performed.

Let's look at some uses of the equality operators. Examine the following code and try to determine its output:

```
6.  int x = 57;
7.  float f = 57.0F;
8.  double d = 5.70;
9.  boolean b = false;
10.
11. boolean one = x == 57;
12. System.out.println(one);
13. boolean two = (f != d);
14. System.out.println(two);
15. boolean three = (b = true);
16. System.out.println(three);
```

Lines 12 and 14 both print out `true`. The order of operations on line 11 ensures that `x` is compared to 57 before the assignment to `one`, even though parentheses would have made that statement easier to read (as in line 13). If you glanced over this code too quickly, you may think that line 16 prints out `false`, but the actual output is `true`. On line 15, `(b = true)` is an assignment, not a test for equality. Following the order of parentheses, `b` is set to `true` first, then `three = b` is evaluated, which sets `three` equal to `true`. The output of these statements is

```
true
true
true
```

The equality operators can also be evaluated on reference types. It is important to understand that evaluating == and != on two references compares the references, not the objects they point to. Two references are equal if and only if they point to the same object (or both point to `null`); otherwise, the two references are not equal.

The following ReferenceDemo program demonstrates comparing references. Examine the code and try to determine its output.

```
1.  import java.io.File;
2.  import java.util.Date;
3.
4.  public class ReferenceDemo {
5.    public static void main(String [] args) {
6.        File f1 = new File("mydata.txt");
7.        File f2 = new File("mydata.txt");
8.        if(f1 != f2) {
9.            System.out.println("f1 != f210.
11.       }
12.        Date today = new Date();
13.        Date now = today;
14.        if(today == now) {
15.            System.out.println("today == now");
16.        }
17.
18.        String s1 = "Hello";
19.        String s2 = "Hello";
20.        if(s1 == s2) {
21.            System.out.println("s1 == s2");
22.        }
23.
24.        String x1 = new String("Goodbye");
25.        String x2 = new String("Goodbye");
26.        if(x1 == x2) {
27.            System.out.println("x1 == x2");
28.        }
29.    }
30. }
```

Let's study this program. The references f1 and f2 point to two different File objects, so the two references cannot be equal. It is irrelevant that the two File objects look the same in memory; they are clearly two different objects so their references are not equal. On the other hand, there is only one Date object in memory and today and now both point to it, so today == now is true.

Comparing String references in Java tends to be confusing because of how the JVM treats string literals. Because String objects are immutable, the JVM can reuse string literals for efficiency and to save memory. Because "Hello" is a String literal known at compile time, the JVM only creates one "Hello" object in memory, and s1 and s2 both

refer to it. Therefore, `s1 == s2` evaluates to `true`. On the other hand, `x1` and `x2` are not literals but actual `String` objects created dynamically at runtime, making them distinct objects. Therefore, `x1` and `x2` point to different objects and cannot be equal. The output of the `ReferenceDemo` program is

```
f1 != f2
today == now
s1 == s2
```

The important point to take from this discussion is that evaluating `==` and `!=` on reference types only compares whether or not the two references point to the same object. If you want to compare the actual contents of two objects, the `equals` method is used, which we discuss next.

# Equality of Objects

The exam objectives address the ability to "determine the equality of two objects or two primitives." As we saw in the previous section, you use the `==` operator to determine if two primitives are equal. We also saw that two references are equal if and only if they point to the same object. But what does it mean for two *objects* to be equal? (Don't forget: references and objects are different entities!) As a Java programmer, you get to decide what it means for two objects to be equal. The `java.lang.Object` class contains an `equals` method with the following signature:

```
public boolean equals (Object obj)
```

The default implementation in `Object` tests for reference equality, which we can already perform with `==`. The general rule of thumb is to override `equals` in all your classes to define what it means for two objects of your class type to be equal. Equality should be based on the business logic of your application.

### The `equals` Method

Because the `equals` method is defined in `Object`, you can invoke `equals` on any object, passing in any other object. For example, the following statements are valid:

```
String s = "Hello";
java.util.Date d = new java.util.Date();
boolean b = s.equals(d);
```

The value of `b` is `false`. Logic would tell us that a `String` object and a `Date` object should never be equal, and that is the case. Typically two objects have to be of the same class type for them to be equal. However, that doesn't stop you from comparing two objects of different types, because the `equals` method can be invoked with any two objects.

Let's look at an example. Suppose we have the following class named `Dog`:

```
1.  public class Dog {
2.    private String name;
3.    private int age;
4.
5.    public Dog(String name, int age) {
6.        this.name = name;
7.        this.age = age;
8.    }
9.  }
```

What does it mean for two `Dog` objects to be equal? Suppose in our application two `Dog` objects are equal if they have the same name and age. Then `Dog` can override `equals` and implement this business logic:

```
1.  public class Dog {
2.     private String name;
3.    private int age;
4.
5.    public Dog(String name, int age) {
6.        this.name = name;
7.        this.age = age;
8.    }
9.
10.    public boolean equals(Object obj) {
11.        if(!(obj instanceof Dog))
12.            return false;
13.        Dog other = (Dog) obj;
14.        if(this.name.equals(other.name) &&
15.          (this.age == other.age)) {
16.            return true;
17.        } else {
18.            return false;
19.        }
20.    }
21. }
```

Within `equals`, we first test to see if the class type of the other object is `Dog`. If the other object is not a `Dog` object, we can quickly deduce the two objects are not equal. Otherwise, the incoming reference is cast to a `Dog` reference and the `name` and `age` are checked for equality. Because the `name` is a `String` object, we use the `equals` method of the `String` class to compare the two `name` objects.

The following DogTest program creates three Dog objects and test them for equality. Examine the code and try to determine its output:

```
1.  public class DogTest {
2.    public static void main(String [] args) {
3.        Dog one = new Dog("Fido", 3);
4.        Dog two = new Dog("Fido", 3);
5.        Dog three = new Dog("Lassie", 3);
6.
7.        if(one.equals(two)) {
8.            System.out.println("Fido");
9.        }
10.
11.        if(one.equals(three)) {
12.            System.out.println("Lassie");
13.        }
14.
15.        if(one == two) {
16.            System.out.println("one == two");
17.        }
18.    }
19. }
```

Because the Dog objects referred to by one and two have the same name and age, one.equals(two) is true and "Fido" is displayed. The "Lassie" object has a different name, so one.equals(three) is false. The test for one == two is false because one and two point to different (but equal) objects.

### The hashCode Method

The Object class contains a method named hashCode with the following signature:

```
public int hashCode()
```

This method is used by hash table data structures. The hashCode and equals methods are related in the sense that two objects that are equal should generate the same hash

code. Therefore, any time you override equals in a class, you should also override hashCode. In the Dog class, the following hashCode method maintains this required relationship of equals and hashCode:

```
public int hashCode() {
    return age;
}
```

If two Dog objects are equal in our example, they have the same age and therefore will have the same hash code.

# Summary

This chapter covered the "Fundamentals" objectives of the SCJP exam. Sun lists these topics last in their official list of objectives, but we needed to discuss these fundamentals first before tackling the more advanced topics of the exam.

The goal of this chapter was to discuss the details of running Java applications, including working with packages and using an appropriate classpath. You should also have a good understanding of garbage collection and when an object becomes eligible for garbage collection.

We also discussed the details of using the many operators in Java. As the title of the chapter suggests, these topics are the "fundamentals" of Java that provide the building blocks for the remainder of this book.

Be sure to test your knowledge of these fundamentals by answering the Review Questions that follow. I tried to write questions that reflect the style and difficulty level of questions on the SCJP exam, so attempt to answer the questions seriously without looking back at the pages of this chapter and do your best. Make sure you have a good understanding of the following Exam Essentials before attempting the Review Questions, and good luck!

# Exam Essentials

**Understand the effect of putting a class in a package.** In the real world, all classes are declared within a package. Know how to run a Java class from a command prompt when the class is in a package, and be sure to recognize what the CLASSPATH environment variable needs to be.

**Get comfortable with looking at code and determining its output.** Many of the exam questions provide either a small program or a snippet of code and ask what the output is. Practice reading code and determining what it does, including whether or not the given code compiles successfully.

**Understand call by value.** I can guarantee at least two or three questions on the exam that have an argument passed into a method and the method alters the parameter. Understand that a method cannot change the argument. The only effect a method can have on an argument is when the argument is a reference, in which case the method can alter the object that the reference points to.

**Be able to determine when an object becomes eligible for garbage collection.** Knowing when an object is eligible for garbage collection demonstrates an important understanding of Java and how it creates and destroys objects. You will see at least one question on the exam that asks you when an object is eligible for garbage collection, and also at least one question involving the `Object.finalize()` method.

**Understand the difference between == and the `equals` method.** Use the == comparison operator to determine if two primitive types are equal and also to determine if two references point to the same object. Use the `equals` method to determine if two objects are "equal," which is whatever equality means in the business logic of the class.

**Familiarize yourself with the Java operators.** The Java operators are a fundamental aspect of the language, and almost all of the exam questions that contain sample code use one or more of the Java operators.

# Review Questions

1. The following code appears in a file named `Plant.java`. What is the result of compiling this source file? (Select one answer.)

```
1.  public class Plant {
2.    public boolean flowering;
3.    public Leaf [] leaves;
4.  }
5.
6.  class Leaf {
7.    public String color;
8.    public int length;
9.  }
```

   **A.** The code compiles successfully and two bytecode files are generated: `Plant.class` and `Leaf.class`

   **B.** The code compiles successfully and one bytecode file is generated: `Plant.class`.

   **C.** A compiler error occurs on line 1.

   **D.** A compiler error occurs on line 3.

   **E.** A compiler error occurs on line 6.

2. Suppose a class named `com.mycompany.Main` is a Java application, and `Main.class` is in the following directory:

```
\projects\build\com\mycompany
```

   Which of the following commands successfully executes `Main`? (Select two answers.)

   **A.** `java -classpath=\projects\build com.mycompany.Main`

   **B.** `java -classpath \projects\build\com\mycompany Main`

   **C.** `java -classpath \projects\build com.mycompany.Main`

   **D.** `java -classpath \projects\build\com mycompany.Main`

   **E.** `java -cp \projects\build com.mycompany.Main`

3. A class named `Test` is in the `a.b.c` package, defined in a file named `Test.java` and saved in the following directory:

```
c:\abcproject\src\Test.java
```

   Assuming the code in `Test.java` uses only classes from `java.lang` and contains no compiler errors, what is the result of the following command line? (Select one answer).

```
c:\abcproject\src>javac -d c:\abcproject\deploy Test.java
```

**A.** A `NoClassDefFoundError` occurs.

**B.** A `ClassNotFoundException` occurs.

**C.** `Test.class` is generated in the `c:\abcproject\deploy` directory.

**D.** `Test.class` is generated in the `c:\abcproject\deploy\abc` directory.

**E.** `Test.class` is generated in the `c:\abcproject\deploy\a\b\c` directory.

**4.** What is the outcome of the following code?

```
1.  public class Employee {
2.    public int employeeId;
3.    public String firstName, lastName;
4.    public java.util.GregorianCalendar hireDate;
5.
6.    public int hashCode() {
7.        return employeeId;
8.    }
9.
10.    public boolean equals(Employee e) {
11.        return this.employeeId == e.employeeId;
12.    }
13.
14.    public static void main(String [] args) {
15.        Employee one = new Employee();
16.        one.employeeId = 101;
17.
18.        Employee two = new Employee();
19.        two.employeeId = 101;
20.
21.        if(one.equals(two)) {
22.            System.out.println("Success");
23.        } else {
24.            System.out.println("Failure");
25.        }
26.    }
27. }
```

**A.** `Success`

**B.** `Failure`

**C.** Line 6 causes a compiler error.

**D.** Line 10 causes a compiler error.

**E.** Line 10 causes a runtime exception to occur.

**5.** What is the result of compiling the following class?

```
1.  public class Book {
2.    private int ISBN;
3.    private String title, author;
4.    private int pageCount;
5.
6.    public int hashCode() {
7.        return ISBN;
8.    }
9.
10.    public boolean equals(Object obj) {
11.        if(!(obj instanceof Book)) {
12.            return false;
13.        }
14.        Book other = (Book) obj;
15.        return this.ISBN == other.ISBN;
16.    }
17. }
```

**A.** The class compiles successfully.

**B.** Line 6 causes a compiler error because `hashCode` does not return a unique value.

**C.** Line 10 causes a compiler error because the `equals` method does not override the parent method correctly.

**D.** Line 14 does not compile because the `ClassCastException` is not handled or declared.

**E.** Line 15 does not compile because `other.ISBN` is a `private` field.

**6.** What is the outcome of the following statements? (Select one answer.)

```
6.  String s1 = "Canada";
7.  String s2 = new String(s1);
8.  if(s1 == s2) {
9.    System.out.println("s1 == s2");
10. }
11. if(s1.equals(s2)) {
12.     System.out.println("s1.equals(s2)");
13. }
```

**A.** There is no output.

**B.** `s1 == s2`

**C.** `s1.equals(s2)`

**D.** Both B and C

**7.** Suppose we have the following class named GC:

```
1.  import java.util.Date;
2.
3.  public class GC {
4.    public static void main(String [] args) {
5.        Date one = new Date();
6.        Date two = new Date();
7.        Date three = one;
8.        one = null;
9.        Date four = one;
10.       three = null;
11.       two = null;
12.       two = new Date();
13.   }
14. }
```

Which of the following statements are true? (Select two answers.)

**A.** The Date object from line 5 is eligible for garbage collection immediately following line 8.

**B.** The Date object from line 5 is eligible for garbage collection immediately following line 10.

**C.** The Date object from line 5 is eligible for garbage collection immediately following line 13.

**D.** The Date object from line 6 is eligible for garbage collection immediately following line 11.

**E.** The Date object from line 6 is eligible for garbage collection immediately following line 13.

**8.** What is the output of the following code?

```
1.  private class Squares {
2.    public static long square(int x) {
3.        long y = x * (long) x;
4.        x = -1;
5.        return y;
6.    }
7.
8.    public static void main(String [] args) {
9.        int value = 9;
10.       long result = square(value);
11.       System.out.println(value);
12.   }
13. }
```

**A.** This code does not compile.

**B.** 9

**C.** -1

**D.** 81

**9.** What is the output of the following code?

```
1.  public class TestDrive {
2.
3.    public static void go(Car c) {
4.        c.velocity += 10;
5.    }
6.
7.    public static void main(String [] args) {
8.        Car porsche = new Car();
9.        go(porsche);
10.
11.       Car stolen = porsche;
12.       go(stolen);
13.
14.       System.out.println(porsche.velocity);
15.    }
16. }
17.
18. class Car {
19.    public int velocity = 10;
20. }
```

**A.** 0

**B.** 10

**C.** 20

**D.** 30

**E.** This code does not compile.

**10.** What is the output of the following code?

```
1.  import java.util.*;
2.
3.  public class DateSwap {
4.
```

```
5.    public static void swap(GregorianCalendar a, GregorianCalendar b)
6.    {
7.        GregorianCalendar temp = a;
8.        a = new GregorianCalendar(2012, 1, 1);
9.        b = temp;
10.   }
11.
12.   public static void main(String [] args) {
13.       GregorianCalendar one = new GregorianCalendar(2010, 1, 1);
14.       GregorianCalendar two = new GregorianCalendar(2011, 1, 1);
15.
16.       swap(one, two);
17.
18.       System.out.print(one.get(Calendar.YEAR));
19.       System.out.println(two.get(Calendar.YEAR));
20.   }
21. }
```

**A.** 20112010

**B.** 20102011

**C.** 20122011

**D.** 20122010

**E.** 20102012

**F.** This code does not compile.

**11.** When does the `String` object instantiated on line 4 become eligible for garbage collection?

```
1.  public class ReturnDemo {
2.
3.    public static String getName() {
4.        String temp = new String("Jane Doe");
5.        return temp;
6.    }
7.
8.    public static void main(String [] args) {
9.        String result;
10.       result = getName();
11.       System.out.println(result);
12.       result = null;
13.       System.gc();
14.   }
15. }
```

**A.** Immediately after line 4
**B.** Immediately after line 5
**C.** Immediately after line 10
**D.** Immediately after line 12
**E.** Immediately after line 13
**F.** Immediately after line 14

**12.** What is the output of the following code?

```
4. byte a = 40, b = 50;
5. byte sum = (byte) a + b;
6. System.out.println(sum);
```

**A.** Line 5 generates a compiler error.
**B.** 40
**C.** 50
**D.** 90
**E.** An undefined value

**13.** What is the output of the following code?

```
5. int x = 5 * 4 % 3;
6. System.out.println(x);
```

**A.** Line 5 generates a compiler error.
**B.** 2
**C.** 3
**D.** 5
**E.** 6

**14.** What is the output of the following code?

```
3. byte y = 14 & 9;
4. System.out.println(y);
```

**A.** Line 3 generates a compiler error.
**B.** 15
**C.** 14
**D.** 9
**E.** 8

**15.** What is the output of the following code?

```
1.  public class FinalTest {
2.
3.    public static void main(String [] args) {
4.        House h = new House();
5.        h.address = "123 Main Street";
6.        h = null;
7.        System.gc();
8.    }
9.  }
10.
11. class House {
12.    public String address;
13.
14.    public void finalize() {
15.        System.out.println("Inside House");
16.        address = null;
17.    }
18. }
```

**A.** There is no output.

**B.** `Inside House`

**C.** The output cannot be determined.

**D.** The code generates a compiler error.

**16.** Given the following class named `House`, which of the following statements is true? (Select two answers.)

```
1.  public class House {
2.    public String address = new String();
3.
4.    public void finalize() {
5.        System.out.println("Inside House");
6.        address = null;
7.    }
8.  }
```

**A.** `"Inside House"` is displayed just before a `House` object is garbage collected.

**B.** `"Inside House"` is displayed twice just before a `House` object is garbage collected.

**C.** The `finalize` method on line 4 never actually gets called.

**D.** There is no need to assign `address` to `null` on line 6.

**E.** The `String` object from line 2 is guaranteed to be garbage collected after its corresponding `House` object is garbage collected.

**17.** Which of the following statements is true about the following `BaseballTeam` class?

```
1.  public class BaseballTeam {
2.    private String city, mascot;
3.    private int numberOfPlayers;
4.
5.    public boolean equals(Object obj) {
6.        if(!(obj instanceof BaseballTeam)) {
7.            return false;
8.        }
9.        BaseballTeam other = (BaseballTeam) obj;
10.       return (city.equals(other.city)
11.               && mascot.equals(other.mascot));
12.   }
13.
14.   public int hashCode() {
15.       return numberOfPlayers;
16.   }
17. }
```

**A.** The class does not compile.

**B.** The class compiles but contains an improper `equals` method.

**C.** The class compiles but contains an improper `hashCode` method.

**D.** The class compiles and has proper `equals` and `hashCode` methods.

**18.** What is the output of the following code?

```
3. int x = 0;
4. String s = null;
5. if(x == s) {
6.    System.out.println("Success");
7. } else {
8.    System.out.println("Failure");
9. }
```

**A.** `Success`

**B.** `Failure`

**C.** Line 4 generates a compiler error.

**D.** Line 5 generates a compiler error.

**19.** What is the output of the following code?

```
3. int x1 = 50, x2 = 75;
4. boolean b = x1 >= x2;
5. if(b = true) {
```

```
6.    System.out.println("Success");
7. } else {
8.    System.out.println("Failure");
9. }
```

**A.** `Success`

**B.** `Failure`

**C.** Line 4 generates a compiler error.

**D.** Line 5 generates a compiler error.

**20.** What is the output of the following code?

```
5. int c = 7;
6. int result = 4;
7. result += ++c;
8. System.out.print(result);
```

**A.** 8

**B.** 11

**C.** 12

**D.** 15

**E.** 16

**F.** Line 7 generates a compiler error.

**21.** Determine the output of the following code when executed with the command:

```
java HelloWorld hello world goodbye
```

```
1.  public static class HelloWorld {
2.    public static void main(String [] args) {
3.        System.out.println(args[1] + args[2]);
4.    }
5.  }
```

**A.** `hello world`

**B.** `world goodbye`

**C.** `null null`

**D.** An `ArrayIndexOutOfBoundsException` occurs at runtime.

**E.** The code does not compile.

# Answers to Review Questions

1. A. The code does not contain any compiler errors. It is valid to define multiple classes in a single file as long as only one of them is `public` and the others have the default access.

2. C and E. C assigns the `-classpath` flag to the appropriate directory. E also set the class path correctly except `-cp` is used. The `-cp` and `-classpath` flags are identical. A uses an equals sign = with the `-classpath` flag, which is not the correct syntax. B and D set the class path to the wrong directory and also incorrectly refer to the `Main` class without its fully qualified name, which is `com.mycompany.Main`.

3. E. The `-d` flag creates the appropriate directory structure that matches the package name. In this case, that directory created is `c:\abcproject\deploy\a\b\c`. Therefore, C and D are wrong. A `NoClassDefFoundError` occurs if the compiler cannot find the source file, but in this example the `javac` command is executed from the same directory that contains the source file, so this error does not occur. A `ClassNotFoundException` is a runtime exception that is not thrown by a compiler, so B is incorrect.

4. A. Based on the definition of the `equals` method, two `Employee` objects are equal if they have the same `employeeId` field, so line 21 evaluates to `true` and `"Success"` is output, so B is incorrect. Line 6 successfully overrides `hashCode`, so C is incorrect. Line 10 is a valid overriding of `equals`, so D and E are incorrect.

5. A. B is incorrect because `hashCode` does not have to return a unique value (not that the compiler could determine if the value was unique anyway). C is incorrect because the `equals` method correctly overrides `equals` in `Object`. D is incorrect because a `ClassCastException` does not need to be handled or declared. E is incorrect because although `ISBN` is a `private` field, the `equals` method is within the class and therefore has access to the `private` field. Therefore, the code compiles successfully and the answer is A.

6. C. The reference `s1` points to a `String` object in the string pool because `"Canada"` is a literal string known at compile time. The reference `s2` points to a `String` object created dynamically at runtime, so this object is created on the heap. Therefore B is incorrect because `s1` and `s2` point to different objects. However, C is correct because `s1` and `s2` are both `String` objects that equal "`Canada`", so `s1.equals(s2)` evaluates to `true`. Because C is correct, A and D must be incorrect.

7. B and D. The `Date` object from line 5 has two references to it — `one` and `three` — and becomes eligible for garbage collection after line 10, so B is a true statement. The reference `four` is set to `null` on line 9, which does not affect the object from line 5. The `Date` object from line 6 only has a single reference to it — `two` — and therefore becomes eligible for garbage collection after line 11 when `two` is set to `null`, so D is a true statement.

8. A. A top-level class cannot be declared `private`, so line 1 causes a compiler error. This is one of those exam questions where you might waste a couple of minutes if you do not notice the compiler error right away. Don't forget to keep an eye out for these subtle types of compiler errors.

**9.** D. The code compiles, so E is incorrect. The `Car` object on line 8 has an initial `velocity` of 10 from line 19. The call to `go` on line 9 changes its `velocity` to 20. The `stolen` reference points to the same `Car` object, so calling `go` with the `stolen` argument changes the `Car` object's `velocity` to 30, so the correct answer is D.

**10.** B. The code compiles successfully, so F is incorrect. The two `GregorianCalendar` references are passed to the `swap` method, which does not change either object. In fact, the only thing swapped in the `swap` method is `b` getting assigned to `a`, but these changes do not affect the references `one` and `two`. Because the objects that `one` and `two` refer to are not changed in the `swap` method, the output is 20102011 and B is the correct answer.

**11.** D. The object on line 4 is referred to by the `temp` reference, which goes out of scope after line 5. However, the `result` reference gets a copy of `temp`, so it refers to the "`Jane Doe`" object until line 12 when `result` is set to `null`, at which point "`Jane Doe`" is no longer reachable and becomes immediately eligible for garbage collection. Therefore, the answer is D.

**12.** A. Line 5 generates a possible loss of precision compiler error. The cast operator has the highest precedence, so it is evaluated first, casting `a` to a byte (which is fine). Then the addition is evaluated, causing both `a` and `b` to be promoted to `int`s. The value 90, stored as an `int`, is assigned to `sum`, which is a byte. This requires a cast, so the code does not compile and therefore the correct answer is A. (This code would compile if parentheses were used around `(a + b)`.)

**13.** B. The `*` and `%` operators have the same level or precedence and are therefore evaluated left-to-right. The result of `5 * 4` is 20 and `20 % 3` is 2 (20 divided by 3 is 18; the remainder is 2). Therefore, the answer is B.

**14.** E. To evaluate the `&` operator, you need to express the numbers in binary and evaluate `&` on each column, as shown here:

```
14      =       0000 1110
9       =       0000 1001
14&9    =       0000 1000
```

The resulting binary number 00001000 is 8 in decimal, so the answer is E.

**15.** C. The code compiles successfully, so D is incorrect. Due to the unpredictable behavior of `System.gc`, the output cannot be determined. The `House` object from line 4 is eligible for garbage collection after line 6, and the call to `System.gc` may free its memory and cause "`Inside House`" to be displayed from the `finalize` method. However, the `System.gc` method may not free the memory of the `House` object, in which case there would be no output. Because A or B may occur, the answer is C.

**16.** A and D. Just before an object is garbage collected, its `finalize` method is invoked once, so A is true but B is incorrect. C is incorrect because it is just not a true statement. D is correct; there is no need to assign `address` to `null` because it is about to be deleted from memory. E is incorrect, though, because `address` may not be the only reference to the `String` object that `address` refers to.

**17.** C. The class compiles successfully, so A is incorrect. B is incorrect because an `equals` method can use any business logic you want to determine if two objects are equal. However, the rule for proper overriding of `equals` and `hashCode` is that if two objects are equal, they should generate the same hash code. The `hashCode` method does not properly follow this rule. Two teams with the same `city` and `mascot` but different `numberOfPlayers` would be equal but would generate different hash codes. Therefore, D is incorrect and the answer is C.

**18.** D. The variable `x` is an `int` and `s` is a reference. These two data types are incomparable because neither variable can be converted to the other variable's type.
The compiler error occurs on line 5 when the comparison is attempted, so the answer is D.

**19.** A. The code compiles successfully, so C and D are incorrect. The value of `b` after line 4 is `false`. However, the `if` statement on line 5 contains an assignment, not a comparison. The value of `b` is assigned to true on line 5, and the assignment operator returns `true`, so line 6 executes and displays `"Success"`.

**20.** C. The code compiles successfully, so F is incorrect. On line 7, `c` is incremented to 8 before being used in the expression because it is a pre-increment. The 8 is added to `result`, which is 4, and the resulting 12 is assigned to `result` and displayed on line 8. Therefore, the answer is C.

**21.** E. The class declaration on line 1 contains the `static` modifier, which is not a valid modifier for a top-level class. This causes a compiler error, so the correct answer is E.