SAMPLE CHAPTER

OCA Java SE 7

Programmer I

CERTIFICATION GUIDE

Prepare for the 1Z0-803 exam

Mala Gupta

FOREWORD BY **Jeanne Boyarsky**

MANNING

OCA Java SE 7 Programmer I Certification Guide

by Mala Gupta

Chapter 1

brief contents

Introduction 1

1 ■ Java basics 13

2 ■ Working with Java data types 69

3 ■ Methods and encapsulation 110

4 ■ String, StringBuilder, Arrays, and ArrayList 174

5 ■ Flow control 243

6 ■ Working with inheritance 295

7 ■ Exception handling 348

8 ■ Full mock exam 405

1 Java basics

Exam objectives covered in this chapter	What you need to know
[1.2] Define the structure of a Java class.	Structure of a Java class, with its components: package and import statements, class declarations, comments, variables, and methods. Difference between the components of a Java class and that of a Java source code file.
[1.3] Create executable Java applications with a `main` method.	The right method signature for the `main` method to create an executable Java application. The arguments that are passed to the `main` method.
[1.4] Import other Java packages to make them accessible in your code.	Understand packages and import statements. Get the right syntax and semantics to import classes from packages and interfaces in your own classes.
[6.6] Apply access modifiers.	Application of access modifiers (`public`, `protected`, default, and `private`) to a class and its members. Determine the accessibility of code with these modifiers.
[7.6] Use `abstract` classes and interfaces.	The implication of defining classes, interfaces, and methods as `abstract` entities.
[6.2] Apply the `static` keyword to methods and fields.	The implication of defining fields and methods as `static` members.

Imagine you've set up a new IT organization that works with multiple developers. To ensure a smooth and efficient workflow, you'll define a structure for your organization and a set of departments with separate assigned responsibilities. These departments will interact with each other whenever required. Also, depending on

confidentiality requirements, your organization's data will be available to employees on an as-needed basis, or you may assign special privileges to only some employees of the organization. This is an example of how organizations work with a well-defined structure and a set of rules to deliver the best results.

Similarly, Java has organized its workflow. The organization's structure and components can be compared with Java's class structure and components, and the organization's departments can be compared with Java packages. Restricting access to all data in the organization can be compared to Java's access modifiers. An organization's special privileges can be compared to nonaccess modifiers in Java.

In the OCA Java SE 7 Programmer I exam, you'll be asked questions on the structure of a Java class, packages, importing classes, and applying access and nonaccess modifiers. Given that information, this chapter will cover the following:

- Understanding the structure and components of a Java class
- Understanding executable Java applications
- Understanding Java packages
- Importing Java packages into your code
- Applying access and nonaccess modifiers

1.1 The structure of a Java class and source code file

[1.2] Define the structure of a Java class

NOTE When you see a certification objective callout such as the preceding one, it means that in this section, we'll cover this objective. The same objective may be covered in more than one section in this chapter or in other chapters.

This section covers the structure and components of both a Java source code file (.java file) and a Java class (defined using the keyword `class`). It also covers the differences between a Java source code file and a Java class.

First things first. Start your exam preparation with a clear understanding of what is required from you in the certification exam. For example, try to answer the following query from a certification aspirant: "I come across the term 'class' with different meanings—class `Person`, the Java source code file—Person.java, and Java bytecode stored in Person.class. Which of these structures is on the exam?" To answer this question, take a look at figure 1.1, which includes the class `Person`, the files Person.java and Person.class, and the relationship between them.

As you can see in figure 1.1, a person can be defined as a class `Person`. This class should reside in a Java source code file (Person.java). Using this Java source code file, the Java compiler (javac.exe on Windows or javac on Mac OS X/Linux/UNIX) generates bytecode (compiled code for the Java Virtual Machine) and stores it in Person.class. The scope of this exam objective is limited to Java classes (class `Person`) and Java source code files (Person.java).

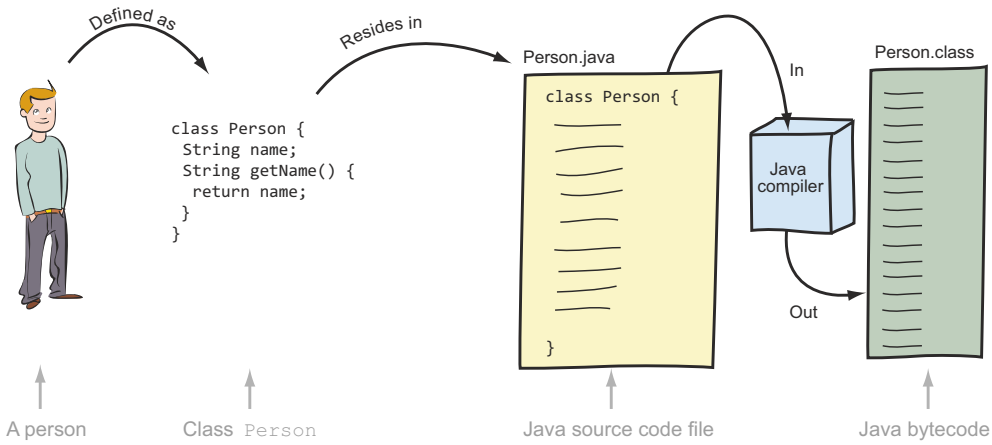


Figure 1.1 Relationship between the class `Person`, the files Person.java and Person.class, and how one transforms into another

1.1.1 Structure of a Java class

The OCA Java SE 7 Programmer I exam will question you on your understanding of the structure and components of a Java class defined using the keyword `class`. A class can define multiple components. All the Java components that you've heard of can be defined within a Java class. Figure 1.2 defines the components and structure of a Java class.

Here's a quick list of the components of a class (the ones that are on this exam), which we'll discuss in detail in this section:

- The `package` statement
- The `import` statement
- Comments
- Class declarations and definitions
- Variables
- Methods
- Constructors

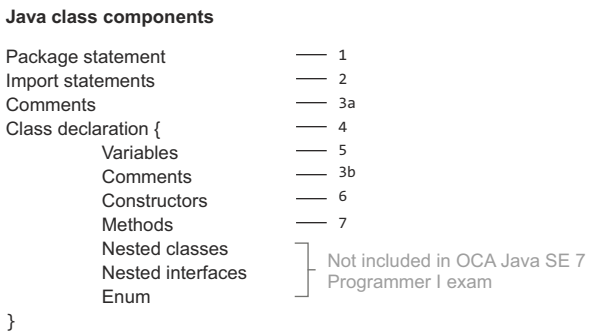


Figure 1.2 Components of a Java class

PACKAGE STATEMENT

All Java classes are part of a package. A Java class can be explicitly defined in a named package; otherwise it becomes part of a *default* package, which doesn't have a name.

A `package` statement is used to explicitly define which package a class is in. If a class includes a `package` statement, it must be the first statement in the class definition:

```
package certification;
class Course {

}
```

The rest of the code for class Course

The package statement should be the first statement in a class

NOTE Packages are covered in detail in section 1.3 of this chapter.

The `package` statement cannot appear within a class declaration or after the class declaration. The following code will fail to compile:

```
class Course {

}
package certification;
```

The rest of the code for class Course

If you place the package statement after the class definition, the code won't compile

The following code will also fail to compile, because it places the `package` statement within the class definition:

```
class Course {
package com.cert;
}
```

A package statement can't be placed within the curly braces that mark the start and end of a class definition

Also, if present, the `package` statement must appear exactly once in a class. The following code won't compile:

```
package com.cert;
package com.exams;
class Course {
}
```

A class can't define multiple package statements

IMPORT STATEMENT

Classes and interfaces in the same package can use each other without prefixing their names with the package name. But to use a class or an interface from another package, you must use its fully qualified name. Because this can be tedious and can make your code difficult to read, you can use the `import` statement to use the simple name of a class or interface in your code.

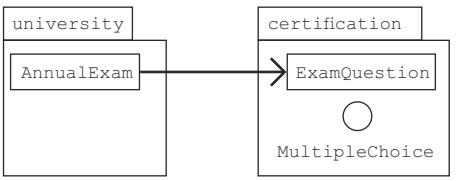


Figure 1.3 UML representation of the relationship between class `AnnualExam` and `ExamQuestion`

Let's look at this using an example class, `AnnualExam`, which is defined in the package `university`. Class `AnnualExam` is associated with the class `certification.ExamQuestion`, as shown using the Unified Modeling Language (UML) in figure 1.3.

Here's the code for class `AnnualExam`:

```
package university;
import certification.ExamQuestion;

class AnnualExam {
    ExamQuestion eq;
}
```

Define a variable of ExamQuestion

Note that the `import` statement follows the `package` statement but precedes the `class` declaration. What happens if the class `AnnualExam` isn't defined in a package? Will there be any change in the code if the class `AnnualExam` and `ExamQuestion` are related, as depicted in figure 1.4?

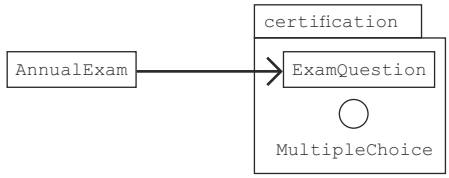


Figure 1.4 A UML representation of the relationship between the unpackaged class `AnnualExam` and `ExamQuestion`

In this case, the class `AnnualExam` isn't part of an explicit package, but the class `ExamQuestion` is part of package `certification`. Here's the code for class `AnnualExam`:

```
import certification.ExamQuestion;
class AnnualExam {
    ExamQuestion eq;
}
```

Define a variable of ExamQuestion

As you can see in the previous example code, the class `AnnualExam` doesn't define the `package` statement, but it defines the `import` statement to import the class `certification.ExamQuestion`.

If a `package` statement is present in a class, the `import` statement must follow the `package` statement. It's important to maintain the order of the occurrence of the `package` and `import` statements. Reversing this order will result in your code failing to compile:

```
import certification.ExamQuestion;
package university;

class AnnualExam {
    ExamQuestion eq;
}
```

The code won't compile because an import statement can't be placed before a package statement

We'll discuss `import` statements in detail in section 1.3 of this chapter.

COMMENTS

You can also add comments to your Java code. Comments can appear at multiple places in a class. A comment can appear before and after a `package` statement, before and after the class definition, before and within and after a method definition. Comments come in two flavors: multiline comments and end-of-line comments.

Multiline comments span multiple lines of code. They start with `/*` and end with `*/`. Here's an example:

```
class MyClass {
    /*
      comments that span multiple
      lines of code
    */
}
```

Multiline comments start with /* and end with */

Multiline comments can contain any special characters (including Unicode characters). Here's an example:

```
class MyClass {
    /*
       Multi-line comments with
       special characters &%^*{}|\|:;"'
       ?/>.<,!@#$%^&*()
    */
}
```

Multiline comment with special characters in it

Most of the time, when you see a multiline comment in a Java source code file (.java file), you'll notice that it uses an asterisk (*) to start the comment in the next line. Please note that this isn't required—it's done more for aesthetic reasons. Here's an example:

```
class MyClass {
    /*
     * comments that span multiple
     * lines of code
     */
}
```

Multiline comments that start with * on a new line—don't they look well organized? The usage of * isn't mandatory; it's done for aesthetic reasons.

End-of-line comments start with // and, as evident by their name, they are placed at the end of a line of code. The text between // and the end of the line is treated as a comment, which you would normally use to briefly describe the line of code. Here's an example:

Brief comment to describe variable fName

```
class Person {
    String fName;    // variable to store Person's first name
    String id;       // a 6 letter id generated by the database
}
```

Brief comment to describe variable id

In the earlier section on the package statement, you read that a package statement, if present, should be the first line of code in a class. The only exception to this rule is the presence of comments. A comment can precede a package statement. The following code defines a package statement, with multiline and end-of-line comments:

```
/**
 * @author MGupta        // first name initial + last name
 * @version 0.1
 *
 * Class to store the details of a monument
 */
package uni;            // package uni
class Monument {
    int startYear;
```

1 End-of-line comment within a multiline comment

2 End-of-line comment

```
    String builtBy;     // individual/ architect      <-- 3 End-of-line comment
}
// another comment                                   <-- 4 End-of-line comment at the beginning of a line
```

Line ❶ defines an end-of-line code comment within multiline code. This is acceptable. The end-of-line code comment is treated as part of the multiline comment, not as a separate end-of-line comment. Lines ❷ and ❸ define end-of-line code comments. Line ❹ defines an end-of-line code comment at the start of a line, after the class definition.

The multiline comment is placed before the `package` statement, which is acceptable because comments can appear anywhere in your code.

CLASS DECLARATION

The class declaration marks the start of a class. It can be as simple as the keyword `class` followed by the name of a class:

```
class Person {          <-- Simplest class declaration: keyword class followed by the class name
//..                    <-- A class can define a lot of things here, but we don't need these details to show the class declaration
//..
}
```

Time to get more details. The declaration of a class is composed of the following parts:

- Access modifiers
- Nonaccess modifiers
- Class name
- Name of the base class, if the class is extending another class
- All implemented interfaces, if the class is implementing any interfaces
- Class body (class fields, methods, constructors), included within a pair of curly braces, {}

Don't worry if you don't understand this material at this point. I'll cover these details as we move through the exam preparation.

Let's look at the components of a class declaration using an example:

```
public final class Runner extends Person implements Athlete {}
```

The components of the preceding class can be pictorially depicted, as shown in figure 1.5. The following list summarizes the optional and compulsory components.

Compulsory	Optional
Keyword `class`	Access modifier, such as `public`
Name of the class	Nonaccess modifier, such as `final`
Class body, marked by the opening and closing curly braces, {}	Keyword `extends` together with the name of the base class
	Keyword `implements` together with the name of the interfaces being implemented

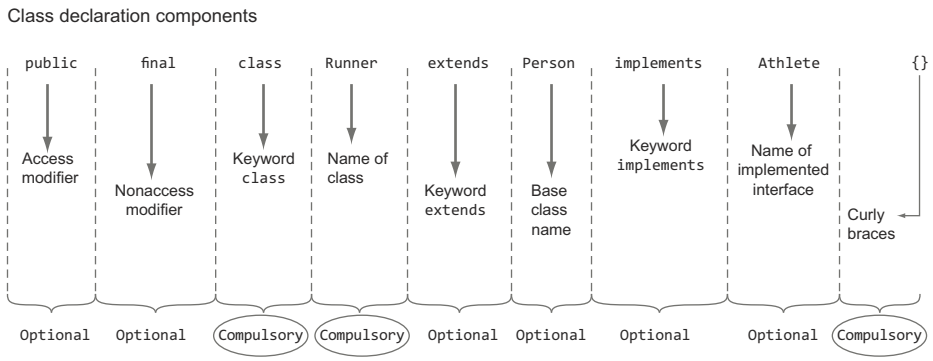


Figure 1.5 Components of a class declaration

We'll discuss the access and nonaccess modifiers in detail in sections 1.4 and 1.5 in this chapter.

CLASS DEFINITION

A class is a design used to specify the properties and behavior of an object. The properties of an object are implemented using *variables*, and the behavior is implemented using *methods*.

For example, consider a class as being like the design or specification of a mobile phone, and a mobile phone as being an object of that design. The same design can be used to create multiple mobile phones, just as the Java Virtual Machine (JVM) uses a class to create its objects. You can also consider a class as being like a mold that you can use to create meaningful and useful objects. A class is a design from which an object can be created.

Let's define a simple class to represent a mobile phone:

```
class Phone {
    String model;
    String company;
    Phone(String model) {
        this.model = model;
    }
    double weight;
    void makeCall(String number) {
        // code
    }
    void receiveCall() {
        // code
    }
}
```

Points to remember:

- A class name starts with the keyword `class`. Watch out for the case of the keyword `class`. Java is case sEnSiTivE. `class` (lowercase *c*) isn't the same as `Class` (uppercase *C*). You can't use the word `Class` (capital *C*) to define a class.
- The state of a class is defined using attributes or instance variables.

- The behavior is defined using methods. The methods will include the argument list (if any). Don't worry if you don't understand these methods. The methods are covered in detail later in this chapter.
- A class definition may also include comments and constructors.

NOTE A class is a design from which an object can be created.

VARIABLES

Revisit the previous example. Because the variables `model`, `company`, and `weight` are used to store the state of an object (also called an *instance*), they are called *instance variables* or *instance attributes*. Each object has its own copy of the instance variables. If you change the value of an instance variable for an object, the value for the same named instance variable won't change for another object. The instance variables are defined within a class but outside all methods in a class.

A single copy of a *class variable* or `static` variable is shared by all the objects of a class. The `static` variables are covered in section 1.5.3 with a detailed discussion of the nonaccess modifier `static`.

METHODS

Again, revisit the previous example. The methods `makeCall` and `receiveCall` are instance methods, which are generally used to manipulate the instance variables.

A *class method* or *static method* is used to work with the `static` variables, as discussed in detail in section 1.5.3.

CONSTRUCTORS

Class `Phone` in the previous example defines a single constructor. A class constructor is used to create and initialize the objects of a class. A class can define multiple constructors that accept different sets of method parameters.

1.1.2 Structure and components of a Java source code file

A Java source code file is used to define classes and interfaces. All your Java code should be defined in Java source code files (text files whose names end with .java). The exam covers the following aspects of the structure of a Java source code file:

- Definition of a class and an interface in a Java source code file
- Definition of single or multiple classes and interfaces within the same Java source code file
- Application of `import` and `package` statements to all the classes in a Java source code file

We've already covered the detailed structure and definition of classes in section 1.1.1. Let's get started with the definition of an interface.

DEFINITION OF INTERFACES IN A JAVA SOURCE CODE FILE

An interface is a grouping of related methods and constants, but the methods in an interface cannot define any implementation. An interface specifies a contract for the classes to implement.

Here's a quick example to help you to understand the essence of interfaces. No matter which brand of television each one of us has, every television provides the common functionality of changing the channel and adjusting the volume. You can compare the controls of a television set to an interface, and the design of a television set to a class that implements the interface controls.

Let's define this interface:

```
interface Controls {
    void changeChannel(int channelNumber);
    void increaseVolume();
    void decreaseVolume();
}
```

The definition of an interface starts with the keyword `interface`. An interface can define constants and methods. Remember, Java is case-sensitive, so you can't use the word `Interface` (with a capital *I*) to define an interface.

DEFINITION OF SINGLE AND MULTIPLE CLASSES IN A SINGLE JAVA SOURCE CODE FILE

You can define either a single class or an interface in a single Java source code file, or many such files. Let's start with a simple example: a Java source code file called SingleClass.java that defines a single class `SingleClass`:

```
class SingleClass {
    //.. we are not detailing this part
}
```

Contents of Java source code file SingleClass.java

Here's an example of a Java source code file, Multiple1.java, that defines multiple interfaces:

```
interface Printable {
    //.. we are not detailing this part
}
interface Movable {
    //.. we are not detailing this part
}
```

Contents of Java source code file Multiple1.java

You can also define a combination of classes and interfaces in the same Java source code file. Here's an example:

```
interface Printable {
    //.. we are not detailing this part
}
class MyClass {
    //.. we are not detailing this part
}
interface Movable {
    //.. we are not detailing this part
}
class Car {
    //.. we are not detailing this part
}
```

Contents of Java source code file Multiple2.java

There is no required order for the multiple classes or interfaces that can be defined in a single Java source code file.

EXAM TIP The classes and interfaces can be defined in any order of occurrence in a Java source code file.

If you define a `public` class or an interface in a class, its name should match the name of the Java source code file. Also, a source code file can't define more than one `public` class or interface. If you try to do so, your code won't compile, which leads to a small hands-on exercise for you that I call *Twist in the Tale*, as mentioned in the Preface. The answers to all these exercises are provided in the appendix.

About the Twist in the Tale exercises

For these exercises, I've tried to use modified code from the examples already covered in the chapter. The *Twist in the Tale* title refers to modified or tweaked code.

These exercises will help you understand how even small code modifications can change the behavior of your code. They should also encourage you to carefully examine all of the code in the exam. The reason for these exercises is that in the exam, you may be asked more than one question that seems to require the same answer. But on closer inspection, you'll realize that the questions differ slightly, and this will change the behavior of the code and the correct answer option!

Twist in the Tale 1.1

Modify the contents of the Java source code file Multiple.java, and define a public interface in it. Execute the code and see how it affects your code.

Question: Examine the following content of Java source code file Multiple.java and select the correct answers:

```
// Contents of Multiple.java
public interface Printable {
    //.. we are not detailing this part
}
interface Movable {
    //.. we are not detailing this part
}
```

Options:

a A Java source code file cannot define multiple interfaces.
b A Java source code file can only define multiple classes.
c A Java source code file can define multiple interfaces and classes.
d The previous class will fail to compile.

If you need help getting your system set up to write Java, refer to Oracle's "Getting Started" tutorial, http://docs.oracle.com/javase/tutorial/getStarted/.

Twist in the Tale 1.2

Question: Examine the content of the following Java source code file, Multiple2.java, and select the correct option.

```
// contents of Multiple2.java
interface Printable {
    //.. we are not detailing this part
}
class MyClass {
    //.. we are not detailing this part
}
interface Movable {
    //.. we are not detailing this part
}
public class Car {
    //.. we are not detailing this part
}
public interface Multiple2 {}
```

Options:

a The code fails to compile.
b The code compiles successfully.
c Removing the definition of class `Car` will compile the code.
d Changing class `Car` to a non-`public` class will compile the code.
e Changing class Multiple2 to a non-`public` class will compile the code.

APPLICATION OF PACKAGE AND IMPORT STATEMENTS IN JAVA SOURCE CODE FILES

In the previous section, I mentioned that you can define multiple classes and interfaces in the same Java source code file. When you use a `package` or `import` statement within such Java files, both the `package` and `import` statements apply to all of the classes and interfaces defined in that source code file.

For example, if you include a `package` and an `import` statement in Java source code file Multiple.java (as in the following code), `Car`, `Movable`, and `Printable` will be become part of the same package `com.manning.code`:

```
// contents of Multiple.java
package com.manning.code;
import com.manning.*;

interface Printable {}
interface Movable {}
class Car {}
```

Printable, Movable, and Car are part of package com.manning.code

All classes and interfaces defined in package com.manning are accessible to Printable, Movable, and Car

EXAM TIP Classes and interfaces defined in the same Java source code file *can't* be defined in separate packages. Classes and interfaces imported using the `import` statement are available to all the classes and interfaces defined in the same Java source code file.

In the next section, you'll create executable Java applications—classes that are used to define an entry point of execution for a Java application.

1.2 Executable Java applications

[1.3] Create executable Java applications with a main method

The OCA Java SE 7 Programmer I exam requires that you understand the meaning of an executable Java application and its requirements, that is, what makes a regular Java class an executable Java class.

1.2.1 Executable Java classes versus nonexecutable Java classes

"Doesn't the Java Virtual Machine execute all the Java classes when they are used? If so, what is a nonexecutable Java class?"

An executable Java class is a class which, when handed over to the JVM, starts its execution at a particular point in the class—the `main` method, defined in the class. The JVM starts executing the code that is defined in the `main` method. You cannot hand over a nonexecutable Java class to the JVM and ask it to start executing the class. In this case, the JVM won't know how to execute it because no entry point is marked, for the JVM, in a nonexecutable class.

Typically, an application consists of a number of classes and interfaces that are defined in multiple Java source code files. Of all these files, a programmer designates one of the classes as an executable class. The programmer can define the steps that the JVM should execute as soon as it launches the application. For example, a programmer can define an executable Java class that includes code to display the appropriate GUI window to a user and to open a database connection.

In figure 1.6, the classes `Window`, `UserData`, `ServerConnection`, and `UserPreferences` don't define a `main` method. Class `LaunchApplication` defines a `main` method and is an executable class.

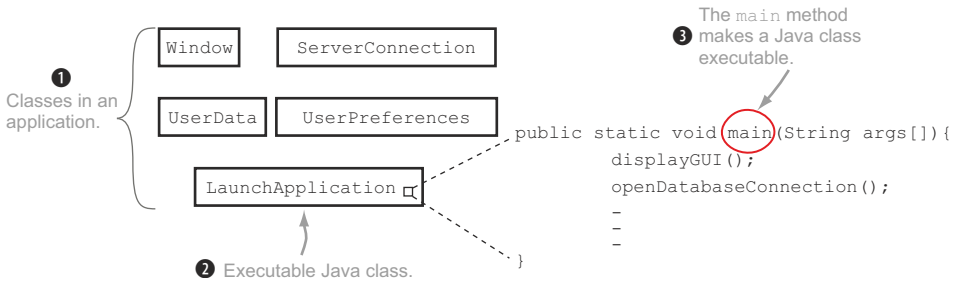


Figure 1.6 Class `LaunchApplication` is an executable Java class, but the rest of the classes—`Window`, `UserData`, `ServerConnection`, and `UserPreferences`—aren't.

1.2.2 Main method

The first requirement in creating an executable Java application is to create a class with a method whose signature (name and method arguments) match the `main` method, defined as follows:

```
public class HelloExam {
    public static void main(String args[]) {
        System.out.println("Hello exam");
    }
}
```

This `main` method should comply with the following rules:

- The method must be marked as a `public` method.
- The method must be marked as a `static` method.
- The name of the method must be `main`.
- The return type of this method must be `void`.
- The method must accept a method argument of a `String` array or a variable argument of type `String`.

Figure 1.7 illustrates the previous code and its related set of rules.

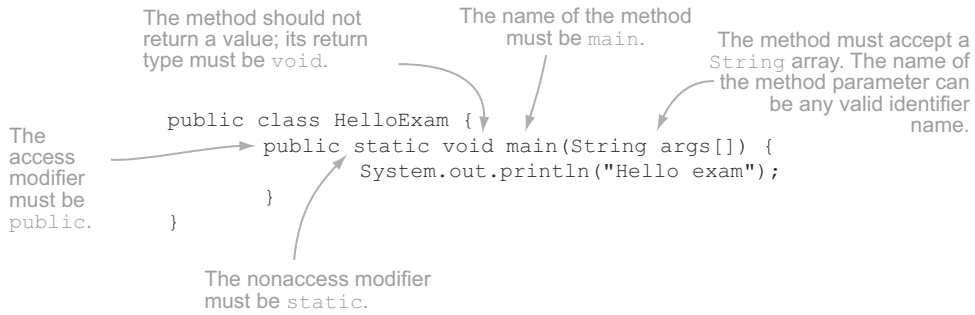


Figure 1.7 Ingredients of a correct `main` method

It's valid to define the method parameter passed to the `main` method as a variable argument (*varargs*) of type `String`:

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

It is valid to define args as a variable argument

To define a variable argument variable, the ellipsis (. . .) should follow the type of the variable and not the variable itself (a mistake made by lot of new programmers):

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

This won't compile. Ellipses should follow the data type, String.

As mentioned previously, the name of the `String` array passed to the `main` method need not be `args` to qualify it as the correct `main` method. Thus, the following examples are also correct definitions of the `main` method:

```
public static void main(String[] arguments)
public static void main(String[] HelloWorld)
```

The names of the method arguments are arguments and HelloWorld, which is acceptable

To define an array, the square brackets, `[]`, can follow either the variable name or its type. The following is a correct method declaration of the `main` method:

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

The square brackets, [], can follow either the variable name or its type

It's interesting to note that the placement of the keywords `public` and `static` can be interchanged, which means that the following are both correct method declarations of the `main` method:

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

The placement of the keywords public and static is interchangeable

On execution, the code shown in figure 1.7 outputs the following:

```
Hello exam
```

Almost all Java developers work with an Integrated Development Environment (IDE). The OCA Java SE 7 Programmer I exam, however, expects you to understand how to execute a Java application, or an executable Java class, using the command prompt. If you need help getting your system set up to compile or execute Java applications using the command prompt, refer to Oracle's detailed instructions at http://docs.oracle.com/javase/tutorial/getStarted/cupojava/index.html.

To execute the code shown in figure 1.7, issue the command `java HelloExam`, as shown in figure 1.8.

We discussed how the `main` method accepts an array of `String` as the method parameter. But how and where do you pass the array to the `main` method? Let's modify the previous code to access and output values from this array. Here's the relevant code:

```
public class HelloExamWithParameters {
    public static void main(String args[]) {
        System.out.println(args[0]);
        System.out.println(args[1]);
    }
}
```

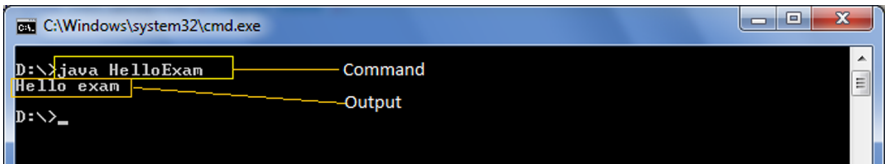


Figure 1.8 Using a command prompt to execute a Java application

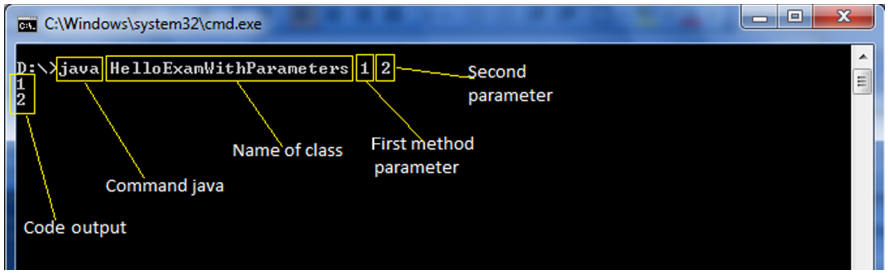


Figure 1.9 Passing command parameters to a `main` method

Execute the class in the preceding code at a command prompt, as shown in figure 1.9.

As you can see from the output shown in figure 1.9, the keyword `java` and the name of the class aren't passed on as command parameters to the `main` method. The OCA Java SE 7 Programmer I exam will test you on your knowledge of whether the keyword `java` and the class name are passed on to the `main` method.

EXAM TIP The method parameters that are passed on to the `main` method are also called command-line parameters or command-line values. As the name implies, these values are passed on to a method from the command line.

If you weren't able to follow the code with respect to the arrays and class `String`, don't worry; we'll cover the class `String` and arrays in detail in chapter 4.

Here's the next Twist in the Tale exercise for you. In this exercise, and in the rest of the book, you'll see the names Shreya, Harry, Paul, and Selvan, who are hypothetical programmers also studying for this certification exam. The answer is provided in the appendix.

Twist in the Tale 1.3

One of the programmers, Harry, executed a program that gave the output "java one". Now he's trying to figure out which of the following classes outputs these results. Given that he executed the class using the command `java EJava java one one`, can you help him figure out the correct option(s)?

a
```
class EJava {
    public static void main(String sun[]) {
        System.out.println(sun[0] + " " + sun[2]);
    }
}
```

b
```
class EJava {
    static public void main(String phone[]) {
        System.out.println(phone[0] + " " + phone[1]);
    }
}
```

c
```
class EJava {
    static public void main(String[] arguments[]) {
```

```
            System.out.println(arguments[0] + " " + arguments[1]);
        }
    }
d   class EJava {
        static void public main(String args[]) {
            System.out.println(args[0] + " " + args[1]);
        }
    }
```

Confusion with command-line parameters

Programming languages like C pass on the name of a class as a command-line argument to the `main` method. Java doesn't do so. This is a simple but important point.

1.3 Java packages

[1.4] Import other Java packages to make them accessible in your code

In this section, you'll learn what Java packages are and how to create them. You'll use the `import` statement, which enables you to use simple names for classes and interfaces defined in separate packages.

1.3.1 The need for packages

Why do you think we need packages? First, answer this question: do you remember having known more than one Amit, Paul, Anu, or John to date? Harry knows more than one Paul (six, to be precise), whom he categorizes as managers, friends, and cousins. These are subcategorized by their location and relation, as shown in figure 1.10.

Similarly, you can use packages to group together a related set of classes and interfaces (I will not discuss enums here because they aren't covered on this exam). Packages also provide access protection and namespace management. You can create separate packages to define classes for separate projects, such as android games and online health-care systems. Further, you can create subpackages within these packages, such as separate subpackages for GUIs, database access, networking, and so on.

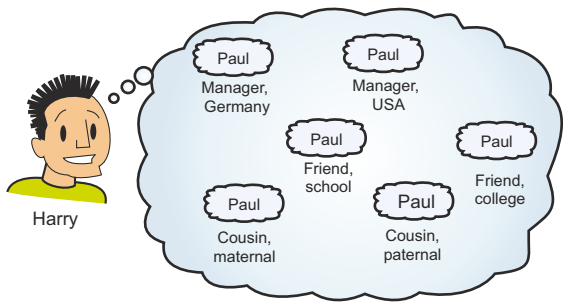


Figure 1.10 Harry knows six Pauls!

PRACTICAL TIP In real-life projects, you will never work with an unpackaged class or interface. Almost all organizations that develop software have strict package-naming rules, which are often documented.

The OCA Java SE 7 Programmer I exam covers importing packaged classes into other classes. But after 12 years of experience, I've learned that before starting to *import* other classes into your own code, it's important to understand what the packaged classes are, how packaged and nonpackaged classes differ, and why you need to import the packaged classes.

Packaged classes are part of a named package—a namespace—and they're defined as being part of a package by including a `package` statement in a class. All classes and interfaces are packaged. If you don't include an explicit `package` statement in a class or an interface, it's part of a *default* package.

1.3.2 Defining classes in a package using the package statement

You can define which classes and interfaces are in a package by using the `package` statement as the first statement in your class or interface. Here's an example:

```
package certification;
class ExamQuestion {
    //..code
}
```

Variables and methods

The class in the previous code defines an `ExamQuestion` class in the `certification` package. You can define an interface, `MultipleChoice`, in a similar manner:

```
package certification;
interface MultipleChoice {
    void choice1();
    void choice2();
}
```

Figure 1.11 shows the UML representation of the package `certification`, with the class `ExamQuestion` and the interface `MultipleChoice`:

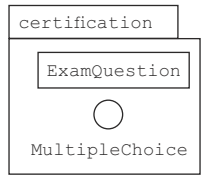


Figure 1.11 A UML representation of the package `certification`, class `ExamQuestion`, and interface `MultipleChoice`

The name of the package in the previous examples is `certification`. You may use such names for small projects that contain only a few classes and interfaces, but it's common for organizations to use subpackages to define *all* their classes. For example, if folks at Oracle define a class to store exam questions for a Java Associate exam, they might use the package name `com.oracle.javacert.associate`. Figure 1.12 shows its UML representation, together with the corresponding class definition:

```
package com.oracle.javacert.associate;
class ExamQuestion {
       // variables and methods
}
```

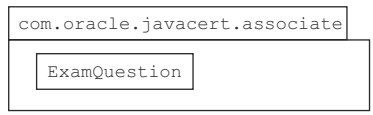


Figure 1.12 A subpackage and its corresponding class definition

The package name `com.oracle.javacert.associate` follows a package-naming convention recommended by Oracle and shown in table 1.1.

Table 1.1 Package-naming conventions used in the package name `com.oracle.javacert.associate`

Package or subpackage name	Its meaning
`com`	Commercial. A couple of the commonly used three-letter package abbreviations are gov—for government bodies edu—for educational institutions
`oracle`	Name of the organization
`javacert`	Further categorization of the project at Oracle
`associate`	Further subcategorization of Java certification

RULES TO REMEMBER

A few of important rules about packages:

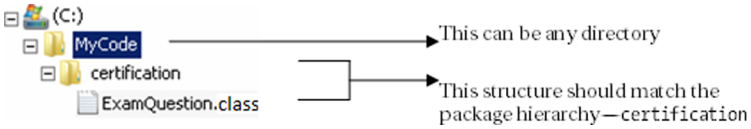
- Per Java naming conventions, package names should all be in lowercase.
- The package and subpackage names are separated using a dot (`.`).
- Package names follow the rules defined for valid identifiers in Java.
- For packaged classes and interfaces, the `package` statement is the first statement in a Java source file (a .java file). The exception is that comments can appear before or after a `package` statement.
- There can be a maximum of one `package` statement per Java source code file (.java file).
- All the classes and interfaces defined in a Java source code file will be defined in the same package. There is no way to package classes and interfaces defined within the same Java source code file in different packages.

NOTE A fully qualified name for a class or interface is formed by prefixing its package name with its name (separated by a period). The fully qualified name of class `ExamQuestion` is `certification.ExamQuestion` in figure 1.11 and `com.oracle.javacert.associate.ExamQuestion` in figure 1.12.

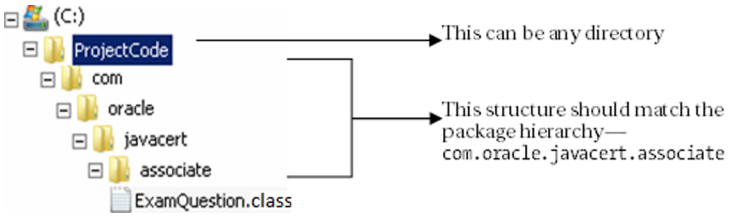
DIRECTORY STRUCTURE AND PACKAGE HIERARCHY

The hierarchy of the packaged classes should match the hierarchy of the directories in which these classes and interfaces are defined in the code. For example, the class `ExamQuestion` in the `certification` package should be defined in a directory with the name "certification."

The name of the directory "certification" and its location are governed by the following rules:



For the package example shown in figure 1.12, note that there isn't any constraint on the location of the base directory in which the directory structure is defined. Examine the following image:



SETTING THE CLASSPATH FOR PACKAGED CLASSES

To enable the Java Runtime Environment (JRE) to find your classes, add the base directory that contains your packaged Java code to the classpath.

For example, to enable the JRE to locate the `certification.ExamQuestion` class from the previous examples, add the directory C:\MyCode to the classpath. To enable the JRE to locate the class `com.oracle.javacert.associate.ExamQuestion`, add the directory C:\ProjectCode to the classpath.

You don't need to bother setting the classpath if you're working with an IDE. But I strongly encourage you to learn how to work with a simple text editor and how to set a classpath. This can be particularly helpful with your projects at work. I have also witnessed many interviewers querying candidates on the need for classpaths.

1.3.3 Using simple names with import statements

The `import` statement enables you to use *simple names* instead of using *fully qualified names* for classes and interfaces defined in separate packages.

Let's work with a real-life example. Imagine your Home and your neighbor's Office. "LivingRoom" and "Kitchen" within your home can refer to each other without mentioning that they exist within the same home. Similarly, in an office, a Cubicle and a ConferenceHall can refer to each other without explicitly mentioning that they exist within the same office. But "Home" and "Office" can't access each other's rooms or cubicles without stating that they exist in a separate home or office. This situation is represented in figure 1.13.

To refer to the LivingRoom in Cubicle, you *must* specify its complete location, as shown in left part of the figure 1.14. As you can see in this figure, repeated references to the location of LivingRoom make the description of LivingRoom look tedious and redundant. To avoid this, you can display a notice in Cubicle that all occurrences of LivingRoom refer to LivingRoom in Home, and thereafter use its simple name. Home

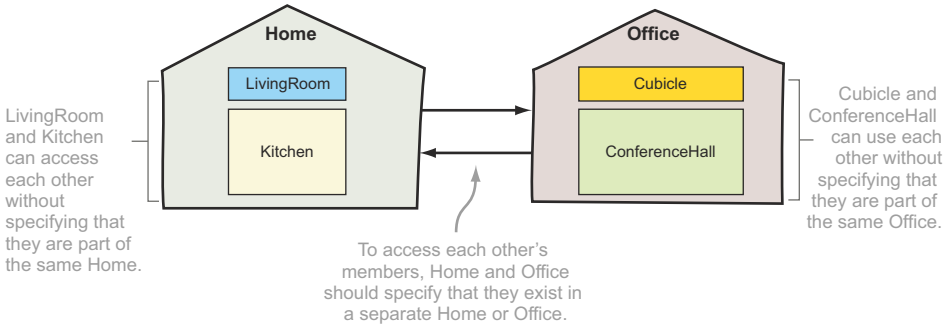


Figure 1.13 To refer to each other's members, Home and Office should specify that they exist in separate places.

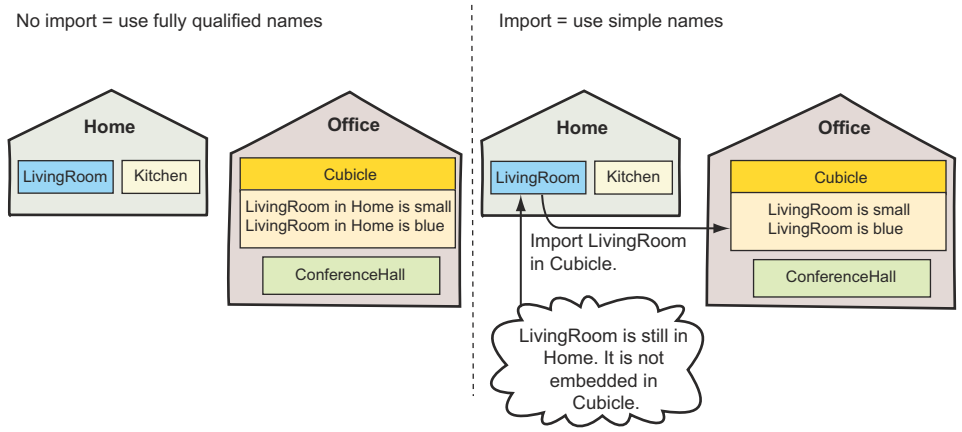


Figure 1.14 LivingRoom can be accessed in Cubicle by using its fully qualified name. It can also be accessed using its simple name if you also use the `import` statement.

and Office are like Java packages, and this notice is the equivalent of the `import` statement. Figure 1.14 shows the difference in using fully qualified names and simple names for Home in Cubicle.

Let's implement the previous example in code, where classes `LivingRoom` and `Kitchen` are defined in the package `home` and classes `Cubicle` and `ConferenceHall` are defined in the package `office`. The class `Cubicle` uses (is associated to) the class `LivingRoom` in the package `home`, as shown in figure 1.15.

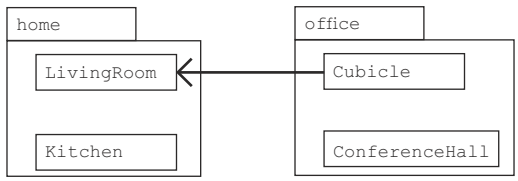


Figure 1.15 A UML representation of classes `LivingRoom` and `Cubicle`, defined in separate packages, with their associations

Class `Cubicle` can refer to class `LivingRoom` without using an `import` statement:

```
package office;
class Cubicle {
    home.LivingRoom livingRoom;
}
```

In the absence of an import statement, use the fully qualified name to access class LivingRoom

Class `Cubicle` can use the simple name for class `LivingRoom` by using the `import` statement:

```
package office;
import home.LivingRoom;

class Cubicle {
    LivingRoom livingRoom;
}
```

import statement

No need to use the fully qualified name of class LivingRoom

NOTE The `import` statement doesn't embed the contents of the imported class in your class, which means that *importing* more classes doesn't increase the size of your own class. It lets you use the simple name for a class or interface defined in a separate package.

1.3.4 Using packaged classes without using the import statement

It is possible to use a packaged class or interface without using the `import` statement, by using its fully qualified name:

```
class AnnualExam {
    certification.ExamQuestion eq;
}
```

Missing import statement

Define a variable of ExamQuestion by using its fully qualified name

This approach can clutter your code if you create multiple variables of interfaces and classes defined in other packages. Use this approach sparingly in actual projects.

For the exam, it's important to note that you can't use the `import` statement to access multiple classes or interfaces with the same names from different packages. For example, the Java API defines the class `Date` in two commonly used packages: `java.util` and `java.sql`. To define variables of these classes in a class, use their fully qualified names with the variable declaration:

```
class AnnualExam {
    java.util.Date date1;
    java.sql.Date date2;
}
```

Missing import statement

Variable of type java.util.Date

Variable of type java.sql.Date

An attempt to use an `import` statement to import both these classes in the same class will not compile:

```
import java.util.Date;
import java.sql.Date;
class AnnualExam { }
```

Code to import classes with the same name from different packages won't compile

1.3.5 Importing a single member versus all members of a package

You can import either a single member or all members (classes and interfaces) of a package using the `import` statement. First, revisit the UML notation of the `certification` package, as shown in figure 1.16.

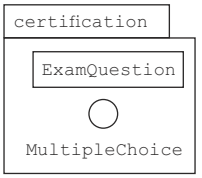


Figure 1.16 A UML representation of the `certification` package

Examine the following code for class `AnnualExam`:

```
import certification.ExamQuestion;
class AnnualExam {
    ExamQuestion eq;
    MultipleChoice mc;
}
```

Imports only the class ExamQuestion

Compiles OK

Will not compile

By using the wildcard character, an asterisk (`*`), you can import all of the `public` members, classes, and interfaces of a package. Compare the previous class definition with the following definition of the class `AnnualExam`:

```
import certification.*;
class AnnualExam {
    ExamQuestion eq;
    MultipleChoice mc;
}
```

Imports all classes and interfaces from certification

Compiles OK

This also compiles OK

Unlike in C or C++, *importing* a class doesn't add to the size of a Java .class file. An `import` statement enables Java to refer to the imported classes without embedding their source code in the target .class file.

When you work with an IDE, it may automatically add `import` statements for classes and interfaces that you reference in your code.

1.3.6 Can you recursively import subpackages?

You can't import classes from a subpackage by using an asterisk in the `import` statement.

For example, the following UML notation depicts the package `com.oracle.javacert` with the class `Schedule`, and two subpackages, `associate` and `webdeveloper`. Package `associate` contains class `ExamQuestion`, and package `webdeveloper` contains class `MarkSheet`, as shown in figure 1.17.

The following `import` statement will import only the class `Schedule`. It won't import the classes `ExamQuestion` and `MarkSheet`:

```
import com.oracle.javacert.*;
```

Imports the class Schedule only

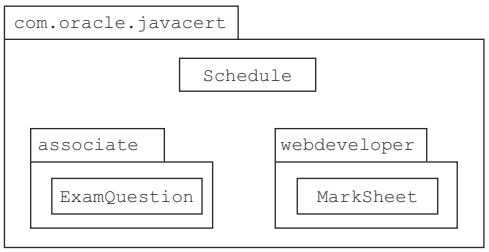


Figure 1.17 A UML representation of package `com.oracle.javacert` and its subpackages

Similarly, the following `import` statement will import all the classes from the packages `associate` and `webdeveloper`:

```
import com.oracle.javacert.associate.*;
import com.oracle.javacert.webdeveloper.*;
```

Imports class ExamQuestion only

Imports class MarkSheet only

1.3.7 Importing classes from the default package

What happens if you don't explicitly package your classes or interfaces? In that case, they're packaged in a *default, no-name* package. This default package is automatically imported in the Java classes and interfaces defined within the same directory on your system.

For example, the classes `Person` and `Office`, which are not defined in an explicit package, can use each other if they are defined in the same directory:

```
class Person {
    // code
}
class Office {
    Person p;
}
```

Not defined in an explicit package

Class Person accessible in class Office

A class from a default package can't be used in any named packaged class, regardless of whether they are defined within the same directory or not.

1.3.8 Static imports

You can import an individual `static` member of a class or all its `static` members by using the `import static` statement.

In the following code, the class `ExamQuestion` defines a `public static` variable named `marks` and a `public static` method named `print`:

```
package certification;
public class ExamQuestion {
    static public int marks;
    public static void print() {
        System.out.println(100);
    }
}
```

public static variable marks

public static method print

The variable `marks` can be accessed in the class `AnnualExam` using the `import static` statement. The order of the keywords `import` and `static` can't be reversed:

```
package university;
import static certification.ExamQuestion.marks;
class AnnualExam {
    AnnualExam() {
        marks = 20;
    }
}
```

Correct statement is import static, not static import

Access variable marks without prefixing it with its class name

To access all `public static` members of class `ExamQuestion` in class `AnnualExam`, you can use an asterisk with the `import static` statement:

```
package university;
import static certification.ExamQuestion.*;
```

Imports all static members of class ExamQuestion

```
class AnnualExam {
    AnnualExam() {
        marks = 20;
        print();
    }
}
```

Accesses variable marks and method print without prefixing them with their class names

Because the variable `marks` and method `print` are defined as `public` members, they are accessible to the class `AnnualExam` using the `import static` statement. These wouldn't be accessible to the class `AnnualExam` if they were defined using any other access modifiers. The accessibility of a class, an interface, and their methods and variables are determined by their access modifiers, which are covered in the next section.

1.4 Java access modifiers

[6.6] Apply access modifiers

In this section, we'll cover all of the access modifiers—`public`, `protected`, and `private`—as well as default access, which is the result when you don't use an access modifier. We'll also look at how you can use access modifiers to restrict the visibility of a class and its members in the same and separate packages.

1.4.1 Access modifiers

Let's start with an example. Examine the definitions of the classes `House` and `Book` in the following code and the UML representation shown in figure 1.18.

```
package building;
class House {}

package library;
class Book {}
```

With the current class definitions, the class `House` cannot access the class `Book`. Can you make the necessary changes (in terms of the access modifiers) to make the class `Book` accessible to the class `House`?

This one shouldn't be difficult. From the discussion of class declarations in section 1.1, you know that a top-level class can be defined only using the `public` or default access modifiers. If you declare the class `Book` using the access modifier `public`, it'll be accessible outside the package in which it is defined.

NOTE A top-level class is a class that isn't defined within any other class. A class that is defined within another class is called a *nested* or *inner class*. Nested and inner classes aren't on the OCA Java SE 7 Programmer I exam.

Figure 1.18 The nonpublic class `Book` cannot be accessed outside the package `library`.

WHAT DO THEY CONTROL?

Access modifiers control the accessibility of a class or an interface, including its members (methods and variables), by other classes and interfaces. For example, you can't access the `private` variables and methods of another class. By using the appropriate access modifiers, you can limit access to your class or interface, and their members, by other classes and interfaces.

CAN ACCESS MODIFIERS BE APPLIED TO ALL TYPES OF JAVA ENTITIES?

Access modifiers can be applied to classes, interfaces, and their members (instance and class variables and methods). Local variables and method parameters can't be defined using access modifiers. An attempt to do so will prevent the code from compiling.

HOW MANY ACCESS MODIFIERS ARE THERE: THREE OR FOUR?

Programmers are frequently confused about the number of access modifiers in Java because the *default access* isn't defined using an explicit keyword. If a Java entity (class, interface, method, or variable) isn't defined using an explicit access modifier, it is said to be defined using the *default access*, also called *package access*.

Java defines four access modifiers:

- `public` (least restrictive)
- `protected`
- *default*
- `private` (most restrictive)

To understand all of these access modifiers, we'll use the same set of classes: `Book`, `CourseBook`, `Librarian`, `StoryBook`, and `House`. Figure 1.19 depicts these classes using UML notation.

The classes `Book`, `CourseBook`, and `Librarian` are defined in the package `library`. The classes `StoryBook` and `House` are defined in the package `building`. Further, classes `StoryBook` and `CourseBook` (defined in separate packages) extend class `Book`. Using these classes, I'll show how the accessibility of a class and its members varies with different access modifiers, from unrelated to derived classes, across packages.

As we cover each of the access modifiers, we'll add a set of instance variables and a method to the class `Book` with the relevant access modifier. We'll then define code for the other classes that try to access class `Book` and its members.

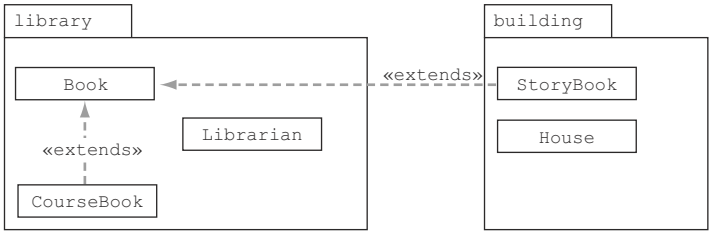


Figure 1.19 A set of classes and their relationships to help understand access modifiers

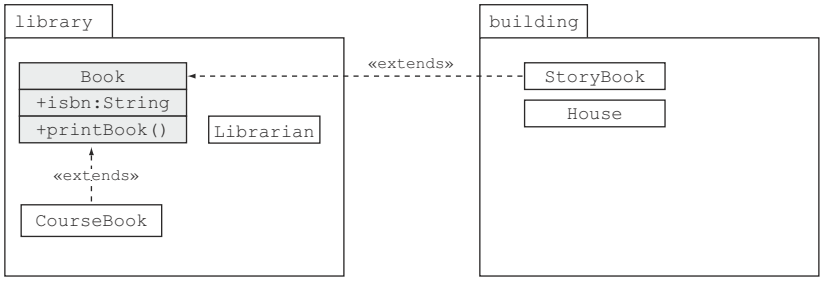


Figure 1.20 Understanding the `public` access modifier

1.4.2 Public access modifier

This is the least restrictive access modifier. Classes and interfaces defined using the `public` access modifier are accessible across all packages, from derived to unrelated classes.

To understand the `public` access modifier, let's define the class `Book` as a `public` class and add a `public` instance variable (`isbn`) and a `public` method (`printBook`) to it. Figure 1.20 shows the UML notation.

Definition of class `Book`:

```
package library;
public class Book {
    public String isbn;
public void printBook() {}
}
```

public class Book

public variable isbn

public method printBook

The `public` access modifier is said to be the least restrictive, so let's try to access the `public` class `Book` and its `public` members from class `House`. We'll use class `House` because `House` and `Book` are defined in separate packages and they're unrelated. Class `House` doesn't enjoy any advantages by being defined in the same package or being a derived class.

Here's the code for class `House`:

```
package building;
import library.Book;
public class House {
    House() {
        Book book = new Book();
        String value = book.isbn;
            book.printBook();
    }
}
```

Class Book is accessible to class House

Variable isbn is accessible in House

Method printBook is accessible in House

As you may notice in the previous example, the class `Book` and its `public` members—instance variable `isbn` and method `printBook`—are accessible to the class `House`. They are also accessible to the other classes: `StoryBook`, `Librarian`, `House`, and `CourseBook`. Figure 1.21 shows the classes that can access a `public` class and its members.

Figure 1.21 Classes that can access a public class and its members

1.4.3 Protected access modifier

The members of a class defined using the `protected` access modifier are accessible to

- Classes and interfaces defined in the same package
- All derived classes, even if they're defined in separate packages

Let's add a `protected` instance variable `author` and method `modifyTemplate` to the class `Book`. Figure 1.22 shows the class representation.

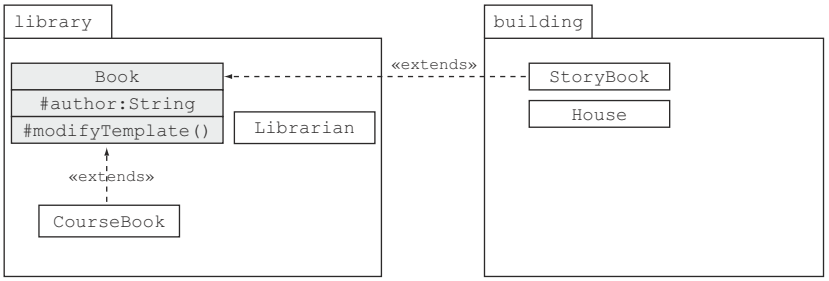


Figure 1.22 Understanding the `protected` access modifier

Here's the code for the class `Book` (I've deliberately left out its `public` members because they aren't required in this section):

```
package library;
public class Book {
    protected String author;
    protected void modifyTemplate() {}
}
```

Protected variable author

Protected method modifyTemplate

Figure 1.23 illustrates how classes from the same and separate packages, derived classes, and unrelated classes access the class `Book` and its `protected` members.

Class `House` throws a compilation error message for trying to access the method `modifyTemplate` and the variable `author`, as follows:

```
House.java:8: modifyTemplate() has protected access in library.Book
        book.modifyTemplate();
             ^
```

Notice that the derived classes `CourseBook` and `StoryBook` can access the class `Book`'s `protected` variable `author` and method `modifyTemplate` as if they were defined in their own classes. If class `StoryBook` tries to create an object of class `Book` and then tries to access its `protected` variable `author` and `modifyTemplate`, it will not compile:

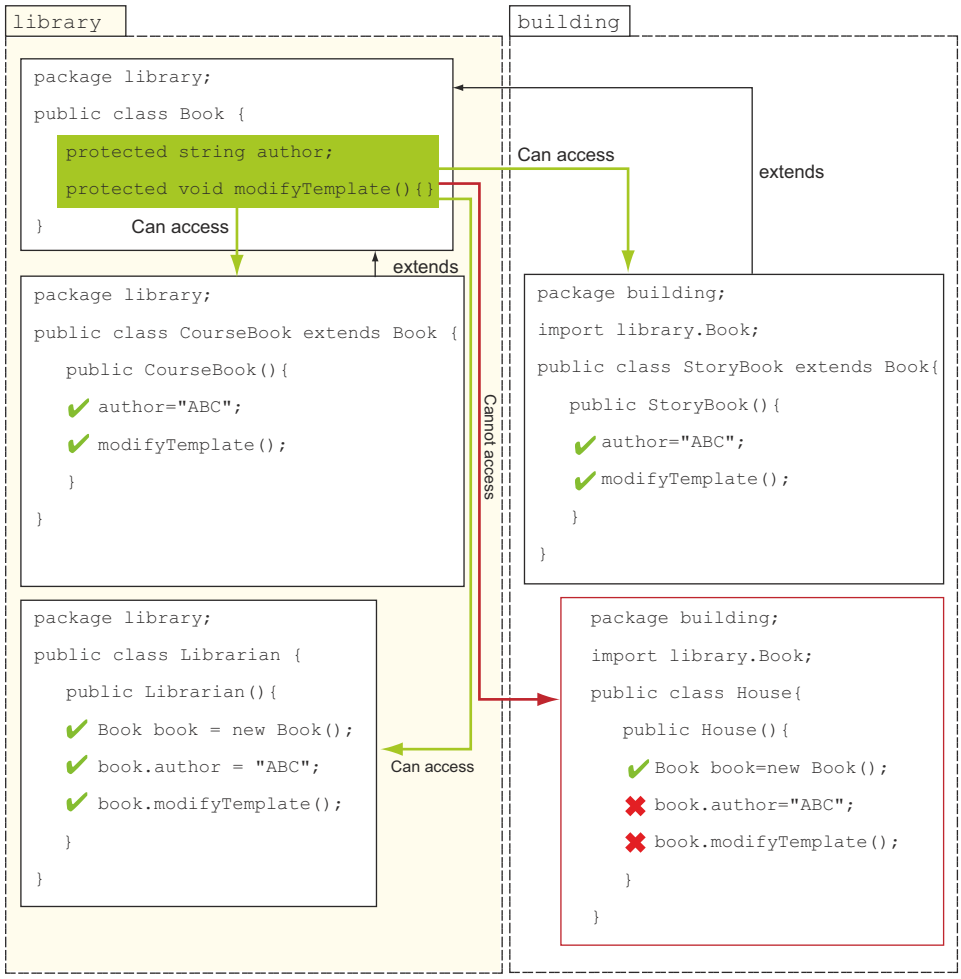


Figure 1.23 Access of `protected` members of the class `Book` in unrelated and derived classes, from the same and separate packages

```
package building;
import library.Book;
class StoryBook extends Book {
    StoryBook() {
        Book book = new Book();
        String v = book.author;
        book.modifyTemplate();
    }
}
```

Classes Book and StoryBook defined in separate packages

Protected members of class Book are not accessible in derived class StoryBook, if accessed using a new object of class Book

EXAM TIP A concise but not too simple way of stating the previous rule is this: a derived class can inherit and access `protected` members of its base class, regardless of the package in which it's defined. A derived class in a separate package can't access `protected` members of its base class using reference variables.

Figure 1.24 Classes that can access protected members

Figure 1.24 shows the classes that can access `protected` members of a class or interface.

1.4.4 Default access (package access)

The members of a class defined without using any explicit access modifier are defined with *package accessibility* (also called *default accessibility*). The members with package access are *only* accessible to classes and interfaces defined in the same package.

Let's define an instance variable `issueCount` and a method `issueHistory` with default access in class `Book`. Figure 1.25 shows the class representation with these new members.

Here's the code for the class `Book` (I've deliberately left out its `public` and `protected` members because they aren't required in this section):

```
package library;
public class Book {
    int issueCount;
    void issueHistory() {}
}
```

Public class Book

Variable issueCount with default access

Method issueHistory with default access

You can see how classes from the same package and separate packages, derived classes, and unrelated classes access the class `Book` and its members (the variable `issueCount` and the method `issueHistory`) in figure 1.26.

Because the classes `CourseBook` and `Librarian` are defined in the same package as the class `Book`, they can access the variables `issueCount` and `issueHistory`. Because the classes `House` and `StoryBook` don't reside in the same package as the class `Book`, they can't access the variables `issueCount` and `issueHistory`. The class `StoryBook` throws the following compilation error message:

```
StoryBook.java:6: issueHistory() is not public in library.Book; cannot be
     accessed from outside package
        book.issueHistory();
             ^
```

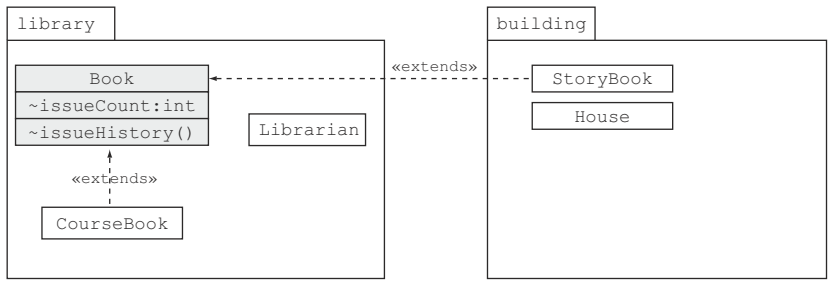


Figure 1.25 Understanding class representation for default access

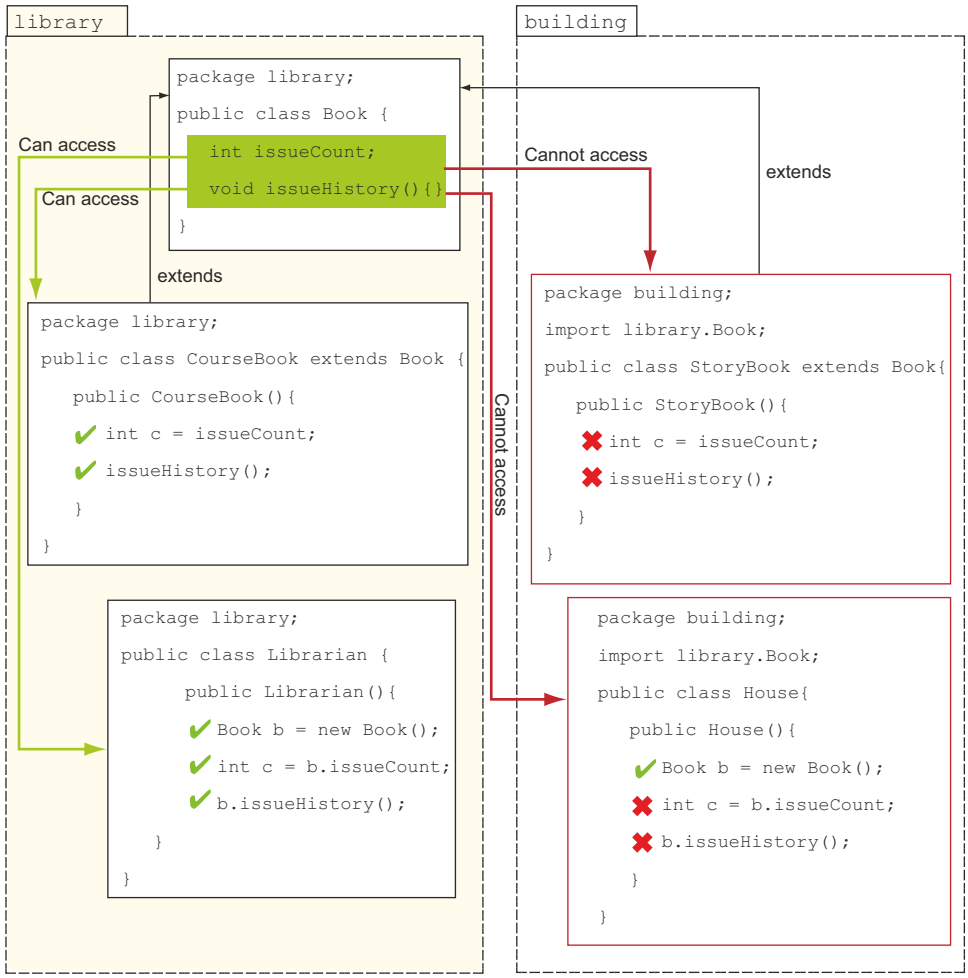


Figure 1.26 Access of members with default access to the class `Book` in unrelated and derived classes from the same and separate packages

The class `House` throws the following compilation error message (for trying to access `issueHistory`):

```
House.java:9: cannot find symbol
symbol  : method issueHistory()
location: class building.House
        issueHistory();
```

DEFINING A CLASS BOOK WITH DEFAULT ACCESS

What happens if we define a class with default access? What will happen to the accessibility of its members if the class itself has *default* (package) accessibility?

Let's consider this situation using an example: assume that Superfast Burgers opens a new outlet on a beautiful island and offers free meals to people from all over

How packages and class hierarchy affect default access to class members

From the compilation errors thrown by the Java compiler when trying to compile the classes `StoryBook` and `House`, you can see that the method `issueHistory` (defined in the class `Book`) is visible to its derived class `StoryBook` (defined in another package), but `StoryBook` cannot access it. The method `issueHistory` (defined in class `Book`) is not even visible to the unrelated class `House` defined in a separate package.

the world, which obviously includes inhabitants of the island. But the island is inaccessible by all means (air and water). Would the existence of this particular Superfast Burgers outlet make any sense to people who don't inhabit the island? An illustration of this example is shown in figure 1.27.

The island is like a package in Java, and the Superfast Burgers like a class defined with default access. In the same way that the Superfast Burgers cannot be accessed from outside the island in which it exists, a class defined with default (package) access is visible and accessible only from within the package in which it is defined. It can't be accessed from outside the package in which it resides.

Let's redefine the class `Book` with default (package) access, as follows:

```
package library;
class Book {
    //.. class members
}
```

Class Book now has default access

The behavior of the class `Book` remains the same for the classes `CourseBook` and `Librarian`, which are defined in the same package. But the class `Book` can't be accessed by classes `House` and `StoryBook`, which reside in a separate package.

Let's start with the class `House`. Examine the following code:

```
package building;
import library.Book;
public class House {}
```

Class Book isn't accessible in class House

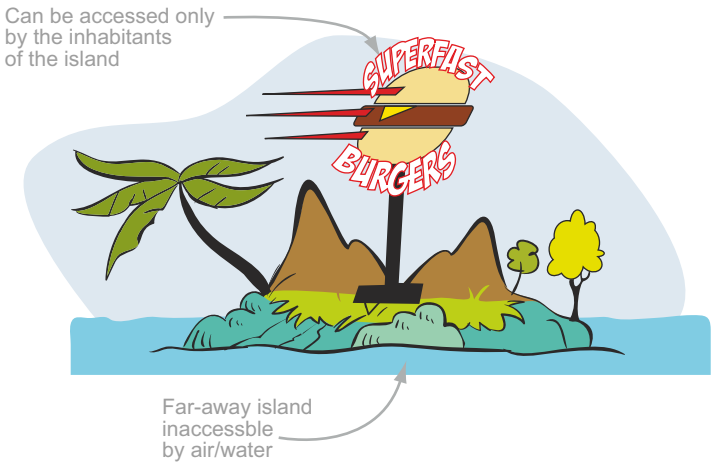


Figure 1.27 This Superfast Burgers cannot be accessed from outside the island because the island is inaccessible by air and water.

Figure 1.28 The classes that can access members with default (package) access

The class `House` generates the following compilation error message:

```
House.java:2: library.Book is not public in library; cannot be accessed from
     outside package
import library.Book;
```

Here's the code of the class `StoryBook`:

```
package building;
import library.Book;
class StoryBook extends Book {}
```

Book isn't accessible in StoryBook

StoryBook cannot extend Book

Figure 1.28 shows which classes can access members of a class or interface with default (package) access.

Because a lot of programmers are confused about which members are made accessible by using the `protected` and default access modifiers, the following exam tip offers a simple and interesting rule to help you remember their differences.

EXAM TIP Default access can be compared to package-private (accessible only within a package) and `protected` access can be compared to package-private + *kids* ("kids" refer to derived classes). Kids can access `protected` methods only by inheritance and not by reference (accessing members by using the dot operator on an object).

1.4.5 Private access modifier

The `private` access modifier is the most restrictive access modifier. The members of a class defined using the `private` access modifier are accessible only to themselves. It doesn't matter whether the class or interface in question is from another package or has extended the class—`private` members are *not* accessible outside the class in which they're defined. `private` members are accessible only to the classes and interfaces in which they are defined.

Let's see this in action by adding a `private` method `countPages` to the class `Book`. Figure 1.29 depicts the class representation using UML.

Examine the following definition of the class `Book`:

```
package library;
class Book {
    private void countPages() {}
    protected void modifyTemplate() {
        countPages();
    }
}
```

private method

Only Book can access its own private method countPages

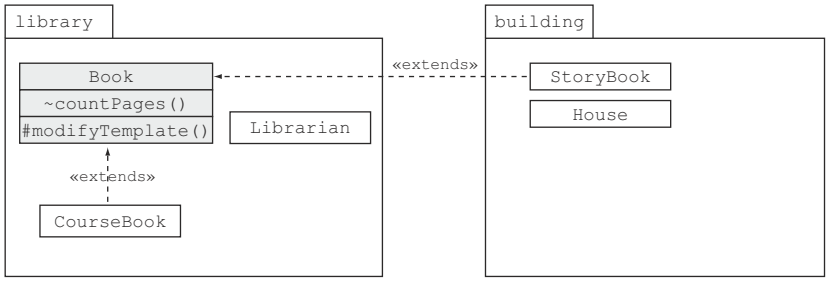


Figure 1.29 Understanding the `private` access modifier

None of the classes defined in any of the packages (whether derived or not) can access the `private` method `countPages`. But let's try to access it from the class `CourseBook`. I chose the class `CourseBook` because both of these classes are defined in the same package, and the class `CourseBook` extends the class `Book`. Here's the code of `CourseBook`:

```
package library;
class CourseBook extends Book {
    CourseBook() {
        countPages();
    }
}
```

CourseBook extends Book

CourseBook cannot access private method countPages

Because the class `CourseBook` tries to access private members of the class `Book`, it will not compile. Similarly, if any of the other classes (`StoryBook`, `Librarian`, `House`, or `CourseBook`) tries to access the `private` method `countPages()` of class `Book`, it will not compile. Figure 1.30 shows the classes that can access the `private` members of a class.

Figure 1.30 No classes can access `private` members of another class

Twist in the Tale 1.4

The following task was assigned to a group of programmers: "How can you declare a class `Curtain` in a package `building` so that it isn't visible outside the package `building`?"

These are the answers submitted by Paul, Shreya, Harry, and Selvan. Which of these do you think is correct, and why? (You can check your Twist in the Tale answers in the appendix.)

Programmer name	Submitted code
Paul	`package building;` `public class Curtain {}`
Shreya	`package building;` `protected class Curtain {}`
Harry	`package building;` `class Curtain {}`
Selvan	`package building;` `private class Curtain {}`

Your job title may assign special privileges or responsibilities to you. For example, if you work as a Java developer, you may be responsible for updating your programming skills or earning professional certifications in Java. Similarly, you can assign special privileges, responsibilities, and behaviors to your Java entities by using *nonaccess modifiers*, which are covered in the next section.

1.5 Nonaccess modifiers

[7.6] Use abstract classes and interfaces

[6.2] Apply the static keyword to methods and fields

This section discusses the nonaccess modifiers `abstract`, `final`, and `static`. Access modifiers control the accessibility of your class and its members outside the class and the package. Nonaccess modifiers change the default properties of a Java class and its members.

For example, if you add the keyword `abstract` to the definition of a class, it'll be considered an `abstract` class. None of the other classes will be able to create objects of this class. Such is the magic of the nonaccess modifiers.

You can characterize your classes, interfaces, methods, and variables with the following nonaccess modifiers (though not all are applicable to each Java entity):

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

The OCA Java SE 7 Programmer I exam covers only three of these nonaccess modifiers: `abstract`, `final`, and `static`, which I'll cover in detail. To ward off any confusion about the rest of the modifiers, I'll describe them briefly here:

- `synchronized`—A `synchronized` method can't be accessed by multiple threads concurrently. This constraint is used to protect the integrity of data that might be accessed and changed by multiple threads concurrently. You can't mark classes, interfaces, or variables with this modifier.
- `native`—A `native` method calls and makes use of libraries and methods implemented in other programming languages such as C or C++. You can't mark classes, interfaces, or variables with this modifier.
- `transient`—A `transient` variable isn't serialized when the corresponding object is serialized. The `transient` modifier can't be applied to classes, interfaces, or methods.
- `volatile`—A `volatile` variable's value can be safely modified by different threads. Classes, interfaces, and methods cannot use this modifier.
- `strictfp`—Classes, interfaces, and methods defined using this keyword ensure that calculations using floating-point numbers are identical on all platforms. This modifier can't be used with variables.

Now let's look at the three nonaccess modifiers that are on the exam.

1.5.1 Abstract modifier

When added to the definition of a class, interface, or method, the `abstract` modifier changes its default behavior. Because it is a nonaccess modifier, `abstract` doesn't change the accessibility of a class, interface, or method.

Let's examine the behavior of each of these with the `abstract` modifier.

ABSTRACT CLASS

When the `abstract` keyword is prefixed to the definition of a concrete class, it changes it to an `abstract` class, even if the class doesn't define any `abstract` methods. The following code is a valid example of an `abstract` class:

```
abstract class Person {
    private String name;
    public void displayName() { }
}
```

An `abstract` class can't be instantiated, which means that the following code will fail to compile:

```
class University {
    Person p = new Person();
}
```

This line of code won't compile

Here's the compilation error thrown by the previous class:

```
University.java:4: Person is abstract; cannot be instantiated
    Person p = new Person();
               ^
1 error
```

EXAM TIP An `abstract` class may or may not define an `abstract` method; you can define an `abstract` class without any `abstract` methods. But a concrete class can't define an `abstract` method.

ABSTRACT INTERFACE

An interface is an abstract entity by default. The Java compiler automatically adds the keyword `abstract` to the definition of an interface. Thus, adding the keyword `abstract` to the definition of an interface is redundant. The following definitions of interfaces are the same:

```
interface Movable {}
abstract interface Movable {}
```

Interface defined without the explicit use of keyword abstract

Interface defined with the explicit use of keyword abstract

ABSTRACT METHOD

An `abstract` method doesn't have a body. Usually, an `abstract` method is implemented by a derived class. Here's an example:

```
abstract class Person {
    private String name;
    public void displayName() { }
    public abstract void perform();
}
```

This isn't an abstract method. It has an empty body: {}.

This is an abstract method. It isn't followed by {}.

EXAM TIP A method with an empty body isn't an `abstract` method.

ABSTRACT VARIABLES

None of the different types of variables (instance, static, local, and method parameters) can be defined as `abstract`.

EXAM TIP Don't be tricked by code that tries to apply the nonaccess modifier `abstract` to a variable. Such code won't compile.

1.5.2 *Final modifier*

The keyword `final` changes the default behavior of a class, variable, or method.

FINAL CLASS

A class that is marked `final` cannot be extended by another class. The class `Professor` will not compile if the class `Person` is marked as `final`, as follows:

```
final class Person {}
class Professor extends Person {}
```

Won't compile

FINAL INTERFACE

An interface cannot be marked as `final`. An interface is `abstract` by default and marking it with `final` will prevent your interface from compiling:

```
final interface MyInterface{}
```

Won't compile

FINAL VARIABLE

A `final` variable can't be reassigned a value. It can be assigned a value only once. See the following code:

```
class Person {
    final long MAX_AGE;
    Person() {
        MAX_AGE = 99;
    }
}
```

Compiles successfully: value assigned once to final variable

Compare the previous example with the following code, which tries to reassign a value to a final variable:

```
class Person {
    final long MAX_AGE = 90;
    Person() {
        MAX_AGE = 99;
    }
}
```

Won't compile; reassignment not allowed

It's easy to confuse reassigning a value to a `final` variable with *calling* a method on a `final` variable. If a reference variable is defined as a `final` variable, you can't reassign another object to it, but you can call methods on this variable:

```
class Person {
    final StringBuilder name = new StringBuilder("Sh");
    Person() {
        name.append("reya");
        name = new StringBuilder();
    }
}
```

Can call methods on a final variable

Won't compile. You can't reassign another object to a final variable.

FINAL METHOD

A `final` method defined in a base class can't be overridden by a derived class. Examine the following code:

```
class Person {
    final void sing() {
        System.out.println("la..la..la..");
    }
}
class Professor extends Person {
    void sing() {
        System.out.println("Alpha.. beta.. gamma");
    }
}
```

Won't compile

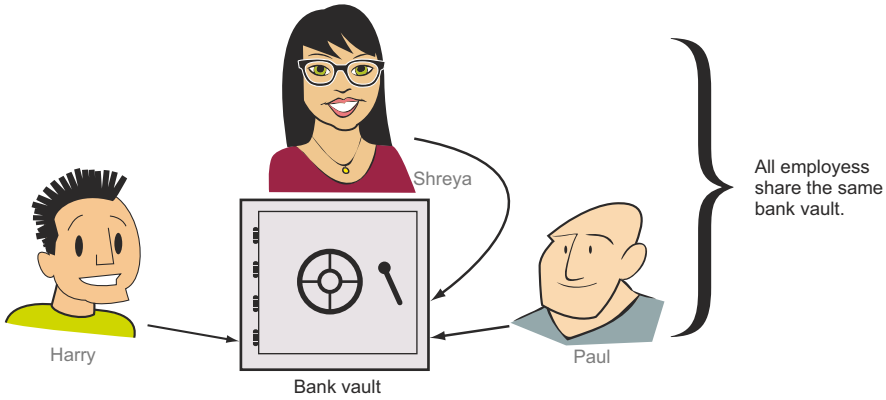


Figure 1.31 Comparing a shared bank vault with a static variable

If a method in a derived class has the same method signature as its base class's method, it is referred to as an *overridden method*. Overridden methods are discussed along with polymorphism in chapter 6.

1.5.3 Static modifier

The nonaccess modifier `static`, when applied to the definitions of variables, methods, classes, and interfaces, changes their default behavior. We'll examine each of them in following sections.

STATIC VARIABLES

`static` variables belong to a class. They are common to all instances of a class and aren't unique to any instance of a class. `static` attributes exist independently of any instances of a class and may be accessed even when no instances of the class have been created. You can compare a `static` variable with a shared variable. A `static` variable is shared by all of the objects of a class.

Think of a `static` variable as being like a common bank vault that's shared by the employees of an organization. Each of the employees accesses the same bank vault, so any change made by one employee is visible to all the other employees, as illustrated in figure 1.31.

Figure 1.32 defines a class `Emp` that defines a non-`static` variable `name` and a `static` variable `bankVault`.

It's time to test what we've been discussing up to this point. The following `TestEmp` class creates two objects of the class `Emp` (from figure 1.32) and modifies the value of the variable `bankVault` using these separate objects:

```
class Emp {
        String name;
        static int bankVault;
}
```

We want this value to be shared by all the objects of class `Emp`.

Figure 1.32 Definition of the class `Emp` with a `static` variable `bankVault` and non-`static` variable `name`

```
class TestEmp {
    public static void main(String[] args) {
    Emp emp1 = new Emp();
        Emp emp2 = new Emp();
        emp1.bankVault = 10;
        emp2.bankVault = 20;

        System.out.println(emp1.bankVault);
        System.out.println(emp2.bankVault);
        System.out.println(Emp.bankVault);
    }
}
```

Reference variables emp1 and emp2 refer to separate objects of class Emp

Variable bankVault of variable emp2 is assigned a value of 20

Variable bankVault of variable emp1 is assigned a value of 10

This will print 20

This will also print 20

This will print 20 as well

In the preceding code example, `emp1.bankVault`, `emp2.bankVault`, and `Emp.bankVault` all refer to the *same* `static` attribute: `bankVault`.

EXAM TIP A `static` variable can be accessed using the name of the object reference variable or the name of a class.

The `static` and `final` nonaccess modifiers can be used to define *constants* (variables whose value can't change). In the following code, the class `Emp` defines the constants `MIN_AGE` and `MAX_AGE`:

```
class Emp {
    public static final int MIN_AGE = 20;
    static final int MAX_AGE = 70;
}
```

Constant MIN_AGE

Constant MAX_AGE

Though you can define a constant as a non-`static` member, it's common practice to define constants as `static` members, as doing so allows the constant values to be used across objects and classes.

STATIC METHODS

`static` methods aren't associated with objects and can't use any of the instance variables of a class. You can define `static` methods to access or manipulate `static` variables:

```
class Emp {
    String name;
    static int bankVault;

    static int getBankVaultValue() {
        return bankVault;
    }
}
```

static method getBankVaultValue returns the value of static variable bankVault

You can also use `static` methods to define *utility methods*, which are methods that usually manipulate the method parameters to compute and return an appropriate value:

```
static double interest(double num1, double num2, double num3) {
    return(num1+num2+num3)/3;
}
```

A `static` method may not always define method parameters. The method `averageOfFirst100Integers` computes and returns the average of numbers `1` to `100`:

```
static double averageOfFirst100Integers() {
    int sum = 0;
    for (int i=1; i <= 100; ++i) {
        sum += i;
    }
    return (sum)/100;
}
```

Method averageOfFirst100Integers doesn't define method parameters

The non-private `static` variables and methods are inherited by derived classes. The `static` members aren't involved in runtime polymorphism. You can't override the `static` members in a derived class, but you can redefine them.

Any discussion of `static` methods and their behavior can be quite confusing if you aren't aware of inheritance and derived classes. But don't worry if you don't understand all of it. I'll cover derived classes and inheritance in chapter 6. For now, note that a `static` method can be accessed using the name of the object reference variables and the class in a manner similar to `static` variables.

NOTE Even though you can use an object reference variable to access `static` members, it's not advisable to do so. Because `static` members belong to a class and not to individual objects, using object reference variables to access `static` members may make them appear to belong to an object. The proper way to access them is by using the class name.

WHAT CAN A STATIC METHOD ACCESS?

Neither `static` methods nor `static` variables can access the non-`static` variables and methods of a class. But the reverse is true: non-`static` variables and methods can access `static` variables and methods because the `static` members of a class exist even if no instances of the class exist. `static` members are forbidden from accessing instance methods and variables, which can exist only if an instance of the class is created.

Examine the following code:

```
class MyClass {
    static int x = count();
    int count() { return 10; }
}
```

Compilation error

This is the compilation error thrown by the previous class:

```
MyClass.java:3: nonstatic method count() cannot be referenced from a static
     context
    static int x = count();
                   ^
1 error
```

The following code is valid:

```
class MyClass {
    static int x = result();
    static int result() { return 20; }
```

static variable referencing a static method

```
    int nonStaticResult() { return result(); }
}
```

Non-static method using static method

EXAM TIP Static methods and variables can't access the instance members of a class.

Static classes and interfaces

Certification aspirants frequently ask questions about `static` classes and interfaces, so I'll quickly cover these in this section to ward off any confusion related to them. But note that `static` classes and interfaces are types of nested classes and interfaces that aren't covered by the OCA Java 7 Programmer I exam.

You can't prefix the definition of a top-level class or an interface with the keyword `static`. A top-level class or interface is one that isn't defined within another class or interface. The following code will fail to compile:

```
static class Person {}
static interface MyInterface {}
```

But you can define a class and an interface as a static member of another class. The following code is valid:

```
class Person {
    static class Address {}
    static interface MyInterface {}
}
```

Also known as a static nested class

1.6 Summary

This chapter started with a look at the structure of a Java class. Although you should know how to work with Java classes, Java source code files (.java files), and Java bytecode files (.class files), the OCA Java SE 7 Programmer I exam will question you only on the structure and components of the first two—classes and source code—not on Java bytecode.

We discussed the components of a Java class and of Java source code files. A class can define multiple components, namely `import` and `package` statements, variables, constructors, methods, comments, nested classes, nested interfaces, annotations, and enums. A Java source code file (.java) can define multiple classes and interfaces.

We then covered the differences and similarities between executable and nonexecutable Java classes. An executable Java class defines the entry point (`main` method) for the JVM to start its execution. The `main` method should be defined with the required method signature; otherwise, the class will fail to be categorized as an executable Java class.

Packages are used to group together related classes and interfaces. They also provide access protection and namespace management. The `import` statement is used to import classes and interfaces from other packages. In the absence of an `import` statement, classes and interfaces should be referred to by their fully qualified names (complete package name plus class or interface name).

Access modifiers control the access of classes and their members within a package and across packages. Java defines four access modifiers: `public`, `protected`, default, and `private`. When default access is assigned to a class or its member, no access modifier is prefixed to it. The absence of an access modifier is equal to assigning the class or its members with default access. The least restrictive access modifier is `public`, and `private` is the most restrictive. `protected` access sits between `public` and default access, allowing access to derived classes outside of a package.

Finally, we covered the `abstract` and `static` nonaccess modifiers. A class or a method can be defined as an `abstract` member. `abstract` classes can't be instantiated. Methods and variables can be defined as `static` members. All the objects of a class share the same copy of `static` variables, which are also known as class-level variables.

1.7 Review notes

This section lists the main points covered in this chapter.

The structure of a Java class and source code file:

- The OCA Java SE 7 Programmer I exam covers the structure and components of a Java class and Java source code file (.java file). It doesn't cover the structure and components of Java bytecode files (.class files).
- A class can define multiple components. All the Java components you've heard of can be defined within a Java class: `import` and `package` statements, variables, constructors, methods, comments, nested classes, nested interfaces, annotations, and enums.
- The OCA Java SE 7 Programmer I exam doesn't cover the definitions of nested classes, nested interfaces, annotations, and enums.
- If a class defines a `package` statement, it should be the first statement in the class definition.
- The `package` statement can't appear within a class declaration or after the class declaration.
- If present, the `package` statement should appear exactly once in a class.
- The `import` statement uses simple names of classes and interfaces from within the class.
- The `import` statement can't be used to import multiple classes or interfaces with the same name.
- A class can include multiple `import` statements.
- If a class includes a `package` statement, all the `import` statements should follow the `package` statement.
- Comments are another component of a class. Comments are used to annotate Java code and can appear at multiple places within a class.
- A comment can appear before or after a `package` statement, before or after the class definition, and before, within, or after a method definition.
- Comments come in two flavors: multiline and end-of-line comments.

- Comments can contain any special characters (including characters from the Unicode charset).
- Multiline comments span multiple lines of code. They start with `/*` and end with `*/`.
- End-of-line comments start with `//` and, as the name suggests, are placed at the end of a line of code. The text between `//` and the end of the line is treated as a comment.
- Class declarations and class definitions are components of a Java class.
- A Java class may define zero or more instance variables, methods, and constructors.
- The order of the definition of instance variables, constructors, and methods doesn't matter in a class.
- A class may define an instance variable before or after the definition of a method and still use it.
- A Java source code file (.java file) can define multiple classes and interfaces.
- A `public` class can be defined only in a source code file with the same name.
- `package` and `import` statements apply to all the classes and interfaces defined in the same source code file (.java file).

Executable Java applications:

- An executable Java class is a class that, when handed over to the Java Virtual Machine (JVM), starts its execution at a particular point in the class. This point of execution is the `main` method.
- For a class to be executable, the class should define a `main` method with the signature `public static void main(String args[])` or `public static void main(String... args)`. The positions of `static` and `public` can be interchanged, and the method parameter can use any valid name.
- A class can define multiple methods with the name `main`, provided that the signature of these methods doesn't match the signature of the `main` method defined in the previous point. These other methods with different signatures aren't considered *the* `main` method.
- The `main` method accepts an array of type `String` containing the method parameters passed to it by the JVM.
- The keyword `java` and the name of the class aren't passed on as command parameters to the `main` method.

Java packages:

- You can use packages to group together a related set of classes and interfaces.
- By default, all classes and interfaces in separate packages and subpackages aren't visible to each other.
- The package and subpackage names are separated using a period.
- All classes and interfaces in the same package are visible to each other.
- An `import` statement allows the use of simple names for packaged classes and interfaces defined in other packages.

- You can't use the `import` statement to access multiple classes or interfaces with the same names from different packages.
- You can import either a single member or all members (classes and interfaces) of a package using the `import` statement.
- You can't import classes from a subpackage by using the wildcard character, an asterisk (`*`), in the `import` statement.
- A class from a default package can't be used in any named packaged class, regardless of whether it's defined within the same directory or not.
- You can import an individual `static` member of a class or all its `static` members by using a `static import` statement.
- An `import` statement can't be placed before a `package` statement in a class. Any attempt to do so will cause the compilation of the class to fail.
- The members of default packages are accessible only to classes or interfaces defined in the same directory on your system.

Java access modifiers:

- The access modifiers control the accessibility of your class and its members outside the class and package.
- Java defines four access modifiers: `public`, `protected`, default, and `private`.
- The `public` access modifier is the least restrictive access modifier.
- Classes and interfaces defined using the `public` access modifier are accessible to related and unrelated classes outside the package in which they're defined.
- The members of a class defined using the `protected` access modifier are accessible to classes and interfaces defined in the same package and to all derived classes, even if they're defined in separate packages.
- The members of a class defined without using an explicit access modifier are defined with package accessibility (also called default accessibility).
- The members with package access are accessible only to classes and interfaces defined in the same package.
- A class defined using default access can't be accessed outside its package.
- The members of a class defined using a `private` access modifier are accessible only to the class in which they are defined. It doesn't matter whether the class or interface in question is from another package or has extended the class. Private members are not accessible outside the class in which they're defined.
- The `private` access modifier is the most restrictive access modifier.

Nonaccess modifiers:

- The nonaccess modifiers change the default properties of a Java class and its members.
- The OCA Java SE 7 Programmer I exam covers only two nonaccess modifiers: `abstract` and `static`.

- The `abstract` keyword, when prefixed to the definition of a concrete class, can change it to an `abstract` class, even if it doesn't define any `abstract` methods.
- An `abstract` class cannot be instantiated.
- An interface is an `abstract` entity by default. The Java compiler automatically adds the keyword `abstract` to the definition of an interface (which means that adding the keyword `abstract` to the definition of an interface is redundant).
- An `abstract` method doesn't have a body, which means it's implemented by the class that extends the class defining the `abstract` method.
- A variable can't be defined as an `abstract` variable.
- The `static` modifier can be applied to inner classes, inner interfaces, variables, and methods. Inner classes and interfaces aren't covered in this exam.
- A method can't be defined both as `abstract` and `static`.
- `static` attributes (fields and methods) are common to all instances of a class and aren't unique to any instance of a class.
- `static` attributes exist independent of any instances of a class and may be accessed even when no instances of the class have been created.
- `static` attributes are also known as *class fields* or *class methods* because they're said to belong to their class, not to any instance of that class.
- A `static` variable or method can be accessed using the name of a reference object variable or the name of a class.
- A `static` method or variable can't access non-`static` variables or methods of a class. But the reverse is true: non-`static` variables and methods can access static variables and methods.
- `static` classes and interfaces are a type of nested classes and interfaces but they aren't covered in the OCA Java SE 7 Programmer I exam.
- You can't prefix the definition of a top-level class or an interface with the keyword `static`. A top-level class or interface is one that isn't defined within another class or interface.

1.8 Sample exam questions

Q1-1. What are the valid components of a Java source file (choose all that apply):

a `package` statement
b `import` statements
c methods
d variables
e Java compiler
f Java Runtime Environment

Q1-2. The following numbered list of Java class components is not in any particular order. Select the correct order of their occurrence in a Java class (choose all that apply):

1. comments
2. import statement
3. package statement
4. methods
5. class declaration
6. variables

a. 1, 3, 2, 5, 6, 4
b. 3, 1, 2, 5, 4, 6
c. 3, 2, 1, 4, 5, 6
d. 3, 2, 1, 5, 6, 4

Q1-3. Which of the following examples define the correct Java class structure?

a.
```
#connect java compiler;
#connect java virtual machine;
class EJavaGuru {}
```

b.
```
package java compiler;
import java virtual machine;
classs EJavaGuru {}
```

c.
```
import javavirtualmachine.*;
package javacompiler;
class EJavaGuru {
    void method1() {}
    int count;
}
```

d.
```
package javacompiler;
import javavirtualmachine.*;
class EJavaGuru {
    void method1() {}
    int count;
}
```

e.
```
#package javacompiler;
$import javavirtualmachine;
class EJavaGuru {
    void method1() {}
    int count;
}
```

f.
```
package javacompiler;
import javavirtualmachine;
Class EJavaGuru {
    void method1() {}
    int count;
}
```

Q1-4. Given the following contents of the Java source code file MyClass.java, select the correct options:

```
// contents of MyClass.java
package com.ejavaguru;
import java.util.Date;
class Student {}
class Course {}
```

- **a** The imported class, `java.util.Date`, can be accessed only in the class `Student`.
- **b** The imported class, `java.util.Date`, can be accessed by both the `Student` and `Course` classes.
- **c** Both of the classes `Student` and `Course` are defined in the package `com.ejavaguru`.
- **d** Only the class `Student` is defined in the package `com.ejavaguru`. The class `Course` is defined in the default Java package.

Q1-5. Given the following definition of the class `EJavaGuru`,

```
class EJavaGuru {
    public static void main(String[] args) {
        System.out.println(args[1]+":"+ args[2]+":"+ args[3]);
    }
}
```

what is the output of the previous class, if it is executed using the following command:

```
java EJavaGuru one two three four
```

- **a** `one:two:three`
- **b** `EJavaGuru:one:two`
- **c** `java:EJavaGuru:one`
- **d** `two:three:four`

Q1-6. Which of the following options, when inserted at `//INSERT CODE HERE`, will print out `EJavaGuru`?

```
public class EJavaGuru {
    // INSERT CODE HERE
    {
        System.out.println("EJavaGuru");
    }
}
```

- **a** `public void main (String[] args)`
- **b** `public void main(String    args[])`
- **c** `static public void main     (String[] array)`
- **d** `public static void main (String args)`
- **e** `static public main (String args[])`

Q1-7. Select the correct options:

- **a** You can start the execution of a Java application through the `main` method.
- **b** The Java compiler calls and executes the `main` method.
- **c** The Java Virtual Machine calls and executes the `main` method.
- **d** A class calls and executes the `main` method.

Q1-8. A class `Course` is defined in a package `com.ejavaguru`. Given that the physical location of the corresponding class file is /mycode/com/ejavaguru/Course.class and execution takes place within the mycode directory, which of the following lines of code, when inserted at `// INSERT CODE HERE`, will import the `Course` class into the class `MyCourse`?

```
// INSERT CODE HERE
class MyCourse {
    Course c;
}
```

- **a** `import mycode.com.ejavaguru.Course;`
- **b** `import com.ejavaguru.Course;`
- **c** `import mycode.com.ejavaguru;`
- **d** `import com.ejavaguru;`
- **e** `import mycode.com.ejavaguru*;`
- **f** `import com.ejavaguru*;`

Q1-9. Examine the following code:

```
class Course {
    String courseName;
}
class EJavaGuru {
    public static void main(String args[]) {
        Course c = new Course();
        c.courseName = "Java";
        System.out.println(c.courseName);
    }
}
```

Which of the following statements will be true if the variable `courseName` is defined as a `private` variable?

- **a** class `EJavaGuru` will print Java.
- **b** class `EJavaGuru` will print null.
- **c** class `EJavaGuru` won't compile.
- **d** class `EJavaGuru` will throw an exception at runtime.

Q1-10. Given the following definition of the class `Course`,

```
package com.ejavaguru.courses;
class Course {
    public String courseName;
}
```

what's the output of the following code?

```
package com.ejavaguru;
import com.ejavaguru.courses.Course;
class EJavaGuru {
    public static void main(String args[]) {
        Course c = new Course();
        c.courseName = "Java";
        System.out.println(c.courseName);
    }
}
```

a The class `EJavaGuru` will print Java.
b The class `EJavaGuru` will print null.
c The class `EJavaGuru` won't compile.
d The class `EJavaGuru` will throw an exception at runtime.

Q1-11. Given the following code, select the correct options:

```
package com.ejavaguru.courses;
class Course {
    public String courseName;
    public void setCourseName(private String name) {
        courseName = name;
    }
}
```

a You can't define a method argument as a `private` variable.
b A method argument should be defined with either `public` or default accessibility.
c For overridden methods, method arguments should be defined with `protected` accessibility.
d None of the above.

1.9 Answers to sample exam questions

Q1-1. What are the valid components of a Java source file (choose all that apply):

a **`package` statement**
b **`import` statements**
c **methods**
d **variables**
e Java compiler
f Java Runtime Environment

Answer: a, b, c, d

Explanation: The Java compiler and Java Runtime Environment aren't components of a Java source file.

Q1-2. The following numbered list of Java class components is not in any particular order. Select the correct order of their occurrence in a Java class (choose all that apply):

1. comments
2. import statement
3. package statement
4. methods
5. class declaration
6. variables

a. **1, 3, 2, 5, 6, 4**
b. **3, 1, 2, 5, 4, 6**
c. 3, 2, 1, 4, 5, 6
d. **3, 2, 1, 5, 6, 4**

Answer: a, b, d

Explanation: The comments can appear anywhere in a class. They can appear before and after `package` and `import` statements. They can appear before or after a class, method, or variable declaration.

The first statement (if present) in a class should be a `package` statement. It can't be placed after an `import` statement or a declaration of a class.

The `import` statement should follow a `package` statement and be followed by a class declaration.

The class declaration follows the `import` statements, if present. It's followed by the declaration of the methods and variables.

Answer (c) is incorrect. None of the variables or methods can be defined before the definition of a class or interface.

Q1-3. Which of the following examples define the correct Java class structure?

a.
```
#connect java compiler;
#connect java virtual machine;
class EJavaGuru {}
```

b.
```
package java compiler;
import java virtual machine;
class EJavaGuru {}
```

c.
```
import javavirtualmachine.*;
package javacompiler;
class EJavaGuru {
    void method1() {}
    int count;
}
```

d.
```
package javacompiler;
import javavirtualmachine.*;
class EJavaGuru {
    void method1() {}
    int count;
}
```

e
```
#package javacompiler;
$import javavirtualmachine;
class EJavaGuru {
    void method1() {}
    int count;
}
```

f
```
package javacompiler;
import javavirtualmachine;
Class EJavaGuru {
    void method1() {}
    int count;
}
```

Answer: d

Explanation: Answer (a) is incorrect because `#connect` isn't a statement in Java. `#` is used to add comments in UNIX.

Option (b) is incorrect because a package name (`Java compiler`) cannot contain spaces. Also, `java virtual machine` isn't a valid package name to be imported in a class. The package name to be imported cannot contain spaces.

Option (c) is incorrect because a `package` statement should be placed before an `import` statement.

Option (e) is incorrect. `#package` and `$import` aren't valid statements or directives in Java.

Option (f) is incorrect. Java is case-sensitive, so the word `class` is not the same as the word `Class`. The correct keyword to define a class is `class`.

Q1-4. Given the following contents of the Java source code file MyClass.java, select the correct options:

```
// contents of MyClass.java
package com.ejavaguru;
import java.util.Date;
class Student {}
class Course {}
```

- **a** The imported class, `java.util.Date`, can be accessed only in the class `Student`.
- **b** **The imported class, java.util.Date, can be accessed by both the `Student` and `Course` classes.**
- **c** **Both of the classes `Student` and `Course` are defined in the package com.ejavaguru.**
- **d** Only the class `Student` is defined in the package `com.ejavaguru`. The class `Course` is defined in the default Java package.

Answer: b, c

Explanation: You can define multiple classes, interfaces, and enums in a Java source code file.

Option (a) is incorrect. The `import` statement applies to all the classes, interfaces, and enums defined within the same Java source code file.

Option (d) is incorrect. If a `package` statement is defined in the source code file, all of the classes, interfaces, and enums defined within it will exist in the same Java package.

Q1-5. Given the following definition of the class `EJavaGuru`,

```
class EJavaGuru {
    public static void main(String[] args) {
        System.out.println(args[1]+":"+ args[2]+":"+ args[3]);
    }
}
```

what is the output of the previous class, if it is executed using the command:

```
java EJavaGuru one two three four
```

a `one:two:three`
b `EJavaGuru:one:two`
c `java:EJavaGuru:one`
d **`two:three:four`**

Answer: d

Explanation: The command-line arguments passed to the `main` method of a class do not contain the word Java and the name of the class.

Because the position of an array is zero-based, the method argument is assigned the following values:

args[0] -> one
args[1] -> two
args[2] -> three
args[3] -> four

The class prints `two:three:four`.

Q1-6. Which of the following options, when inserted at `//INSERT CODE HERE`, will print out `EJavaGuru`?

```
public class EJavaGuru {
    // INSERT CODE HERE
    {
        System.out.println("EJavaGuru");
    }
}
```

a `public void main (String[] args)`
b `public void main(String    args[])`
c **`static public void main   (String[] array)`**
d `public static void main (String args)`
e `static public main (String args[])`

Answer: c

Explanation: Option (a) is incorrect. This option defines a valid method but not a valid `main` method. The `main` method should be defined as a `static` method, which is missing from the method declaration in option (a).

Option (b) is incorrect. This option is similar to the method defined in option (a), with one difference. In this option, the square brackets are placed after the name of the method argument. The `main` method accepts an array as a method argument, and to define an array, the square brackets can be placed after either the data type or the method argument name.

Option (c) is correct. Extra spaces in a class are ignored by the Java compiler.

Option (d) is incorrect. The `main` method accepts an array of `String` as a method argument. The method in this option accepts a single `String` object.

Option (e) is incorrect. It isn't a valid method definition and doesn't specify the return type of the method. This line of code will not compile.

Q1-7. Select the correct options:

a **You can start the execution of a Java application through the `main` method.**

b The Java compiler calls and executes the `main` method.

c **The Java Virtual Machine calls and executes the `main` method.**

d A class calls and executes the `main` method.

Answer: a, c

Explanation: The Java Virtual Machine calls and executes the `main` method.

Q1-8. A class `Course` is defined in a package `com.ejavaguru`. Given that the physical location of the corresponding class file is /mycode/com/ejavaguru/Course.class and execution takes place within the mycode directory, which of the following lines of code, when inserted at `// INSERT CODE HERE`, will import the `Course` class into the class `MyCourse`?

```
// INSERT CODE HERE
class MyCourse {
    Course c;
}
```

a `import mycode.com.ejavaguru.Course;`

b **`import com.ejavaguru.Course;`**

c `import mycode.com.ejavaguru;`

d `import com.ejavaguru;`

e `import mycode.com.ejavaguru*;`

f `import com.ejavaguru*;`

Answer: b

Explanation: Option (a) is incorrect. The path of the imported class used in an `import` statement isn't related to the class's physical location. It reflects the package and subpackage that a class is in.

Options (c) and (e) are incorrect. The class's physical location isn't specified in the `import` statement.

Options (d) and (f) are incorrect. `ejavaguru` is a package. To import a package and its members, the package name should be followed by `.*`, as follows:

```
import com.ejavaguru.*;
```

Q1-9. Examine the following code:

```
class Course {
    String courseName;
}
class EJavaGuru {
    public static void main(String args[]) {
        Course c = new Course();
        c.courseName = "Java";
        System.out.println(c.courseName);
    }
}
```

Which of the following statements will be true if the variable `courseName` is defined as a `private` variable?

- **a** class `EJavaGuru` will print Java.
- **b** class `EJavaGuru` will print null.
- **c** **class `EJavaGuru` won't compile.**
- **d** class `EJavaGuru` will throw an exception at runtime.

Answer: c

Explanation: If the variable `courseName` is defined as a `private` member, it won't be accessible from the class `EJavaGuru`. An attempt to do so will cause it to fail at compile time. Because the code won't compile, it can't execute.

Q1-10. Given the following definition of the class `Course`,

```
package com.ejavaguru.courses;
class Course {
    public String courseName;
}
```

what's the output of the following code?

```
package com.ejavaguru;
import com.ejavaguru.courses.Course;
class EJavaGuru {
    public static void main(String args[]) {
        Course c = new Course();
        c.courseName = "Java";
        System.out.println(c.courseName);
    }
}
```

- **a** The class `EJavaGuru` will print Java.
- **b** The class `EJavaGuru` will print null.
- **c** **The class `EJavaGuru` will not compile.**
- **d** The class `EJavaGuru` will throw an exception at runtime.

Answer: c

Explanation: The class will fail to compile because a non-`public` class cannot be accessed outside a package in which it is defined. The class `Course` therefore can't be accessed from within the class `EJavaGuru`, even if it is explicitly imported into it. If the class itself isn't accessible, there's no point in accessing a public member of a class.

Q1-11. Given the following code, select the correct options:

```
package com.ejavaguru.courses;
class Course {
    public String courseName;
    public void setCourseName(private String name) {
        courseName = name;
    }
}
```

a **You can't define a method argument as a `private` variable.**
b A method argument should be defined with either `public` or default accessibility.
c For overridden methods, method arguments should be defined with `protected` accessibility.
d None of the above.

Answer: a

Explanation: You can't add an explicit accessibility keyword to the method parameters. If you do, the code won't compile.

JAVA CERTIFICATION

OCA Java SE 7

Programmer I Certification Guide

Mala Gupta

To earn the OCA Java SE 7 Programmer Certification, you need to know your Java inside and out, and to pass the exam it's good to understand the test itself. This book cracks open the questions, exercises, and expectations you'll face on the OCA exam so you'll be ready and confident on test day.

OCA Java SE 7 Programmer I Certification Guide is a comprehensive guide to the 1Z0-803 exam. You'll explore important Java topics as you systematically learn what is required. Each chapter starts with a list of exam objectives, followed by sample questions and exercises designed to reinforce key concepts. It provides multiple ways to digest important techniques and concepts, including analogies, diagrams, flowcharts, and lots of well-commented code.

What's Inside

- Covers all exam topics
- Hands-on coding exercises
- How to avoid built-in traps and pitfalls

Written for developers with a working knowledge of Java who want to earn the OCA Java SE 7 Programmer I Certification.

Mala Gupta has been training programmers to pass Java certification exams since 2006. She holds OCA Java SE7 Programmer I, SCWCD, and SCJP certifications.

"The author knows her stuff and is a great teacher of Java."
—From the Foreword by Jeanne Boyarsky, CodeRanch

"Filled with cool illustrations and neat code examples, this book packs a punch!"
—Ashwin Mhatre
Midasis Technologies

"A very instructive study guide, with a bunch of sample questions and explanations included!"
—Roel De Nijs, Javaroe

"Excellent breakdown of the exam objectives."
—Michael Piscatello
MBP Enterprises, LLC

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
manning.com/OCAJavaSE7ProgrammerICertificationGuide

ISBN 13: 978-1-617291-04-3
ISBN 10: 1-617291-04-8
54999
9 781617 291043

MANNING

$49.99 / Can $52.99 [INCLUDING eBOOK]