SCJP Exam for J2SE 5

A Concise and Comprehensive Study Guide for The Sun Certified Java Programmer Exam

Paul Sanghera, Ph.D.

Apress®

SCJP Exam for J2SE 5: A Concise and Comprehensive Study Guide for The Sun Certified Java Programmer Exam

ISBN-13 (pbk): 978-1-59059-697-5

ISBN-10 (pbk): 1-59059-697-8

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Technical Reviewer: Simon Liu
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Kylie Johnston
Copy Edit Manager: Nicole LeClerc
Copy Editor: Bill McManus
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Cheu
Compositor: Lynn L'Heureux, M&M Composition, LLC
Proofreader: Kim Burton
Indexer: Julie Grady
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 4

Java Language Fundamentals

Exam Objectives

1.1 Develop code that declares classes (including abstract and all forms of nested classes), interfaces, and enums, and includes the appropriate use of package and import statements (including static imports).

7.1 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.

5.3 Explain the effect of modifiers on inheritance with respect to constructors, instance or static variables, and instance or static methods.

7.2 Given an example of a class and a command-line, determine the expected runtime behavior.

7.3 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.

7.4 Given a code example, recognize the point at which an object becomes eligible for garbage collection, and determine what is and is not guaranteed by the garbage collection system. Recognize the behaviors of System.gc and finalization.

7.5 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.

In the previous chapter, you learned about classes, methods, and interfaces. In this chapter, we explore various aspects of a class and its members (variables and methods). You will learn how to organize your classes into an application. You will also learn what can happen to variables when you pass them as arguments in a method call. Another important issue covered in this chapter is from where in the application you can access a class or its members. This depends upon the access modifiers assigned to a class or its members. Modifiers modify (further specify) the behavior of a class or its members, and you can specify them when you write the code. Finally, you'll learn how memory management is performed in Java. You instantiate a class into an object, and the object occupies some memory. You can free the memory from the object when you no longer need it, which is called garbage collection in Java.

So, the core issue in this chapter is nothing other than the fundamentals of the Java language. To enable you to put your arms around this issue, we explore three avenues: organizing your Java application, modifiers, and garbage collection.

Organizing Your Java Application

A Java application is composed of a set of files generally distributed in a directory structure. This set of files may comprise groups of files, called packages, based on their functionalities. Here are some bare-bones facts about these files:

- The files that contain the original source code are called source files and have the extension `.java`. All the code goes into classes, which are defined inside the `.java` files.
- When you compile a `.java` file, it produces a `.class` file corresponding to each class declared (defined) in the `.java` file, including nested classes and interfaces.
- The `.class` file has the same name as the class in the `.java` file to which it corresponds.
- The compiler searches for a class file when it encounters a reference to a class in a `.java` file. Similarly, the interpreter, during runtime, searches the `.class` files.
- Both the compiler and the interpreter search for the `.class` files in the list of directories listed in the `classpath` variable.

In order to compile and execute an application correctly, you need to understand the relationship between the class name, `classpath`, and package name and how these three elements determine the directory structure in which you are supposed to store the `.class` files.

We cover all of these concepts in this section. Let's begin by identifying the point in your application where the execution starts.

Entering Through the Main Gate

When you issue a command to execute a Java application, the Java virtual machine (JVM) loads the class mentioned in the command, and invokes the `main(…)` method of this class. In other words, the `main(…)` method of a class in a Java application is the starting point for the execution control. You write the application by writing the Java source (`.java`) files. A source file may contain interfaces or classes. One of the classes in the application must have the `main(…)` method with the signature:

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

The method `main(…)` must be declared `public`, `static`, and `void`. These keywords are explained here:

- `public`: The method can be accessed from the code outside the class in which it is defined (remember, it is invoked by the JVM, which exists outside the class in which it is defined).
- `static`: The method can be accessed without instantiating the class in which it is declared. Again, this keyword also allows the JVM to invoke this method without instantiating the class.
- `void`: The method does not return any data.

A source file may have one or more classes defined in it. Out of these classes, only one class at most may be declared `public`. If there is one class declared `public` in the file, then the file name must match the name of this `public` class. When the source file is compiled, it generates one class file (a file with the `.class` extension) corresponding to each class in the source file. The name of the generated class file matches the name of the corresponding class in the source file.

The parameter of type `array` in the `main(…)` method indicates that you can pass arguments to this method. You provide those arguments in the command line when you execute the application by specifying the class that contains the `main(…)` method. For example, consider Listing 4-1.

Listing 4-1. *TestArgs.java*

```
1. public class TestArgs {
2.  public static void main (String [] args) {
3.   System.out.println("Length of arguments array: " + args.length);
4.   System.out.println("The first argument: " + args[0]);
5.   System.out.println("The second argument: " + args[1]);
6.  }
7. }
```

You can compile this code by issuing the following command from the command line:

```
javac TestArgs.java
```

This generates a file called `TestArgs.class`. You can execute this program and pass in the arguments for the `main(…)` method. For example, you could issue the following command:

```
java  TestArgs  Ruth Srilatha
```

The output of execution follows:

```
Length of arguments array: 2
The first argument: Ruth
The second argument: Srilatha
```

■ **Note** In the `javac` command, you include the file extension `.java` in the name of the file that you want to compile. However, you do not use any file extension when issuing the execution command, `java`.

If you do not provide all the arguments in the command line that the `main(…)` method is expecting, you receive a runtime exception. Note the following two facts regarding the arguments of the `main(…)` method:

- The length of the arguments array is not fixed. It becomes equal to the number of arguments that you pass.
- The parameter name in the `main(…)` method does not have to be `args`; it could be any valid variable name. However, its type must be a `String` array.

Important points about the names are summed up here:

- A `.java` file name should match the name of a class in the file. If one of those classes is declared `public`, then the name of the `.java` file must match the name of the `public` class.
- There can be only one `public` class at maximum in a source file.
- The compiler generates a file with extension `.class` corresponding to each class in the source file that is compiled.
- The name of the `.class` file matches the name of the corresponding class.

■ **Caution** The name of a `.java` file must match the name of a class in the file. If the file has a `public` class, the file name must match the name of the `public` class.

And there is more to names.

What Is in a Name?

A beginner in Java usually runs into compiler and execution errors related to finding the classes. These errors arise due to the confusion about the namespace. So, if it happens to you, rest assured that you are not the only one. The package name, the class name, and the `classpath` variable are the three players that you must be familiar with to avoid any confusion in the namespace area.

You know from the previous section that the `.class` file name matches the name of the class. You can bundle related classes and interfaces into a group called a package. You store all the class files related to a package in a directory whose name matches the name of the package. How do you specify to which package a class belongs? You specify it in the source file in which you write the class by using the keyword `package`. There will be at most one package statement in a `.java` file. For example, consider a class `Student` defined in a file, as shown in Figure 4-1.

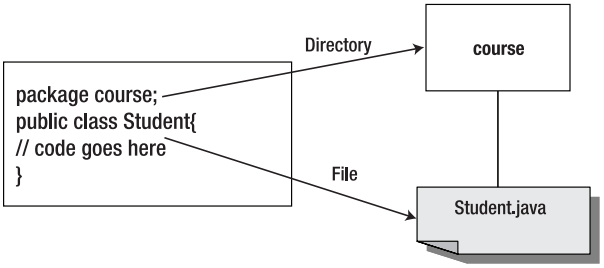

Figure 4-1. *The Student class is specified to be in the package course.*

The file name of the file in which the class `Student` exists will be `Student.java`, which will exist inside a directory named `course`, which may exist anywhere on the file system. The qualified name for the class is `course.Student`, and the path name to it is `course/Student.java`.

You can use this class in the code by specifying its qualified name, as in this example:

```
course.Student student1 = new course.Student();
```

However, it is not convenient to use the qualified name of a class over and over again. The solution to this problem is to import the package in which the class exists, and use the simple name for the class, as shown here:

```
import course.Student;
```

You place the `import` statement in the file that defines the class in which you are going to use the `Student` class. The `import` statement must follow any `package` statement and must precede the first class defined in the file. After you have imported the package this way, you can use the `Student` class by its simple name:

```
Student student1 = new Student();
```

You can import all the classes in the package by using the wildcard character *:

```
import  course.*;
```

However, you cannot use the wildcard character in specifying the names of the classes. For example, the following statement to refer to the class `Student` will generate a compiler error:

```
import course.Stud*;
```

You can have more than one `import` statement in a `.java` file. Bundling related classes and interfaces into a package offers the following advantages:

- It makes it easier to find and use classes.
- It avoids naming conflicts. Two classes with the same name existing in two different packages do not have a name conflict, as long as they are referenced by their fully qualified name.
- It provides access control. You will learn more about access control when access modifiers are discussed later in this chapter.

So, the files in a directory may be composed into a package by the declaration with keyword `package`:

```
package  <PackageName>;
```

Then you may use this package in another file by the statement with keyword `import`:

```
import <PackageName>;
```

In a file, package declaration must precede the `import` statement, which must precede the class definition. For example, the `.java` file with the following code will not compile because the `import` statement appears before the `package` statement:

```
import otherPackage;
package thisPackage;
class A { }
```

■ **Caution** In a Java file, the package declaration, the `import` statement, and the class definition must appear in this order.

You need to manage your source and class files in such a way that the compiler and the interpreter can find all the classes and interfaces that your program uses. Companies usually follow the convention of stating the package names with the reversed domain name. For example, the full package name of `course` in a fictitious company netalong.com will be `com.netalong.course`. Each component of a package name corresponds to a directory. In our example, the directory `com` contains the directory `netalong`, which in turn contains the directory `course`, and the `Student` file is in the `course` directory. This relationship is shown in Figure 4-2.

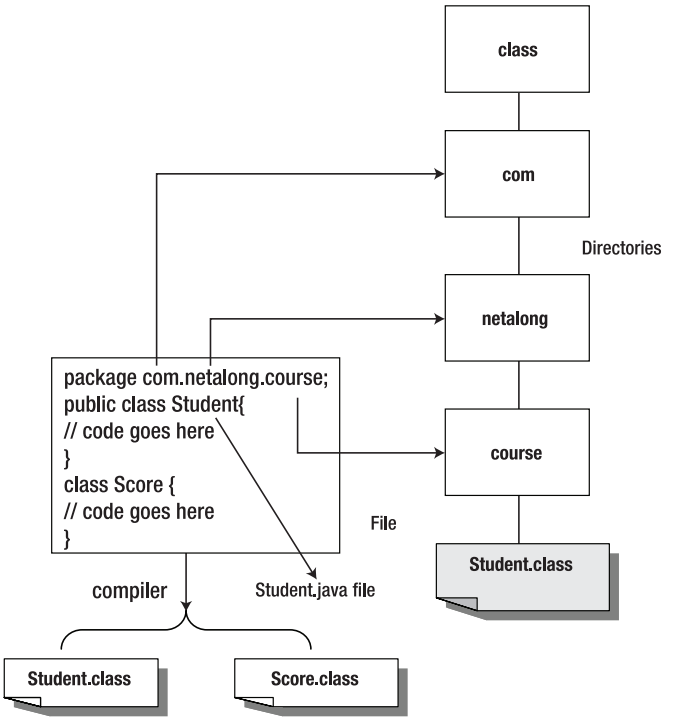

Figure 4-2. *Relationship between a package name and the corresponding directory structure*

The directory structure corresponding to the package name goes into a directory called the top-level directory. We assume in this example that the `com` directory is in the `class` directory. When you compile the source file `Student.java`, it will produce two class files, named `Student.class` and `Score.class`. It's a good practice to keep the class files separate from the source files. Let's assume that we put the class files in the `class/com/netalong/course` directory, and that the `source` and the `class` directories exist in the `C:\app` directory on a Microsoft Windows machine. So, the top-level directory for our package is

```
c:\app\class
```

When the compiler encounters a class name in your code, it must be able to find the class. In fact, both the compiler and the interpreter must be able to find the classes. As said earlier, they look for classes in each directory or a JAR (Java Archive) file listed in the `classpath`: an environment variable that you defined after installing JDK. In our example, the `classpath` must include this path name:

```
c:\app\class
```

■ **Note** A `classpath` is an environment variable whose value is a list of directories or JAR files in which the compiler and the interpreter searches for the class files. Following is an example of a `classpath`:

```
c:\jdk1.5.0_01; c:\jdk1.5.0_01\bin; c:\myclasses
```

Note that the directories are separated by a semicolon (;).

So, each directory listed in the `classpath` is a top-level directory that contains the package directories. The compiler and interpreter will construct the full path of a class by appending the package name to the top-level directory name (in the `classpath`) followed by the class name, as shown in Figure 4-3.

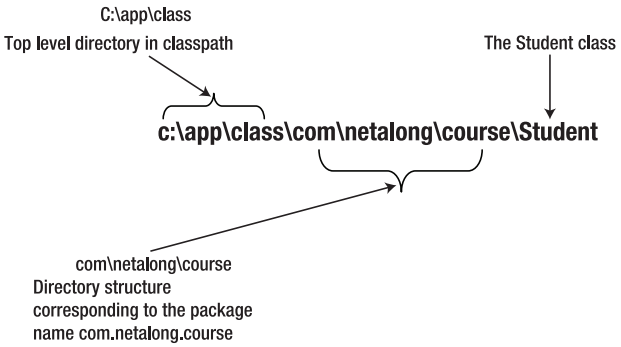

Figure 4-3. *This is how the compiler and the interpreter construct a full path name for a class mentioned with the name Student in a code file that imports the package com.netalong.course.*

The `.java` files and `.class` files live in directories. All the directories of an application can be compressed into what is called a JAR file.

The JAR Files

When the compiler (or the interpreter) encounters a class name in your code, it looks for that class in each directory or a JAR (Java Archive) file listed in the `classpath`. You can think of a JAR file as a tree of directories. Using the JAR facility of Java, the whole application (a number of source files and class files in a number of folders) can be compressed into one file, the JAR file. The file extension of a JAR file is `.jar`, and can be created with the `jar` command. For example, consider the following command:

```
jar  cf  myApp.jar topDir
```

This command will compress the whole directory tree (along with files in the directories) with `topDir` as the root directory into one file named `myApp.jar`. You can look at the list of directories and files in this JAR file by issuing the following command:

```
jar  -tf  myApp.jar
```

You can even execute your application contained in the JAR file by issuing the following command:

```
java  -jar   myApp.jar
```

When you specify the path to a JAR file, you must include the JAR file name at the end of the path, such as the following:

```
c:\jdk1.5.0\jre\lib\charsets.jar
```

■ **Caution** To make the `java -jar` command work, you need to specify which class contains the `main(…)` method by adding an entry like the following to the `MANIFEST.MF` file:

```
Main-Class: MyClass
```

In this entry, `MyClass` is the name of the class (and also the name of a `.class` file) that contains the `main(…)` method.

To the contrary, when you mention a directory name in a path, all the files in the directory will automatically be included in the path. By including a JAR file in the `classpath`, you can use the classes in it that may be referred to by another application. Another way of including the external classes in your application is by importing them, as discussed earlier in the chapter. J2SE 5.0 introduces a special kind of import called *static import*.

The Static Import

Assume that you have imported a package that contains a class, `A`. If you want to use a `static` member of that class, say a variable `v`, you need to refer to it as `A.v`. The *static import* feature introduced in J2SE 5.0 lets you import the `static` variables of a class so that you can refer to them without having to mention the class—that is, just as `v` instead of as `A.v` in our example.

As an example, consider Listing 4-2 without static import and Listing 4-3 with static import. Lines 3 and 4 of Listing 4-2 refer to `PI` and `E`, two `static` fields of the class `Math`, as `Math.PI` and `Math.E`. If you exclude the word `Math`, you will get a compiler error. Remember that the class `Math` is in the `java.lang` package, which is automatically imported when you compile your application.

Listing 4-2 imports the `static` members `PI` and `E` of the `Math` class in the `java.lang` package (lines 1 and 2). Now, you can use the imported `static` members without qualifying them with the class name (lines 5 and 6).

Listing 4-2. *NoStaticImport.java*

```
1. class NoStaticImport{
2.  public static void main(String[] args) {
3.   System.out.println("Pi:" + Math.PI);
4.   System.out.println("E:" + Math.E);
5.  }
6. }
```

Listing 4-3. *StaticImportTest.java*

```
1. import static java.lang.Math.PI;
2. import static java.lang.Math.E;
3. class StaticImportTest{
4.  public static void main(String[] args) {
5.   System.out.println("Pi:" + PI);
6.   System.out.println("E:" + E);
7.  }
8. }
```

The output from both Listing 4-2 and Listing 4-3 is the same:

```
Pi:3.141592653589793
E:2.718281828459045
```

Remember that you can use the static import feature for the classes that you write, as well, without inheriting them from those classes. Note that the static import declaration is very similar to the normal import declaration. A normal import declaration imports classes from a package, enabling you to use a class without package qualification (that is, without referring to the package name in the code), whereas a static import declaration imports `static` members from a class, enabling you to use the `static` members without class qualification (that is, without referring to the class name in the code).

Remember the following while using the static import feature:

- It is an `import` statement: `import` comes before `static`, even though the name of the feature is static import.
- You can use the wildcard * to import all the `static` members of a class just as you can use it to import all the classes of a package.
- You can use static import to import `static` variables, `static` object references, and `static` methods.

So, a Java application is a collection of packages, which in turn are collections of classes. The classes are composed of data and methods that operate on data using some logic. One of the ways to pass data to a method is through its arguments.

Passing Arguments into Methods

You learned in Chapter 3 that a method declaration may have parameters defined in the parentheses following the method name. The values of these parameters are called *arguments* and can be passed during a method call. These parameters may be either primitive variables or reference variables. Assume you declare a variable in your method, and then you pass that variable as an argument in a method call. The question is: What kind of effect can the called method have on the variable in your method?

The key point to remember is that regardless of whether the passed variable is a primitive or a reference variable, it is always the copy of the variable, and not the original variable itself. Technically speaking, it is pass-by-value and not pass-by-reference within the same virtual machine.

Passing a Primitive Variable

Recall that a primitive variable holds a data item as its value. When a primitive variable is passed as an argument in a method call, only the copy of the original variable is passed. Therefore any change to the passed variable in the called method will not affect the variable in the calling method.

As an example, consider Listing 4-4. An integer variable `score` is defined in the `main(…)` method (line 3) and is passed as an argument in a method call to the `modifyStudent(…)` method (line 5). The value of the passed variable is changed in the called method (line 9).

Listing 4-4. *Student.java*

```
1. class Student {
2. public static void main (String [] args) {
3.  int score = 75;
4.  Student st = new Student();
```

```
5.  st.modifyStudent(score);
6.  System.out.println("The original student score: " + score);
7. }
8. void modifyStudent(int i){
9.    i = i+10;
10.   System.out.println("The modified student score: " + i);
11. }
12.}
```

The output from the execution of Listing 4-4 follows:

```
The modifed student score: 85
The original student score: 75
```

This demonstrates that the change in value of the passed variable in the called method did not affect the value of the original variable in the calling method. What if we pass a reference variable as an argument in a method call?

Passing a Reference Variable

Recall that an object reference variable points to an object, and it is not the object itself. When you pass a reference variable in a method, you pass a copy of it and not the original reference variable. Because the copy of the reference variable points to the same object to which the original variable points, the called method can change the object properties by using the passed reference. Now, changing the object and the reference to the object are two different things, so note the following:

- The original object can be changed in the called method by using the passed reference to the object.
- However, if the passed reference itself is changed in the called method, for example, set to `null` or reassigned to another object, it has no effect on the original reference variable in the calling method. After all, it was a copy of the original variable that was passed in.

As an example, consider Listing 4-5.

Listing 4-5. *TestRefVar.java*

```
1.  class TestRefVar {
2.   public static void main (String [] args) {
3.    Student st = new Student("John", 100);
4.    System.out.println("The original student info:");
5.    System.out.println("name: " + st.getName() + " score: " +
6.    st.getScore());
7.    TestRefVar tr = new TestRefVar();
8.    tr.modifyRef(st);
9.    System.out.println("The modified student info in the calling method:");
10.      System.out.println("name: " + st.getName() + " score: " + st.getScore());
11. }
12. void modifyRef(Student student){
13.        student.setScore(50);
14.        student = new Student("Mary", 75);
15.        System.out.println("The modified student info in the called method:");
16.        System.out.println("name: " + student.getName() + " score: " +
                 student.getScore());
17. }
18. }
```

```
19. class Student {
20.  int score;
21.  String name;
22.  Student(String st, int i){
23.   score=i;
24.   name=st;
25. }
26. String getName(){
27.     return name;
28. }
29. int getScore(){
30.  return score;
31. }
32. void setScore(int i){
33.  score = i;
34. }
35.}
```

The following is the output from the execution of Listing 4-5:

```
The original student info:
name: John score: 100
The modified student info in the called method:
name: Mary score: 75
The modified student info in the calling method:
name: John score: 50
```

The output demonstrates that the called method used the passed reference to change the score of the existing student `John (line 13)`. Then the called method re-pointed the passed reference to the new student named `Mary (line 14)`. However, it did not affect the reference in the calling method, which is still pointing to `John (line 10)`.

You already know about classes and packages containing groups of classes. The next question to ask is: From where can these classes and their members (variables and methods) be accessed? This is determined by the access modifiers that you can assign to the classes and their members when you write the classes.

Using Access Modifiers

Access modifiers, also called visibility modifiers, determine the accessibility scope of the Java elements they modify. If you do not explicitly use an access modifier with a Java element, the element implicitly has the default access modifier. The explicit access modifiers may be used with a class and its members (that is, instance variables and methods). They cannot be used with the variables inside a method.

■ **Caution** Data members of a method cannot explicitly use access modifiers.

The Java language offers three explicit access modifiers, `public`, `protected`, and `private`, and a default modifier, which is enforced when you do not specify a modifier.

The public Modifier

The `public` modifier makes a Java element most accessible. It may be applied to classes and to their members (that is, instance variables and methods). A class, variable, or method, declared as `public`, may be accessed from anywhere in the Java application. For example, you declare the `main(…)` method of any application `public` so that it may be invoked from any Java runtime environment. Other `public` methods may be called from anywhere inside the application. However, generally speaking, it is not a good object-oriented programming practice to declare the instance variables `public`. If you declare them `public`, they could be accessed directly, whereas they should always be accessed through the class methods. For example, consider the code fragment in Listing 4-6.

Listing 4-6. *InstanceTest.java*

```
1. class MyClass {
2.   public int  myNumber = 10;
3.      public int getMyNumber(){
4.       return myNumber;
5.     }
6.   }
7.  class InstanceTest {
8.    public static void main(String[] args) {
9.        MyClass  mc = new MyClass();
10.      System.out.println (" The value of myNumber is " + mc.myNumber);
11.      System.out.println (" The value returned by the method is " +
              mc.getMyNumber());
12.   }
13.  }
```

The output of Listing 4-6 follows:

```
The value of myNumber is 10
The value returned by the method is 10
```

Note that the `myNumber` variable is directly accessed in line 10, and also accessed using a method in line 11. If you replace the access modifier `public` with `private` in line 3 and try to recompile it, line 10 will generate the compiler error because you cannot directly access a `private` class member from outside the class.

The private Modifier

The `private` modifier makes a Java element (a class or a class member) least accessible. The `private` modifier cannot be applied to a top-level class. It can be applied only to the members of a top-level class—that is, instance variables, methods, and inner classes. Recall that a class that is not defined inside another class is called a top-level class. A `private` member of a class may only be accessed from the code inside the same class in which this member is declared. It can be accessed neither from any other class nor from a subclass of the class in which it is declared.

■ **Caution** A top-level class cannot be declared `private`; it can only be `public`, or default (that is, no access modifier is specified).

As an example, consider Listing 4-7.

Listing 4-7. *TestPrivateTest.java*

```
1. class PrivateTest {
2.
3.   // public int  myNumber = 10;
4.   private int  myNumber = 10;
5.  public int getMyNumber(){
6.     return myNumber;
7.   }
8. }
9. class SubPrivateTest extends PrivateTest {
10.    public void printSomething(){
11.     System.out.println (" The value of myNumber is " + this.myNumber);
12.      System.out.println (" The value returned by the method is " +
               this.getMyNumber());
13.    }
14. }
15. class TestPrivateTest{
16.   public static void main(String[] args) {
17.      SubPrivateTest spt = new SubPrivateTest();
18.      spt.printSomething();
19.  }
20. }
```

This code will not compile because line 11 will generate a compiler error. You cannot access a private data variable of the parent class directly, as the private class members are not inherited. If you comment out line 11 and then compile and execute the program, the output of Listing 4-7 follows:

```
The value returned by the method is 10.
```

In Listing 4-7, by declaring the data variable `myNumber private`, you have disabled the direct access to it from outside the class in which it is declared, and have enforced the rule that this data variable may only be accessed from inside the class, for example, through the method `getMyNumber()`. This is a good programming practice in the world of object-oriented programming.

The fact that you cannot directly access the private members of a class from its subclass can be looked upon this way: a subclass does not inherit the `private` members of its parent class. The `public` and `private` access modifiers are on the two extremes of access: access from everywhere and access from nowhere outside of the class. There is another access modifier called `protected` that covers the middle ground between these two extremes.

The protected Modifier

The `protected` modifier makes a class member more accessible than the `private` modifier would, but still less accessible than `public`. This modifier may be applied only to class members—that is, the variables, methods, and inner classes—but not to the class itself. A class member declared `protected` is accessible to the following elements:

- All the classes in the same package that contains the class that owns the `protected` member.
- All the subclasses of the class that owns the `protected` member. These subclasses have access even if they are not in the same package as the parent class.

For example, consider these two code fragments:

```
1. package networking;
2. class Communicator {
3.         void sendData() {}
4.         protected void receiveData() {}
5. }
```

```
1. package internetworking;
2. import networking.*;
3. class Client extends Communicator {
4.    void  communicate(){
5.       receiveData();
6.    }
7. }
```

The class `Client` is a subclass of the class `Communicator`. But both classes are in different packages. The method `receiveData()` is declared `protected` in the class `Communicator`. It may be called from the class `Client` (line 5), even though `Communicator` and `Client` are in different packages. This is because `Client` is a subclass of `Communicator`.

■ **Caution** You cannot specify any modifier for the variables inside a method, and you cannot specify the `protected` modifier for a top-level class.

You don't have to specify an access modifier for a class or a class member, in which case the access is assumed to be default.

The Default Modifier

You cannot specify any modifier for the variables inside a method, and you cannot specify the `protected` modifier for a class. The compiler only tells you what access modifier you cannot specify for a given element; however, it does not require you to specify any access modifier for any element. When you do not specify any access modifier for an element, it is assumed that the access is default. In other words, there is no keyword `default` for the default modifier. If an element does not explicitly use any modifier, the default access is implied. It may be applied to a class, a variable, or a method.

■ **Caution** Although there is no keyword `default` for the default modifier, there is a Java keyword `default` related to the `switch` statement, as discussed in Chapter 6.

A class or a class member declared default (no access modifier specified) is accessible from anywhere (classes or subclasses) in the same package in which the accessed class exists. As an example, consider the following code fragment:

```
1. package internetworking;
2. import networking.*;
3. class Client extends Communicator {
4.    void  communicate(){
5.       receiveData();
6.       sendData();  // compiler error.
7.   }
8. }
```

Line 5 would generate a compiler error, because the method `sendData()` is declared default in the class `Communicator`, which is in a different package:

```
1. package networking;
2. class Communicator {
3.         void sendData() {}
4.         protected void receiveData() {}
5. }
```

Note the difference between the `protected` modifier and the default modifier. A member with a default access can be accessed only if the accessing class belongs to the same package, whereas a member with the `protected` modifier can be accessed not only from the same package but also from a different package if the accessing class is a subclass of the accessed class.

■ **Note** A method cannot be overridden to be less public.

The final word about access modifiers: a method may not be overridden to be less accessible. For example, a `protected` method may be overridden as `protected` or `public`, but not as `private` or default. Recall that overriding a method means reimplementing a method inherited from a class higher in the hierarchy. You will learn more about method overriding in Chapter 5.

In a nutshell, you can use access modifiers to protect both variables and methods of a class. The Java language supports four distinct access levels for member variables and methods by offering four access modifiers: `private`, `protected`, `public`, and default (i.e. left unspecified). These access modifiers are summarized in Table 4-1.

Table 4-1. *Access Level that a Class Has Granted to Other Classes by Using Different Modifiers*

Access Modifier	Class	Subclass	Package	World
`private`	Yes	No	No	No
`protected`	Yes	Yes	Yes	No
`public`	Yes	Yes	Yes	Yes
Default	Yes	No	Yes	No

The first column in Table 4-1 specifies the possible access modifiers of a class member. The headings of the other columns identify the elements that are trying to access this member. The value Yes means an element has access to the member with the specified access modifier. For example, the second column indicates that a class always has access to its own members regardless of what access modifier they have. The third column indicates whether the subclasses of the class, regardless of which package they are in, have access to the class members with different modifiers. The fourth column indicates whether another class in the same package as the class in question has access to the class members. The fifth column indicates whether all other classes have access to the class members with different access modifiers.

So, the decision to use the modifiers is generally driven by striking a balance between accessibility and security. The underlying effect of any modifier, in general, is to modify (or further specify) the behavior of a class or a class member. The access modifiers specify the access behavior, while there are some non-access modifiers that specify how a class or a class member can be used. I call them *usage modifiers* in this book.

Understanding Usage Modifiers

There are some modifiers that are not access modifiers but still modify the way a class or a class member is to be used. Collectively, we call these modifiers the *usage modifiers*. Some of them, such as `final`, `abstract`, and `static`, may be more familiar than others to a beginner.

The final Modifier

The `final` modifier may be applied to a class, a method, or a variable. It means, in general, that the element is final. The specific meaning slightly depends upon the element it applies to. If the element declared `final` is a variable, that means the value of the variable is constant, and cannot be changed. If a class is declared `final`, it means the class cannot be extended, and a `final` method cannot be overridden.

For example, consider Listing 4-8. The variable `dime` in the class `Calculator` is declared `final`. Also, the object reference `calc` in class `RunCalculator` is declared `final`. The code lines 11 and 13 will generate compiler errors because they attempt to modify the values of the `final` variables `calc` and `dime`, respectively. However, note that line 12 will compile fine, which shows that the `final` object reference may be used to modify the value of a non-final variable.

Listing 4-8. *RunCalculator.java*

```
1. class Calculator {
2.  final int dime = 10;
3.  int count = 0;
4.  Calculator (int i) {
5.     count = i;
6.  }
7. }
8. class RunCalculator {
9.     public static void main(String[] args) {
10.      final Calculator calc = new Calculator(1);
11.      calc = new Calculator(2); // compiler error.
12.      calc.count = 2;  //ok
13.      calc.dime  = 11; // compiler error.
14.     System.out.println("dime: " + calc.dime);
15.   }
16.  }
```

If you comment out lines 11 and 13, the code will compile, and the result of execution will be as follows:

```
dime: 10
```

■ **Caution** Although a `final` object reference may not be modified, it can be used to modify the value of a non-final variable in the object to which the `final` object reference refers.

If you declare a `final` method inside a non-final class, it will be legal to extend the class, but you cannot override the `final` method of the parent class in the subclass. Similarly, you can pass the `final` variable to a method through arguments, but you cannot change their value even inside the method.

So, the `final` modifier is related to changing the value of a variable. There is another property of a variable, and that is visibility: from where can you see the value (or a change in value) of a variable?

The static Modifier

The `static` modifier can be applied to variables, methods, and a block of code inside a method. The `static` elements of a class are visible to all the instances of the class. As a result, if one instance of the class makes a change to a `static` element, all the instances will see that change.

Consider Listing 4-9. The variable `instanceCounter` is declared `static` in line 2, and another variable, `counter`, is not declared `static` in line 3. When an instance of the class `StaticExample` is created, both variables are incremented by one (lines 5 and 6). Each instance has its own copy of the variable `counter`, but they share the variable `instanceCounter`. A `static` variable belongs to the class, and not to a specific instance of the class, and therefore is initialized when the class is loaded. A `static` variable may be referenced by an instance of the class (lines 13 and 14) in which it is declared, or by the class name itself (line 15).

Listing 4-9. *RunStaticExample.java*

```
1. class StaticExample {
2.  static int instanceCounter = 0;
3.  int  counter = 0;
4.  StaticExample() {
5.         instanceCounter++;
6.         counter++;
7.   }
8. }
9. class RunStaticExample {
10.   public static void main(String[] args) {
11.          StaticExample se1 = new StaticExample();
12.          StaticExample se2 = new StaticExample();
13.          System.out.println("Value of instanceCounter for se1: " +
                    se1.instanceCounter);
14.          System.out.println("Value of instanceCounter for se2: " +
                    se2.instanceCounter);
15.          System.out.println("Value of instanceCounter: " +
                    StaticExample.instanceCounter);
16.          System.out.println("Value of  counter for se1: " + se1.counter);
17.          System.out.println("Value of  counter for se2: " + se2.counter);
18.  }
19.}
```

The following is the output from Listing 4-9:

```
Value of instanceCounter for se1: 2
Value of instanceCounter for se2 2
Value of instanceCounter: 2
Value of  counter for se1: 1
Value of  counter for se2 1
```

The following line of code outside the `StaticExample` class will set the value of `instanceCounter` to 100 for all the instances of the class `StaticExample`:

```
StaticExample.instanceCounter = 100;
```

Just like a `static` variable, a `static` method also belongs to the class in which it is defined, and not to a specific instance of the class. Therefore, a `static` method can only access the `static` members of the class. In other words, a method declared `static` in a class cannot access the non-static variables and methods of the class. Because a `static` method does not belong to a particular instance of the class in which it is defined, it can be called even before a single instance of the class exists. For example, every Java application has a method `main(…)`, which is the entry point for the application execution. It is executed without instantiating the class in which it exists. Also, a `static` method may not be overridden as non-static and vice versa.

■ **Note** A `static` method cannot access the non-static variables and methods of the class in which it is defined. Also, a `static` method cannot be overridden as non-static.

In addition to `static` variables and `static` methods, a class may have a `static` code block that does not belong to any method, but only to the class. For example, you may like to execute a task before the class is instantiated, or even before the method `main(…)` is called. In such a situation, the `static` code block will help because it will be executed when the class is loaded.

Consider Listing 4-10. When you run the application `RunStaticCodeExample`, the `static` variable `counter` (line 2) is initialized, and the `static` code block (lines 3 to 6) are executed at the class `StaticCodeExample` load time. Then, the method `main(…)` is executed.

Listing 4-10. *RunStaticCodeExample.java*

```
1.  class StaticCodeExample {
2.  static int counter=0;
3.  static {
4.       counter++;
5.       System.out.println("Static Code block: counter: " + counter);
6.  }
7.  StaticCodeExample() {
8.          System.out.println("Constructor:  counter: " + counter);
9.  }
10.  static {
11.                System.out.println("This is another static block");
12.               }
13.}
14.  public class  RunStaticCodeExample {
15.  public static void main(String[] args) {
16.        StaticCodeExample sce = new StaticCodeExample();
17.        System.out.println("main: " + sce.counter);
18.  }
19.}
```

The output from Listing 4-10 follows:

```
Static Code block: counter: 1
This is another static block
Constructor: counter: 1
main: counter: 1
```

This output demonstrates that the `static` code block is executed exactly once, and before the class constructor is executed—that is, at the time the class is loaded. It will never be executed again

during the execution lifetime of the application. Note that all the `static` blocks will be executed in order before the class initialization regardless of where they are located in the class.

So, remember these points about the `static` modifier:

- The `static` elements (variables, methods, and code fragments) belong to the class and not to a particular instance of the class.
- Any change in a `static` variable of a class is visible to all the instances of the class.
- A `static` variable is initialized at class load time. Also, a `static` method and a `static` code fragment are executed at class load time.
- A `static` method of a class cannot access the non-static members of the class.
- You cannot declare the following elements as `static`: constructor, class (that is, the top-level class), interface, inner class (the top-level nested class can be declared `static`), inner class methods and instance variables, and local variables.
- It is easier to remember what you can declare `static`: top-level class members (methods and variables), the top-level nested class, and code fragments.

The `static` modifier cannot be applied to a top-level class or a class constructor. However, you can apply the `final` modifier to a class, which means the class cannot be extended. You may face the opposite situation, where you want the class to be extended before it can be instantiated. This situation is handled by the `abstract` modifier.

The abstract Modifier

The `abstract` modifier may be applied to a class or a method, but not to a variable. A class that is declared `abstract` cannot be instantiated. Instantiation of an `abstract` class is not allowed, because it is not fully implemented yet.

There is a relationship between an `abstract` class and an `abstract` method. If a class has one or more `abstract` methods, it must be declared `abstract`. A class may have one or more `abstract` methods in any of the following ways:

- The class may have one or more `abstract` methods originally defined in it.
- The class may have inherited one or more `abstract` methods from its superclass, and has not provided implementation for all or some of them.
- The class declares that it implements an interface, but does not provide implementation for at least one method in the interface.

In any of the preceding cases, the class must be declared `abstract`. However, if there is no `abstract` method in the class, it could still be declared `abstract`. Even in this case, it cannot be instantiated, obviously.

■ **Note** A class with one or more `abstract` methods must be declared `abstract`. However, a class with no `abstract` method may also be declared `abstract`. An `abstract` class cannot be instantiated.

Consider Listing 4-11, in which the method `draw()` in class `Shape` is `abstract`. Note its signatures:

```
abstract void draw();
```

Each of the subclasses of `Shape` (that is, `Cone` and `Circle`) implements its own version of the method `draw()`. Therefore, when the method `draw()` is called from `main(…)` in class `RunShape`, its action depends upon which implementation of `draw()` is invoked.

Listing 4-11. *RunShape.java*

```
1. abstract class Shape {
2.   abstract void draw();  //Note that there are no curly braces here.
3.   void message() {
4.       System.out.println("I cannot live without being a parent.");
5.   }
6. }
7.  class Circle extends Shape {
8.   void draw() {
9.       System.out.println("Circle drawn.");
10.   }
11. }
12. class Cone extends Shape {
13.   void draw() {
14.       System.out.println("Cone drawn.");
15.   }
16.  }
17.  public class RunShape {
18.   public static void main(String[] args) {
19.       Circle circ = new Circle();
20.       Cone cone = new Cone();
21.       circ.draw();
22.       cone.draw();
23.       cone.message();
24.   }
25. }
```

The output from Listing 4-11 follows:

```
Circle drawn.
Cone drawn.
I cannot live without being a parent.
```

Notice that the effect of calling the same method `draw()` depends upon the caller. This feature of an object-oriented language is called *polymorphism*. Note that you can also implement polymorphism by simply overriding the method of the parent class in the subclasses. (You will learn more about polymorphism in Chapter 5.) However, in case of an `abstract` method, the overriding is enforced. If a subclass leaves at least one method unimplemented that was declared `abstract` in the parent class, it cannot be instantiated. Also note the difference between an `abstract` class and an interface from the perspective of a subclass. A subclass that extends an `abstract` class only has to implement all the unimplemented methods of the `abstract` class if you need to instantiate the subclass, whereas a subclass implementing an interface has to provide implementation for all the interface methods.

The `abstract` class is to some extent opposite to the `final` class. A `final` class cannot be extended, whereas an `abstract` class must be extended (before it can be instantiated). An `abstract` class or an `abstract` method means it's not fully implemented yet. So, if you want to use it, you have to implement it. In case of a method, there is another way of saying the same thing when the implementation is not the responsibility of the programmer who extends the class: the `native` modifier.

The native Modifier

In your applications, sometimes you will want to use a method that exists outside of the JVM. In this case, the `native` modifier can help you. The `native` modifier can only apply to a method. Like `abstract`, the keyword `native` indicates that the implementation of the method exists elsewhere. In case of

abstract, the implementation may exist in a subclass of the class in which the abstract method is declared. In case of native, the implementation of the method exists in a library outside of the JVM.

The native method is usually implemented in a non-Java language such as C or C++. Before a native method can be invoked, a library that contains the method must be loaded. The library is loaded by making the following system call:

```
System.loadLibrary("<libraryName>");
```

For example, the following code fragment presents an example of loading a library named NativeMethodsLib that contains a method (function) named myNativeMethod():

```
1. class MyNativeExample {
2.    native void myNativeMethod();
3.    static {
4.       System.loadLibrary("NativeMethodLib");
5.    }
6. }
```

Notice that the library is loaded in a static code block (lines 3 to 5). Therefore, the library is loaded at the class load time, so it is there when a call to the native method is made. You can use the native method in the same way as you use a non-native method. For example, the following two lines of code would invoke the native method:

```
MyNativeExample myNative  =  new MyNativeExample();
myNative.myNativeMethod();
```

■ **Note** The exam does not require you to know how to use the native methods. Just remember that it is a modifier (and hence a Java keyword) and that it can only be applied to methods.

The native modifier applies only to methods, while another modifier called transient applies only to variables.

The transient Modifier

When an application is running, the objects live in the random access memory (RAM) of the computer. This limits the scope and life of the object. However, an object may be stored in persistent storage (say disk) outside of the JVM, for later use by the same application, or by a different application. The process of storing an object is called *serialization*. For an object to be serializable, the corresponding class must implement the interface Serializable, or Externalizable.

So, the transient modifier is related to storing an object on the disk. Such storage is called the object's *persistent state*. A variable declared transient is not stored, and hence does not become part of the object's persistent state. One use of transient is to prevent a security-sensitive piece of data from copying to a file where there is no security mechanism in effect.

The transient modifier can only be applied to instance variables. When you are declaring an instance variable transient, you are instructing the JVM not to store this variable when the object in which it is declared is being serialized.

In a multithreaded environment, more than one process may try to access the same class element concurrently. To handle that situation, there are a couple of modifiers that you need to know about.

The Thread-Related Modifiers

A computer program may have launched more than one process executing concurrently. This is called *multithreaded programming*, and you will learn more about it in Chapter 10. But for now, just imagine that if there are more than one process in a program executing concurrently, they may attempt to access a class element at the same time. There are a couple of modifiers that relate to such a situation.

The volatile Modifier

Like the `transient` modifier, the `volatile` modifier only applies to instance variables. The variables declared `volatile` are subject to asynchronous modifications. In other words, declaring a variable `volatile` informs the compiler that this variable may be changed unexpectedly by other parts of the program. So, the compiler takes some special precautions to keep this variable properly updated. The `volatile` variables are generally used in multithreaded or multiprocessor environments. The `volatile` modifier tells the accessing thread that it should synchronize its private copy of the variable with the master copy in the memory.

The synchronized Modifier

The `synchronized` modifier is used in multithreaded programming to control access to critical sections in the program. This modifier is discussed in detail in Chapter 10 in conjunction with threads.

You have explored a multitude of modifiers. Let's take a look at the big picture.

Modifiers: The Big Picture

As you have noticed, not all modifiers can be applied to all Java elements such as classes, methods, and variables. For example, classes cannot be declared `private` and methods cannot be declared `transient` or `volatile`. Table 4-2 summarizes the use of different modifiers by Java classes and class members. Note that the constructors can use only access modifiers and no other type of modifiers. To the contrary, a code block cannot use any explicit access modifier. To be specific, it can only use `static` or `synchronized` modifiers.

Table 4-2. *Summary of Modifiers Used by Java Classes and Class Members*

Modifier	Top-Level Class	Variable	Method	Constructor	Code Block
`public`	Yes	Yes	Yes	Yes	No
`private`	No	Yes	Yes	Yes	No
`protected`	No	Yes	Yes	Yes	No
Default	Yes	Yes	Yes	Yes	N/A
`final`	Yes	Yes	Yes	No	No
`static`	No	Yes	Yes	No	Yes
`abstract`	Yes	No	Yes	No	No
`native`	No	No	Yes	No	No
`transient`	No	Yes	No	No	No
`volatile`	No	Yes	No	No	No
`synchronized`	No	No	Yes	No	Yes

All of these modifiers are specified with different Java elements in a Java application. Applications running on a computer use memory, which makes memory management a significant issue for any programming language. Because Java is a relatively high-level language, the memory management in Java is automatic. However, to make it more efficient, you need to understand *garbage collection*—that is, freeing memory from objects that are no longer in use.

Understanding Garbage Collection in Java

When you create an object by instantiating a class, the object is put on the heap; in other words, it uses some memory. A Java application creates and uses objects. After an object in memory has been used and is no longer needed, it is sensible to free memory from that object. The process of freeing memory from the used objects is called *garbage collection*. How do you accomplish this in Java?

In Java, garbage collection is done automatically by what is called the *garbage collector*.

Understanding the Garbage Collector

The garbage collector in Java automates memory management by freeing up the memory from objects that are no longer in use. The advantage of this is that you do not need to code the memory management into your application. The price you pay for this service is that you have no control over when the garbage collector runs. There are two things that you can do in the code to help memory management:

- Make an object eligible for garbage collection, because a garbage collector will only free up memory from an eligible object.
- Make a request for garbage collection by making a system call to the garbage collector: `System.gc();`.

You can also invoke the `gc()` method by using an instance of the `Runtime` class that a running application always has. You get hold of this instance by calling the `static` method `getRuntime()` of the `Runtime` class:

```
Runtime rt = Runtime.getRuntime();
```

Then, you can use this instance to invoke methods in order to perform some runtime tasks, such as to get memory information or to run the garbage collector:

```
rt.gc();
rt.getTotalMemory() // Returns the total amount of memory allocated to the JVM.
rt.freeMemory()   // Returns the amount of free JVM memory.
```

These uses of the `Runtime` class are demonstrated in Listing 4-12.

Listing 4-12. *RuntimeTest.java*

```
1. class RuntimeTest {
2.   public static void main (String [] args) {
3.     Runtime rt = Runtime.getRuntime();
4.     System.out.println("JVM free memory before running gc: " + rt.freeMemory());
5.     rt.gc();
6.     System.out.println("JVM free memory after running gc: " + rt.freeMemory());
7.   }
8. }
```

Remember that an application cannot create its own instance of the `Runtime` class. Therefore the following code will be invalid:

```
new Runtime().gc();
```

A call to the garbage collector is no guarantee that the memory will be free. It is possible, for example, that the JVM *in which* your program is running did not even implement the `gc()` method. The Java language specification allows a dummy `gc()` method.

The basic requirement for garbage collection is that you must make your object eligible for garbage collection. An object is considered eligible for garbage collection when there is no reference pointing to it. You can remove the references to an object in two ways:

- Set the object reference variable pointing to the object to `null`; for example:

```
myObject = null;
```

- Reassign a reference variable of an object to another object. For example, if a reference variable `myObject` is pointing to an object of the `MyClass` class, you can free this object from this reference by pointing the reference to another object:

```
myObject =  new YourClass();
```

Now, the object reference `myObject` is pointing to an object of the class `YourClass` and not to an object of `MyClass`, to which it was pointing previously.

What if you want an object to clean up its state before it is deleted? Well, you can declare the `finalize()` method in the class, and this method will be called by the garbage collector before deleting any object of this class.

The finalize() Method

The object that has no object references pointing to it can be deleted by the garbage collector to reclaim the memory. If the object has a `finalize()` method, it will be executed before reclaiming the memory in order to give the object a last chance to clean up after itself—for example, to release the resources that the object was using.

The `finalize()` method is inherited from the `Object` class by any class you define. The signature of the `finalize()` method in the object class is shown here:

```
protected void finalize()
```

You can override this method in your class. The Java programming language specifies that the `finalize()` method will be called before the object memory is reclaimed, but it does not guarantee exactly when it will happen. Remember that the `finalize()` method that your class inherited does not do anything. If you want your object to clean up after itself, you have to override the `finalize()` method. Then, what is the point of putting the `finalize()` method in the `Object` class? It makes it safe for the `finalize()` method of any class to invoke the `finalize()` of the superclass, as shown here:

```
protected void finalize() {
 super.finlaize();
 // clean up code follows.
}
```

This is generally a good practice.

■ **Note** Unlike a constructor, a `finalize()` method in any class will not automatically call the `finalize()` method of the super class. You need to call it explicitly, if you want to.

When an object is instantiated from a class, it is called *reachable* and *unfinalized*. When no reference is pointing to an object, the object can only be reached by the `finalize()` method, and hence it is called *finalizer reachable*. However, it is possible to make the object reachable again for any live thread by creating a reference to this object in its `finalize()` method. The `finalize()` method for an object is only run once. If you make an object ineligible for garbage collection in its `finalize()` method, it does not mean that the object will never be garbage collected because its `finalize()` method now will never be called. The object can still become eligible for garbage collection when it has no reference pointing to it. The only difference is that, this time, the garbage collector will remove it without calling its `finalize()` method, because it has already been called. The garbage collector is not guaranteed to be invoked, and thus the `finalize()` method is not guaranteed to be called. It is a good practice for you, the programmer, to free up the resources when they are no longer required.

Codewalk Quicklet

The code for the codewalk quicklet exercise in this chapter is presented in Listing 4-13.

Listing 4-13. *CodeWalkThree.java*

```
1.  class CodeWalkThree {
2.     public static void main(String [] args) {
3.           CodeWalkThree cw = new CodeWalkThree();
4.           CodeWalkThree cw2 = new CodeWalkThree();
5.           System.out.print(cw == cw2);
6.           cw2 = operate(cw,cw2);
7.           System.out.print(" " + (cw == cw2));
8.       }
9.       static CodeWalkThree operate(CodeWalkThree cw1, CodeWalkThree cw2) {
10.            CodeWalkThree cw3 = cw1;
11.             cw1 = cw2;
12.              return cw3;
13.      }
14.   }
```

To get you started on this exercise, following are some hints:

- *Input*: Lines 3, 4, and 10
- *Operations*: Lines 5, 6, 7, and 11
- *Output*: Lines 5 and 7
- *Rules*: The effect of a method call on object references

Looking at the code from this angle, see if you can handle the last question in the "Review Questions" section based on this code.

The three most important takeaways from this chapter are the following:

- The full path name to a class is constructed from the `classpath` followed by the package name followed by the class name.
- You specify access modifiers in order to strike a balance between security and access.
- You can make an object eligible for garbage collection by making sure that no reference is pointing to it. But you have no control over when the garbage collector will collect an eligible object.

Summary

A Java application is composed of classes, which are written in source files with the `.java` extension. The related classes and interfaces can be bundled together into a package by using the `package` keyword in the source files in which classes are written. A package can be imported into another source file by using the keyword `import`. In a source file, the package declaration, the `import` statement, and the class definition must appear in this order. The compiler generates a `.class` file corresponding to each class in the source file, and the name of the `.class` file matches the corresponding class. The compiler and the interpreter make the full path of a directory in which a class file exists by appending the package name to the `classpath`.

From where a class or a class member can be accessed is determined by its access modifier. A class or a class member declared `public` can be accessed from anywhere inside the application. A private member of a class may only be accessed from the code inside the same class in which that member is declared. It can be accessed neither from any other class nor from a subclass of the class in which it is declared. For this reason, a top-level class can never be declared `private`, but its members can be. A class member declared `protected` can be accessed from any other class in the same package as the accessed class, or from a subclass of the accessed class in any package. A class or a class member with a default access (which means no access modifier is specified) can be accessed only if the accessing class belongs to the same package as the accessed class.

When you pass a variable in a method call, only a copy of the variable is passed and not the original variable, regardless of whether the variable is a primitive variable or a reference variable. Therefore, any change in the passed variable in the called method does not affect the variable in the calling method. However, in case of a reference variable, the called method can use the passed variable to change the properties of the object to which the passed variables refers.

Memory management in Java is automatic. The Java garbage collector automatically frees up the memory from objects that are no longer being used. In order to get an object garbage collected, you must make it eligible by making sure no reference is pointing to it. The garbage collector calls the `finalize()` method on the object before freeing up its memory. The `finalize()` method is inherited by your class from the `Object` class, but you can override it to clean up after the object (for example, to release resources). However, there is no way to guarantee when the garbage collector will be invoked.

In this chapter, we explored the fundamentals of Java, which is an object-oriented language. In the next chapter, we explore some salient features of object-oriented programming implemented in Java.

EXAM'S EYE VIEW

Comprehend

- Only eligible objects will be removed by the garbage collector. An object is eligible for garbage collection if no reference points to it.
- A member with a default access can be accessed only if the accessing class belongs to the same package, whereas a member with the `protected` modifier can be accessed not only from the same package but also from a different package if the accessing class is a subclass of the accessed class.
- It is legal to use the `final` object reference to change the value of a non-final variable of the object.
- An `abstract` class can be extended but cannot be instantiated. You must implement all of its `abstract` methods in a subclass before you can instantiate the subclass.
- A `final` class can be instantiated but cannot be extended.

Look Out

- The name of a `.java` file must match the name of a class in the file. If the file has a `public` class, the file name must match the name of the `public` class.
- In a `.java` file, the package declaration, the `import` statement, and the class definition must appear in this order.
- Variables inside a method cannot have explicit access modifiers such as `public`, `private`, and `protected` assigned to them.
- A top-level class cannot be declared `private`; it can only be `public` or default (that is, no access modifier specified).
- There is no such modifier named `default`, but the default access is assumed if no modifier is specified.
- A `static` method of a class cannot access non-static members of the class.
- It is possible to make an object ineligible for garbage collection by creating references to it in its `finalize()` method.
- The called method can change the properties of the object to which a passed reference to the method points. However, any change in the reference in the called method will not affect the reference in the calling method.

Memorize

- You cannot change the value of a `final` variable, you cannot extend the class that is declared `final`, and you cannot override a `final` method.
- A variable cannot be declared `abstract`.
- While the `native` modifier only applies to methods, the `transient` and `volatile` modifiers only apply to instance variables.
- Executing `System.gc()` does not necessarily mean that memory will be made free; that is, you cannot force the garbage collection.
- In a method call, it is always the copy of a variable that is passed, regardless of whether the variable is a primitive variable or a reference variable.

Review Questions

1. Which of the following declarations will result in compiler error? (Choose all that apply.)
   A. `abstract final class MyClass {};`
   B. `abstract int i;`
   C. `default class MyClass {};`
   D. `native myMethod();`

2. Which of the following statements is false?
   A. An `abstract` class must have at least one `abstract` method.
   B. An `abstract` class cannot have a `finalize()` method.
   C. A `final` class cannot have `abstract` methods.
   D. A top-level class cannot be declared `private`.

3. Assume that a variable exists in a class and that the variable must not be copied into a file when the object corresponding to this class is serialized. What modifier should be used in the declaration of this variable?
   A. `private`
   B. `protected`
   C. `public`
   D. `transient`
   E. `native`

4. Which of the following statements is true about the `static` modifier?
   A. A `static` variable cannot change its value.
   B. A `static` method cannot be overridden to be non-static.
   C. A `static` method is often written in a non-Java language and exists outside of JVM.
   D. The `static` code lies outside of any class.

5. Consider the following code fragment:

```
1. class MySuperClass {
2.   public void message() {
3.        System.out.println("From the super class!");
4.   }
5. }
6. public class MySubClass extends MySuperClass {
7.    void message() {
8.      System.out.println("From the subclass!");
9.    }
10.  public static void main(String args[]) {
11.    MySubClass mysub = new MySubClass();
12.    mysub.message();
13.  }
14. }
```

Which of the following statement is true about this code?

A. The code would compile and execute, and generate the output: `From the subclass!`.

B. The code would compile and execute, and generate the output: `From the super class!`.

C. Line 7 would generate a compiler error.

D. Line 11 would generate a compiler error.

6. Consider the following code fragment:

```
1. class MySuperClass {
2.   static void message() {
3.        System.out.println("From the super class!");
4.    }
5.  }
6. public class MySubClass extends MySuperClass {
7.    void message() {
8.      System.out.println("From the subclass!");
9.    }
10.   public static void main(String args[]) {
11.      MySubClass mysub = new MySubClass();
12.      mysub.message();
13.   }
14. }
```

Which of the following modifiers placed in the beginning of line 7 will make the code compile and execute without error?

A. `static`

B. `public`

C. `protected`

D. `transient`

7. Which of the following statements is true?

A. The `final` variable may only be used with a variable or a method.

B. The `final` variable may not be copied to a file during object serialization.

C. The `final` method may not be overridden.

D. The class that has a `final` method may not be extended.

8. Consider the following code fragment:

```
1.   class MyClass {
2.       public void message (int i) {
3.         public   int j= i;
4.         System.out.println("Value of  j: " + j);
5.       }
6.       public static void main(String[] args) {
7.          MyClass ma = new MyClass();
8.          ma.message(15);
9.       }
10. }
```

Which of the following statements is true about this code?

A. The code will compile and execute fine, and the output will be `Value of j: 15`.

B. Line 2 will generate a compiler error.

C. The code will compile but give an error at execution time.

Consider the following code fragment for questions 9 and 10:

```
package robots;
public class FunnyRobot {
protected void dance () {
    System.out.println("The funny robot is dancing!");
}
void shyAway () {
  System.out.println("The funny robot is shying away!");
}
private void freeze () {
  System.out.println("The robot has come to a stop!");
}
}
```

9. Consider the following code fragment:

```
1. package  RobotDrivers ;
2. import robots.*;
3.  public class RobotPlayer extends FunnyRobot {
4.         static int i =5;
5.    public static void main(String[] args){
6.                    i = 6;
7.         RobotPlayer rp = new RobotPlayer();
8.         rp.dance();
9.   }
10. }
```

Which of the following statements is true about this code fragment?

A. The code will compile and execute correctly, and generate the output: `The funny robot is dancing!`.

B. There would be a compiler error at line 7 because the method `dance()` is `protected` and the classes `RobotPlayer` and `FunnyRobot` are in different packages.

C. There would be a compiler error at line 6.

D. The code will compile, but will generate a runtime exception.

10. Consider the following code fragment:

```
1. package  RobotDrivers ;
2. import robots.*;
3.  public class RobotPlayer{
4.         static int i =5;
5.    public static void main(String[] args){
6.         i = 6;
7.         FunnyRobot fr = new FunnyRobot();
8.         fr.dance();
9.   }
10. }
```

Which of the following statements is true about this code fragment?

A. The code will compile and execute correctly, and generate the output: `The funny robot is dancing!`.

B. There would be a compiler error at line 7 because the method `dance()` is `protected` and the classes `RobotPlayer` and `FunnyRobot` are in different packages.

C. There would be a compiler error at line 6.

D. The code will compile, but will generate a runtime exception.

11. Consider the code in Listing 4-13. What is the result?

A. `false false`

B. `true true`

C. `false true`

D. `true false`

E. Compilation fails.