SAMPLE CHAPTER

OCA Java SE 8

Programmer I

CERTIFICATION GUIDE

MANNING

Mala Gupta

OCA Java SE 8 Programmer I Certification Guide

by Mala Gupta

Chapter 1

brief contents

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; run a Java program from the command line; including console output.	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.
[6.4] 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.
[7.5] 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.
[1.5] Compare and contrast the features and components of Java such as: platform independence, object orientation, encapsulation, etc.	The features and components that are relevant or irrelevant to Java.

Imagine you're setting up a new IT organization that works with multiple developers. To ensure smooth and efficient working, you'll define a structure for your organization and a set of departments with separate 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 might work with a well-defined structure and a set of rules to deliver the best results.

Similarly, Java has a well-defined structure and hierarchy. 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 some 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 8 Programmer I exam, you'll be asked questions on the structure of a Java class, packages, importing classes, and applying access and nonaccess modifiers and features and components of Java. Given that information, this chapter will cover the following:

- 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
- Features and components of Java

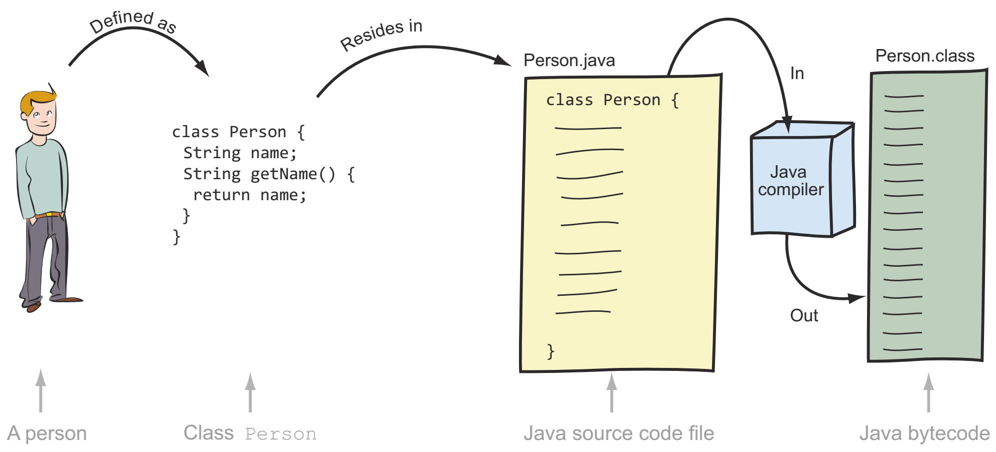
1.1 The structures of a Java class and a source code file

[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 structures 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's 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.



```
class Person {
 String name;
 String getName() {
  return name;
 }
}
```

```
class Person {
}
```

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

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).

1.1.1 Structure of a Java class

The OCA Java SE 8 Programmer I exam will question you on the structure and components of a Java source file and the classes or interfaces that you can define in it. Figure 1.2 shows the components of a Java class file (interfaces are covered in detail in chapter 6).

In this section, I'll discuss all Java class file components. Let's get started with the `package` statement.

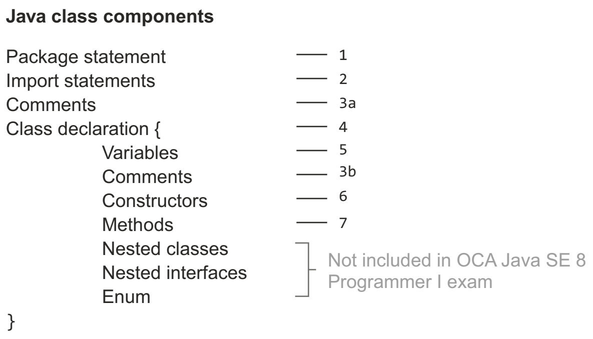


Figure 1.2 Components of a Java class

NOTE The code in this book doesn't include a lot of spaces—it imitates the kind of code that you'll see on the exam. But when you work on real projects, I strongly recommend that you use spaces or comments to make your code readable.

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 package statement should be the first statement in a class.

The rest of the code for class Course

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

The `package` statement can't 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, that is, `packageName.anySubpackageName.ClassName`. For example, the fully qualified name of class `String` is `java.lang.String`.

Because using fully qualified names 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.

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) *class diagram* in figure 1.3.

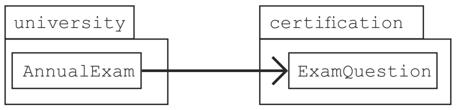


Figure 1.3 UML representation of the relationship between class `AnnualExam` and `ExamQuestion`

NOTE A UML class diagram represents the static view of an application. It shows entities like packages, classes, interfaces, and their attributes (fields and methods) and also depicts the relationships between them. It shows which classes and interfaces are defined in a package. It depicts the inheritance relationship between classes and interfaces. It can also depict the associations between them—when a class or an interface defines an attribute of another type. All UML representations in this chapter are class diagrams. The exam doesn't cover UML diagrams. But using these quick and simple diagrams simplifies the relationship between Java entities—both on the exam and in your real-world projects.

NOTE Throughout this book, **bold font** will be used to indicate specific parts of code that we're discussing, or changes or modifications in code.

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 classes `AnnualExam` and `ExamQuestion` are related, as depicted in figure 1.4?

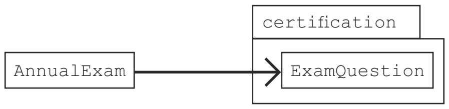


Figure 1.4 Relationship between the packageless class `AnnualExam` and `ExamQuestion`

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

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

Multiline comment with special characters in it

In the preceding code, the comments don't start with an asterisk on every line. But 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're placed at the end of a line of code or on a blank line. The text between // and the end of the line is treated as a comment, which you'd 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

Though usage of multiline comments in the following code is uncommon, the exam expects you to know that the code is valid:

```
String name = /* Harry */ "Paul";
System.out.println(name);
```

Outputs Paul

Here's what happens if you include multiline comments within quotes while assigning a string value:

```
String name = "/* Harry */ Paul";
System.out.println(name);
```

Outputs /* Harry */ Paul

When included within double quotes, multiline comments are treated as regular characters and not as comments. So the following code won't compile because the value assigned to variable `name` is an unclosed string literal value:

```
String name = "Shre /* ya
                */ Paul";
System.out.println(name);
```

Won't compile

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;
    String builtBy;    // individual/ architect
}
// another comment
```

❶ End-of-line comment within a multiline comment

❷ End-of-line comment

❸ End-of-line comment

❹ 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.

Javadoc comments

Javadoc comments are special comments that start with `/**` and end with `*/` in a Java source file. These comments are processed by Javadoc, a JDK tool, to generate API documentation for your Java source code files. To see it in action, compare the API documentation of the class `String` and its source code file (String.java).

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.

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. We'll go through 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 declaration can be illustrated as shown in figure 1.5.

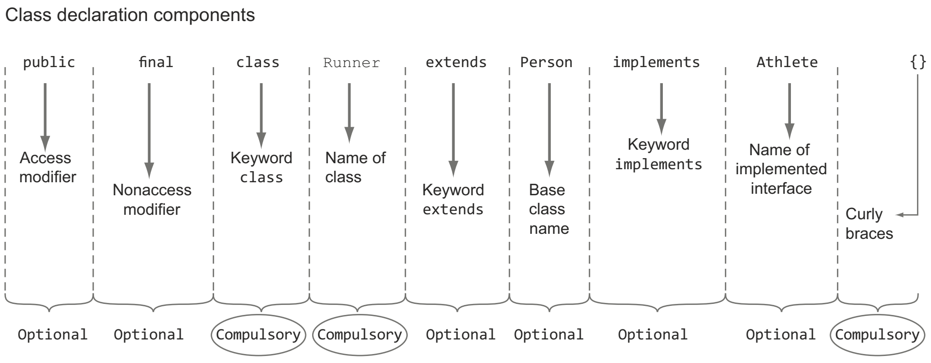


Figure 1.5 Components of a class declaration

Table 1.1 summarizes the compulsory and optional components.

Table 1.1 Components of a class declaration

Mandatory	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 names of the interfaces being implemented

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 attributes and behavior of an object. The attributes 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` (uppercase *C*) to define a class.
- The state of a class is defined using attributes or instance variables.
- It isn't compulsory to define all attributes of a class before defining its methods (the variable `weight` is defined after `Phone`'s constructor). But this is far from being optimal for readability.
- The behavior is defined using methods, which may include method parameters.
- 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 definition of the class `Phone` in the previous example. Because the variables `model`, `company`, and `weight` are used to store the state of an object (also called an *instance*), they're 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* can be used to manipulate 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 Java entities such as a class, interface, enum, and annotation.

NOTE Java annotations are not on the exam and so won't be discussed in this book.

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 AN INTERFACE IN A JAVA SOURCE CODE FILE

An interface specifies a contract for the classes to implement. You can compare implementing an interface to signing a contract. An interface is a grouping of related methods and constants. Prior to Java 8, interface methods were implicitly abstract. But starting with Java version 8, the methods in an interface can define a default implementation. With Java 8, interfaces can also define `static` methods.

Here's a quick example to help you understand the essence of interfaces. No matter which brand of television each 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`. Remember, Java is case-sensitive, so you can't use the word `Interface` (with a capital *I*) to define an interface. This section provides a brief overview of interfaces. You'll work with interfaces in detail in chapter 6.

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 Java source code file or multiple such entities. 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

No particular order is required to define multiple classes or interfaces 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.

When you define a `public` class or an interface in a Java source file, the names of the class or interface and Java source file must match. 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 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 this modification affects your code.

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

```
// 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 can't 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(s):

```
// 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 nonpublic class will compile the code.
- **e** Changing interface `Multiple2` to a nonpublic interface 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 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; run a Java program from the command line; including console output.

The OCA Java SE 8 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. You also need to know how to execute a Java program from the command line.

1.2.1 Executable Java classes versus non-executable Java classes

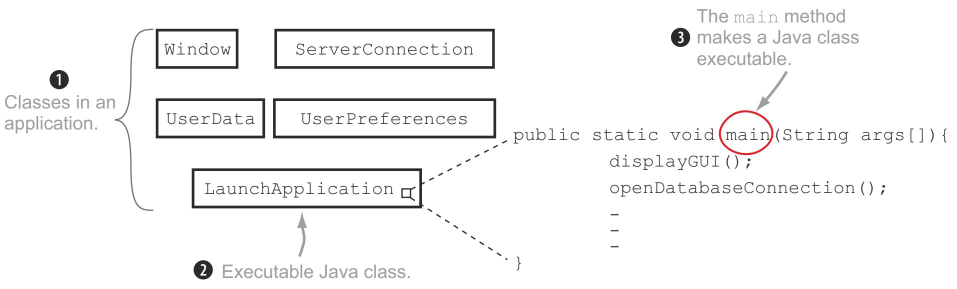
Doesn't the Java Virtual Machine execute all the Java classes when they are used? If so, what is a non-executable Java class?

An executable Java class, when handed over to the JVM, starts its execution at a particular point in the class—the `main` method. The JVM starts executing the code that's defined in the `main` method. You can't hand over a non-executable Java class (class without a `main` method) to the JVM and ask it to execute it. In this case, the JVM won't know which method to execute because no entry point is marked.

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.

NOTE A Java application can define more than one executable class. We choose one (and exactly one) when the time comes to start its execution by the JVM.



```
public static void main(String args[]){
        displayGUI();
        openDatabaseConnection();
        -
        -
        -
}
```

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 *The main method*

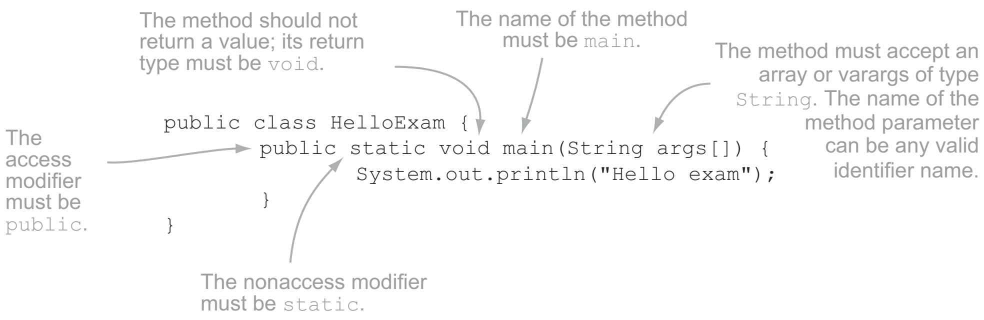
The first requirement in creating an executable Java application is to create a class with a method whose signature (name and method arguments) matches the `main` method, defined as follows:

```
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 (varargs) of type `String`.

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


The method should not return a value; its return type must be void.
The name of the method must be main.
The method must accept an array or varargs of type String. The name of the method parameter can be any valid identifier name.
The access modifier must be public.
The nonaccess modifier must be static.

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

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's valid to define args as a variable argument.

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

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

This won't compile. Ellipsis must 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. 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 placements of the keywords public and static are interchangeable.

NOTE Though both public static and static public are the valid order of keywords to declare the main method, public static is more common and thus more readable.

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

```
Hello exam
```

If a class defines a main method that doesn't match the signature of *the* main method, it's referred to as an *overloaded method* (overloaded methods are discussed in detail in chapter 3). Overloaded methods are methods with the same name but different signatures. For a quick example, class `HelloExam` can define multiple methods with the method name `main`:

```
public class HelloExam {
    public static void main(String args) {
        System.out.println("Hello exam 2");
    }
    public static void main(String args[]) {
        System.out.println("Hello exam");
    }
    public static void main(int number) {
        System.out.println("Hello exam 3");
    }
}
```

JVM will execute this main method.

On execution, JVM will execute *the* `main` method, resulting in the output `Hello exam`.

1.2.3 Run a Java program from the command line

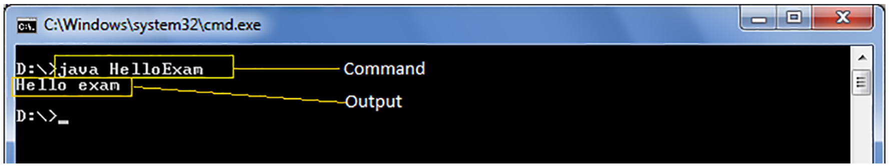
Almost all Java developers work with an Integrated Development Environment (IDE). This exam, however, expects you to understand how to execute a Java application, or an executable Java class, using the command prompt. For this reason, I suggest you work with a simple text editor and command line (even if this might never be the approach you use in the real world).

> **NOTE** 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.

Let's revisit the code shown in figure 1.7:

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

To execute the preceding code using a command prompt, issue the command `java HelloExam`, as shown in figure 1.8.



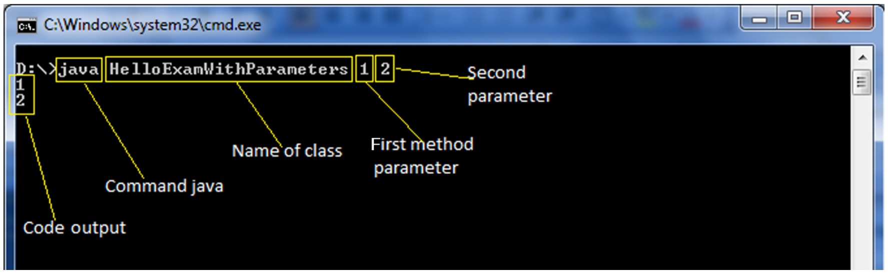
```
D:\>java HelloExam
Hello exam

D:\>_
```

Figure 1.8 Using the command prompt to execute a Java application

I mentioned that 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:

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

Now let's execute the preceding code using the command prompt, as shown in figure 1.9.



```
D:\>java HelloExamWithParameters 1 2
1
2
```

Figure 1.9 Passing command parameters to a `main` method

As you can see from the output shown in figure 1.9, the keyword `java` and the name of the class aren't passed as command parameters to the `main` method. The OCA Java SE 8 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 to the `main` method are also called command-line parameters or command-line values. As the name implies, these values are passed to a method from the command line.

If you weren't able to follow the code with respect to the arrays and the 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, as usual.

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

If you've programmed in languages like C, you might get confused by the command-line parameters. Programming languages like C pass the name of a *program* as a command-line argument to the `main` method. But Java doesn't pass the name of the *class* as an argument to the `main` method.

1.3 Java packages

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

This exam covers importing packages into other classes. But with more than a decade and a half of experience, I've learned that before starting to *import* other packages into your own code, it's important to understand what packages are, the difference between classes that are defined in a package and the classes that aren't defined in a package, and why you need to import packages 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 in your life? 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.

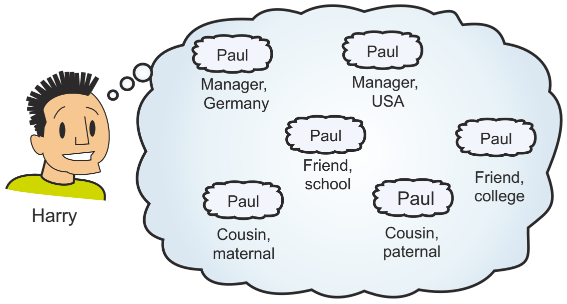


Figure 1.10 Harry knows six Pauls!

Similarly, you can use a package to group together a related set of classes and interfaces (I won't 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 healthcare systems. Further, you can create subpackages within these packages, such as separate subpackages for GUIs, database access, networking, and so on.

NOTE In real-life projects, you'll rarely work with a package-less class or interface. Almost all organizations that develop software have strict package-naming rules, which are often documented.

All classes and interfaces are defined in a package. 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 indicate that a class or an interface is defined in a package by using the `package` statement as the first statement in code. Here's an example:

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

Variables and methods

The class in the preceding 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 a UML class diagram depicting the relationship of the package `certification` to the class `ExamQuestion` and the interface `MultipleChoice`.

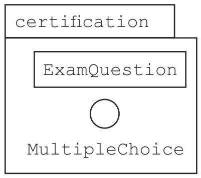


Figure 1.11 A UML class diagram showing the relationship shared by package `certification`, class `ExamQuestion`, and interface `MultipleChoice`

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 the folks at Oracle were to 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
}
```

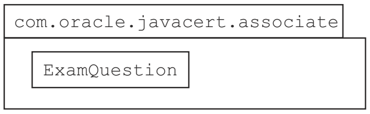


Figure 1.12 A subpackage and its corresponding class definition

A package is made of multiple sections that go from the more-generic (left) to the more-specific (right). The package name `com.oracle.javacert.associate` follows a package-naming convention recommended by Oracle and shown in table 1.2.

Table 1.2 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

Here are 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 classes and interfaces defined in a package, 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 are defined in the same package. They can't be defined in separate packages.

NOTE A fully qualified name for a class or interface is formed by prefixing its package name with its name (separated by a dot). The fully qualified name of the 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 classes and interfaces defined in packages must 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 rules shown in figure 1.13.

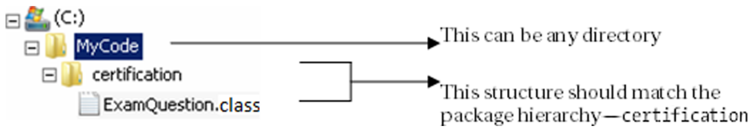

(C:)
MyCode
certification
ExamQuestion.class
This can be any directory
This structure should match the package hierarchy—certification

Figure 1.13 Matching directory structure and package hierarchy

For the package example shown in figure 1.13, note that there isn't any constraint on the location of the base directory in which the directory structure is defined, as shown in figure 1.14.

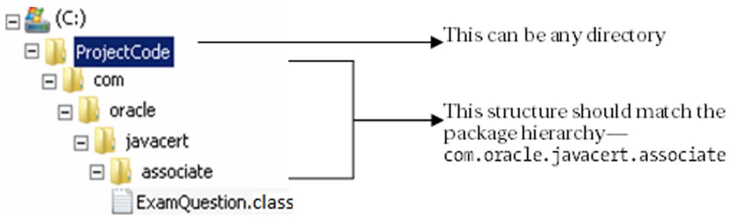

(C:)
ProjectCode
com
oracle
javacert
associate
ExamQuestion.class
This can be any directory
This structure should match the package hierarchy—com.oracle.javacert.associate

Figure 1.14 There's no constraint on the location of the base directory to define directories corresponding to package hierarchy.

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.

NOTE You needn't 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 helpful with your projects at work. The exam expects you to spot code with compilation errors, which isn't easy to do if you didn't learn how to do it without an IDE (IDEs usually include code autocorrection or autocompletion features).

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 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 reference 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 1.15.

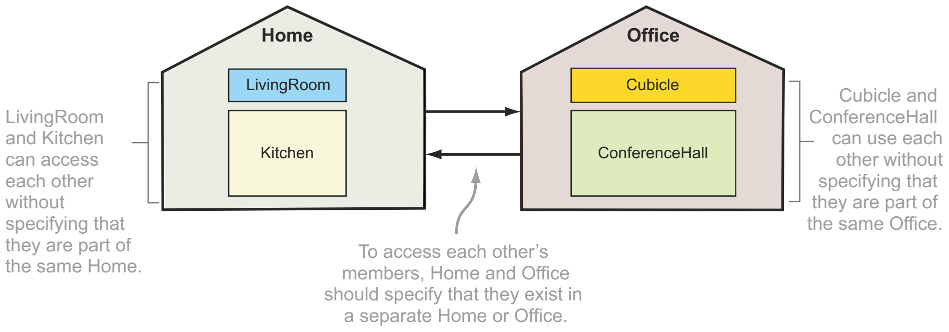

Home
LivingRoom
Kitchen
Office
Cubicle
ConferenceHall
LivingRoom and Kitchen can access each other without specifying that they are part of the same Home.
Cubicle and ConferenceHall can use each other without specifying that they are part of the same Office.
To access each other's members, Home and Office should specify that they exist in a separate Home or Office.

Figure 1.15 To refer to each other's members, Home and Office should specify that they exist in separate places.

To refer to the LivingRoom in Cubicle, you *must* specify its complete location, as shown in the left part of the figure 1.16. 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 and Office are like Java packages, and this notice is the equivalent of the `import` statement. Figure 1.16 shows the difference in using fully qualified names and simple names for LivingRoom in Cubicle.

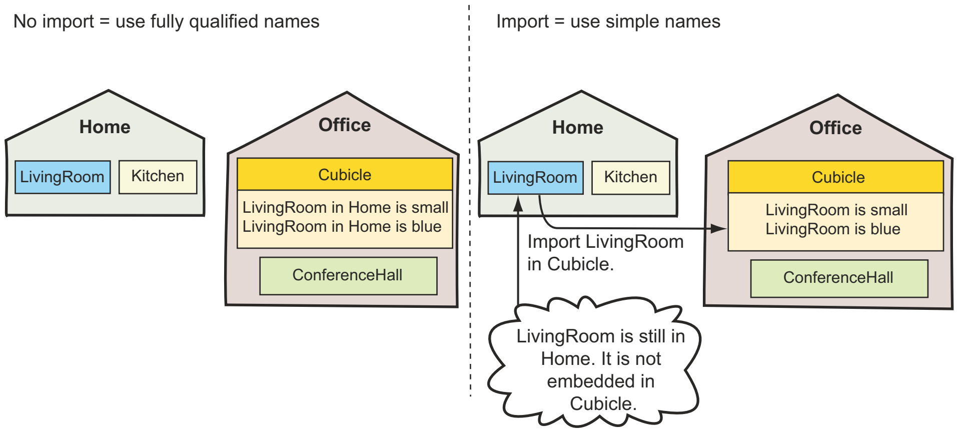

No import = use fully qualified names
Import = use simple names
Home
LivingRoom
Kitchen
Office
Cubicle
LivingRoom in Home is small
LivingRoom in Home is blue
ConferenceHall
Home
LivingRoom
Kitchen
Office
Cubicle
LivingRoom is small
LivingRoom is blue
ConferenceHall
Import LivingRoom in Cubicle.
LivingRoom is still in Home. It is not embedded in Cubicle.

Figure 1.16 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.

Let's implement the preceding 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`. Class `Cubicle` uses (is associated to) class `Living-Room` in the package `home`, as shown in figure 1.17.

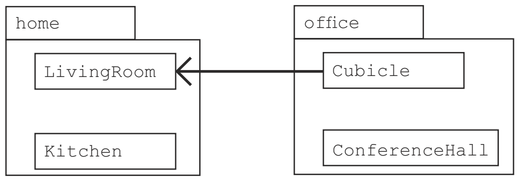


Figure 1.17 A UML representation of classes `LivingRoom` and `Cubicle`, defined in separate packages, with their associations

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.

1.3.4 Using packaged classes without using the import statement

It's 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.

But using a fully qualified class name can clutter your code if you create multiple variables of interfaces and classes defined in other packages. *Don't* use this approach in real projects.

EXAM TIP You don't need an explicit `import` statement to use members from the `java.lang` package. Classes and interfaces in this package are automatically imported in *all* other Java classes, interfaces, or enums.

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 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;
}
```

import statement not required

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.

An alternate approach (which works well in real projects) is to use the `import` definition with the class or interface that you use more often and fully reference the one that you use just from time to time:

```
import java.util.Date;          <-- import class you use often
class AnnualExam {
    Date date1;                 <-- Use simple class name for java.util.Date
    java.sql.Date date2;        <-- Use fully qualified name for java.sql.Date
}
```

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 1.18.

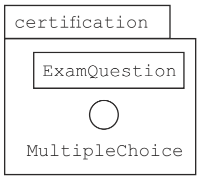


Figure 1.18 A UML representation of the `certification` package

Examine the following code for the class `AnnualExam`:

```
import certification.ExamQuestion;    <-- Imports only the class ExamQuestion
class AnnualExam {
    ExamQuestion eq;                  <-- Compiles OK
    MultipleChoice mc;                <-- Will not compile
}
```

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

```
import certification.*;              Imports all classes and
                                     interfaces from certification
class AnnualExam {
    ExamQuestion eq;                 Compiles OK
    MultipleChoice mc;               Also compiles OK
}
```

NOTE When overused, using an asterisk to import all members of a package has a drawback. It may be harder to figure out which imported class or interface comes from which package.

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 *The import statement doesn't import the whole package tree*

You can't import classes from a subpackage by using an asterisk in the `import` statement. For example, the UML notation in figure 1.19 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`.

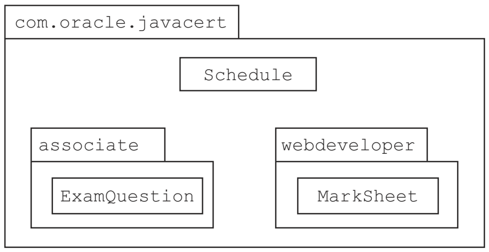


Figure 1.19 A UML representation of package `com.oracle.javacert` and its subpackages

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
```

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 include a package statement in your classes or interfaces? In that case, they become part of 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 aren't defined in an explicit package, can use each other if they're 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're defined within the same directory or not.

EXAM TIP Members of a named package can't access classes and interfaces defined in the *default* package.

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. Although accessible using an instance, the `static` members are better accessed by prefixing their name with the class or interface names. By using `static import`, you can drop the prefix and just use the name of the `static` variable or method. In the following code, class `ExamQuestion` defines a `public static` variable `marks` and a `public static` method `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 marks variable 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;      <-- Correct statement is import static, not static import
class AnnualExam {
    AnnualExam() {
        marks = 20;       <-- Access variable marks without prefixing it with its class name
    }
}
```

EXAM TIP This feature is called *static imports*, but the syntax is import static.

To access all public and static members of class ExamQuestion in class AnnualExam without importing each of them individually, 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;      | Accesses variable marks and method print
        print();         | without prefixing them with their class names
    }
}
```

Because the variable marks and method print are defined as public members, they're accessible to the class AnnualExam. By using the import static statement, you don't have to prefix them with their class name.

NOTE On real projects, avoid overusing static imports; otherwise, the code might become a bit confusing about which imported component comes from which class.

The accessibility of a class, an interface, and their methods and variables is determined by their access modifiers, which are covered in the next section.

1.4 Java access modifiers

[6.4] Apply access modifiers

In this section, we'll cover all 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 accessibility 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 1.20.

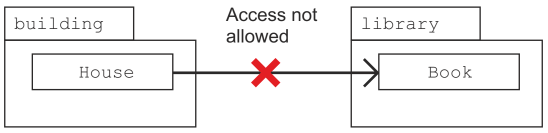


Figure 1.20 The nonpublic class `Book` can't be accessed outside the package `library`.

```
package building;
class House {}
package library;
class Book {}
```

With the current class definitions, the class `House` can't 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 by 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 8 Programmer I exam.

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 within the same or separate packages. By using the appropriate access modifiers, you can limit access to your class or interface and their members.

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 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 has four access levels:

- `public` (least restrictive)
- `protected`

- default
- `private` (most restrictive)

To understand all of these access levels, we'll use the same set of classes: `Book`, `CourseBook`, `Librarian`, `StoryBook`, and `House`. Figure 1.21 depicts these classes using UML notation.

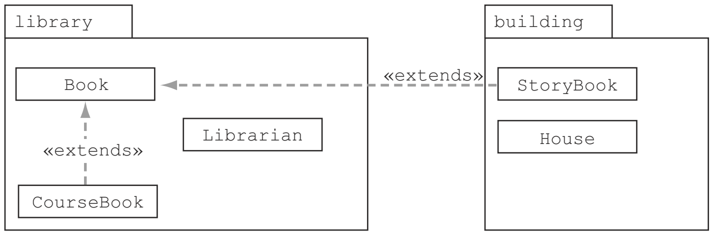


Figure 1.21 A set of classes and their relationships to help you understand access modifiers

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 I cover each of the access modifiers, I'll add a set of instance variables and a method to the class `Book` with the relevant access modifier. I'll then define code in other classes to access class `Book` and its members.

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 1.22 shows the UML notation.

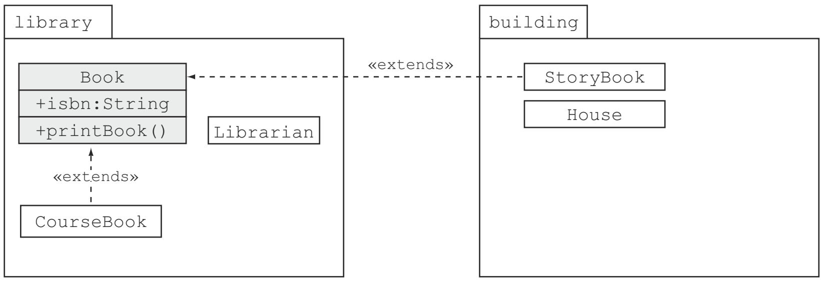


Figure 1.22 Understanding the `public` access modifier

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*.

> **NOTE** The term *unrelated classes* in this chapter refers to classes that don't share inheritance relation. For instance, classes `House` and `Book` are unrelated, if neither `House` derives from `Book` nor `Book` derives from `House`.

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`:

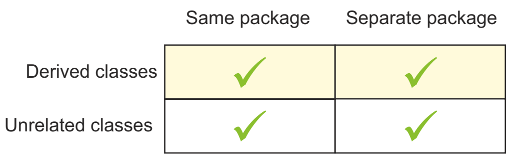
```
package building;
import library.Book;
public class House {
    House() {
        Book book = new Book();
        String value = book.isbn;
        book.printBook();
    }
}
```

Class Book is accessible to class House.

Variable isbn is accessible in House.

Method printBook is accessible in House.

In the preceding example, class `Book` and its `public` members—instance variable `isbn` and method `printBook`—are accessible to class `House`. They are also accessible to the other classes: `StoryBook`, `Librarian`, `House`, and `CourseBook`. Figure 1.23 shows the classes that can access a `public` class and its members.



	Same package	Separate package
Derived classes	✓	✓
Unrelated classes	✓	✓

Figure 1.23 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 a method `modifyTemplate` to the class `Book`. Figure 1.24 shows the class representation.

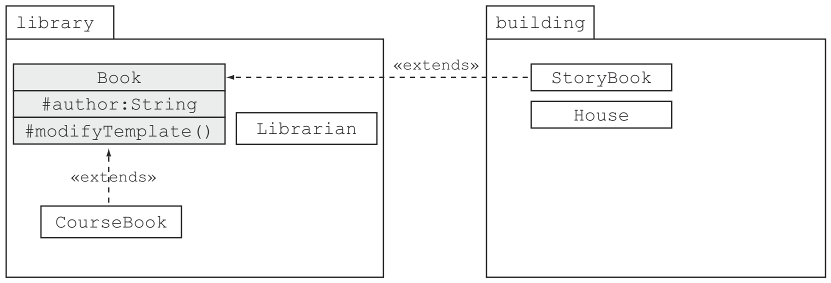


Figure 1.24 Understanding the `protected` access modifier

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

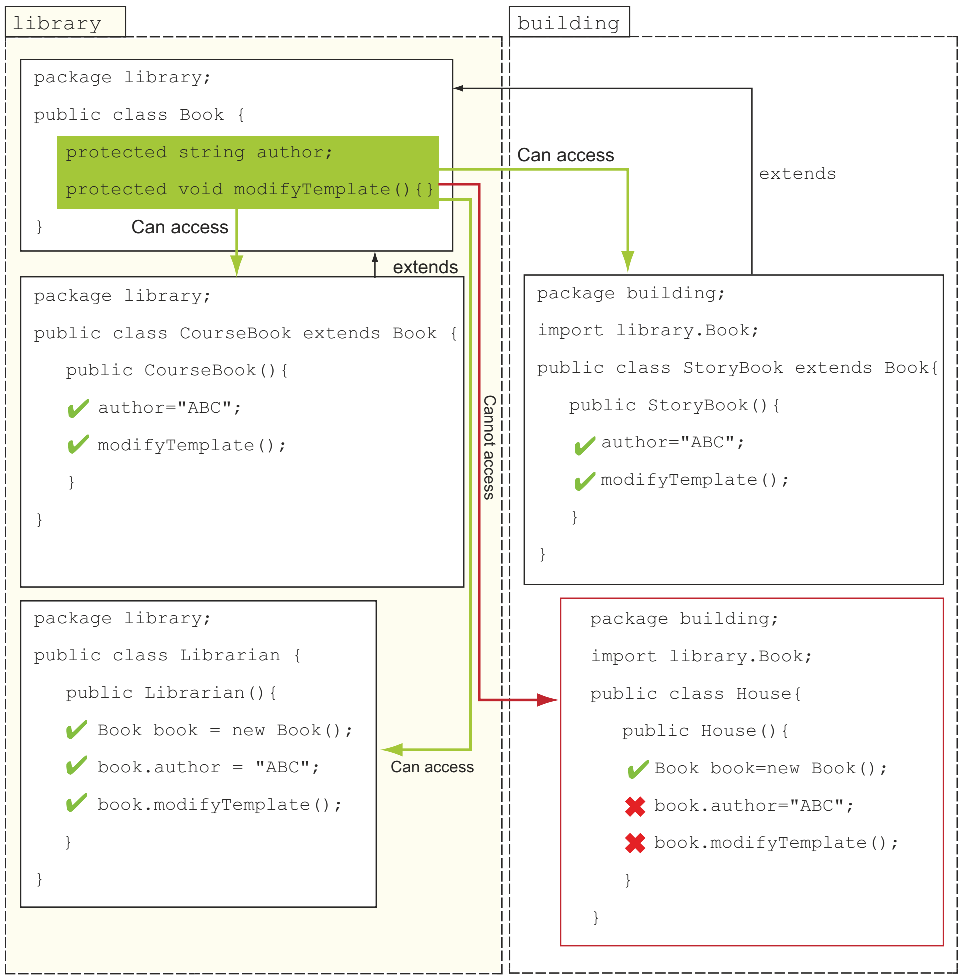
Class `House` fails compilation for trying to access the method `modifyTemplate` and the variable `author`. Following is the compilation error message:

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

NOTE Java code fails compilation because of syntax errors. In such a case, the Java compiler notifies the offending code with its line number and a short description of the error. The preceding code is output from the compilation process. This book uses the command prompt to compile all Java code.

A derived class inherits the protected members of its base class, irrespective of the packages in which they're defined.

Notice that the derived classes `CourseBook` and `StoryBook` inherit class `Book`'s protected member variable `author` and method `modifyTemplate()`. If class `StoryBook`



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

```
package library;
public class CourseBook extends Book {
    public CourseBook(){
    ✓ author="ABC";
    ✓ modifyTemplate();
    }
}
```

```
package library;
public class Librarian {
    public Librarian(){
    ✓ Book book = new Book();
    ✓ book.author = "ABC";
    ✓ book.modifyTemplate();
    }
}
```

```
package building;
import library.Book;
public class StoryBook extends Book{
    public StoryBook(){
    ✓ author="ABC";
    ✓ modifyTemplate();
    }
}
```

```
package building;
import library.Book;
public class House{
    public House(){
    ✓ Book book=new Book();
    ✗ book.author="ABC";
    ✗ book.modifyTemplate();
    }
}
```

Figure 1.25 Access of `protected` members of the class `Book` in unrelated and derived classes, from the same and separate packages

tries to instantiate `Book` using a reference variable and then tries to access its protected variable `author` and method `modifyTemplate()`, it won't compile:

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

	Same package	Separate package	
Derived classes	✓	✓ Using inheritance	✗ Using reference variable
Unrelated classes	✓	✗	

Figure 1.26 Classes that can access protected members

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. The default access is also referred to as *package-private*. Think of a package as your home, classes as rooms, and things in rooms as variables with default access. These things aren't limited to one room—they can be accessed across all the rooms in your home. But they're still private to your home—you wouldn't want them to be accessed outside your home. Similarly, when you define a package, you might want to make members of classes accessible to all the other classes across the same package.

NOTE Although the package-private access is as valid as the other access levels, in real projects it often appears as the result of inexperienced developers forgetting to specify the access mode of Java components.

Let's define an instance variable `issueCount` and a method `issueHistory` with default access in class `Book`. Figure 1.27 shows the class representation with these new members.

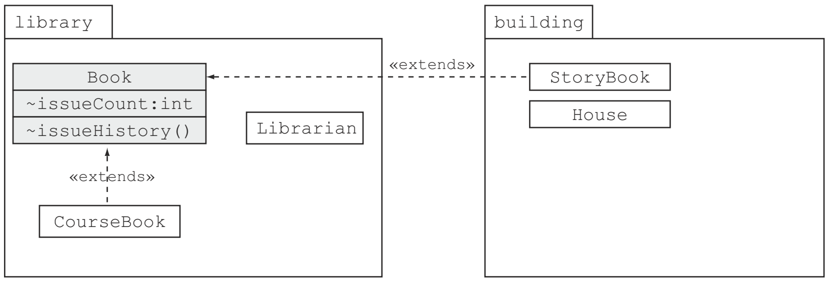


Figure 1.27 Understanding class representation for default access

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):

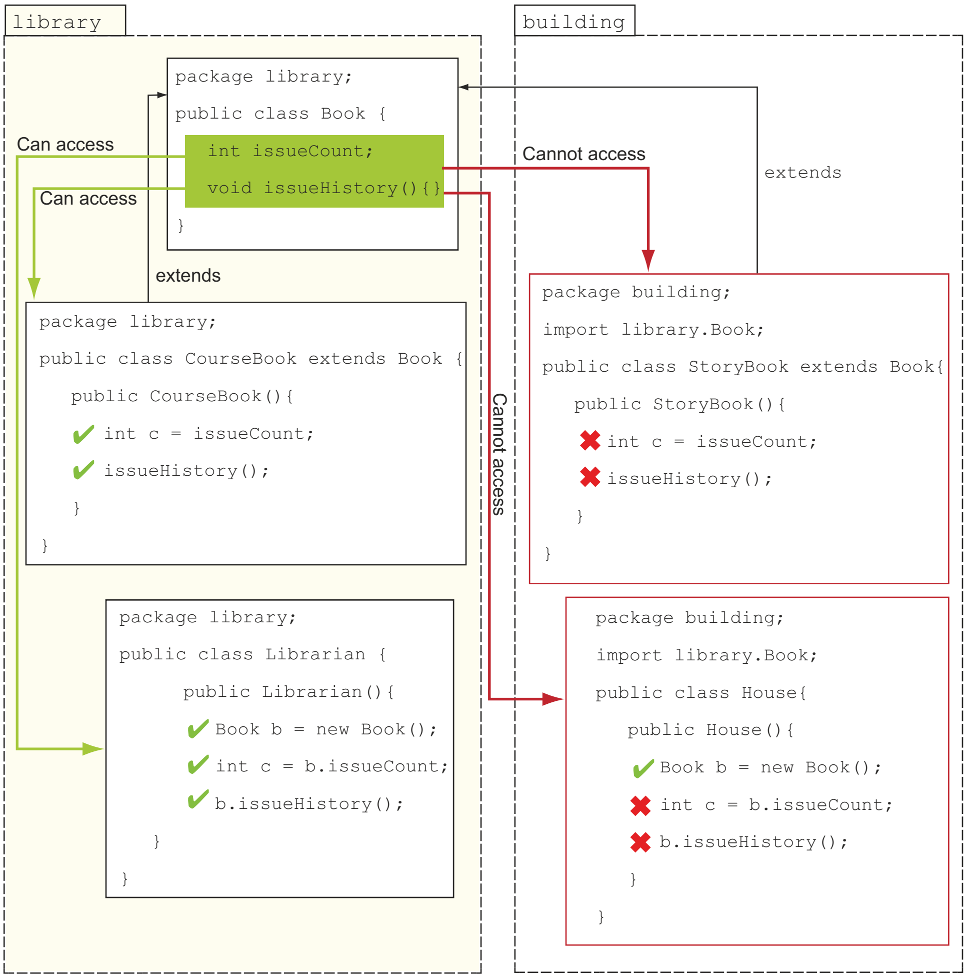
```
package library;
public class Book {
    int issueCount;
    void issueHistory()  {}
}
```

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 1.28.



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

```
package library;
public class CourseBook extends Book {
    public CourseBook(){
    ✓ int c = issueCount;
    ✓ issueHistory();
    }
}
```

```
package library;
public class Librarian {
       public Librarian(){
       ✓ Book b = new Book();
       ✓ int c = b.issueCount;
       ✓ b.issueHistory();
    }
}
```

```
package building;
import library.Book;
public class StoryBook extends Book{
    public StoryBook(){
     ✖ int c = issueCount;
     ✖ issueHistory();
    }
}
```

```
package building;
import library.Book;
public class House{
    public House(){
     ✓ Book b = new Book();
     ✖ int c = b.issueCount;
     ✖ b.issueHistory();
    }
}
```

Figure 1.28 Access of members with default access to the class `Book` in unrelated and derived classes from the same and separate packages

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();
            ^
```

Class House is unaware of the existence of issueHistory()—it fails compilation with the following error message:

```
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?

Consider this situation: Assume that Superfast Burgers opens a new outlet on a beautiful island and offers free meals to people from all over the world, which obviously includes inhabitants of the island. But the island is inaccessible by all means (air and water). Would awareness of 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 1.29.

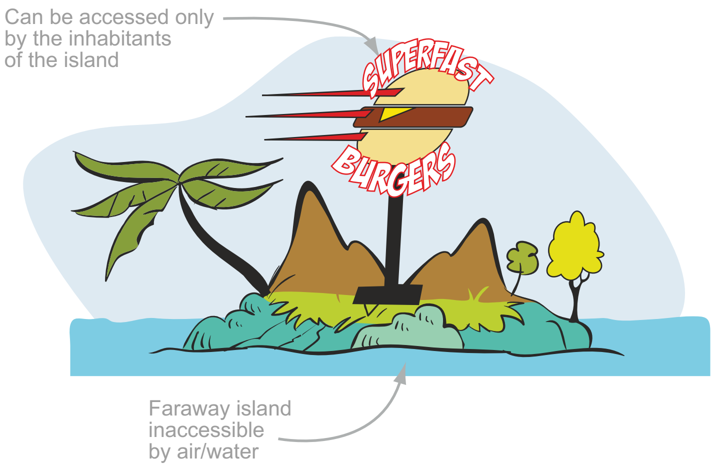

Can be accessed only by the inhabitants of the island
SUPERFAST
BURGERS
Faraway island inaccessible by air/water

Figure 1.29 This Superfast Burgers can't be accessed from outside the island because the island is inaccessible by air and water.

The island is like a package in Java, and Superfast Burgers is like a class defined with default access. In the same way that Superfast Burgers can't 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's 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 class `Book` remains the same for the classes `CourseBook` and `Librarian`, which are defined in the same package. But 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.

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 class `StoryBook`:

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

Book isn't accessible in StoryBook.

StoryBook can't extend Book.

Figure 1.30 shows which classes can access members of a class or interface with default (package) access.

	Same package	Separate package
Derived classes	✓	✗
Unrelated classes	✓	✗

Figure 1.30 The classes that can access members 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 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're defined.

Let's see this in action by adding a `private` method `countPages` to the class `Book`. Figure 1.31 depicts the class representation using UML.

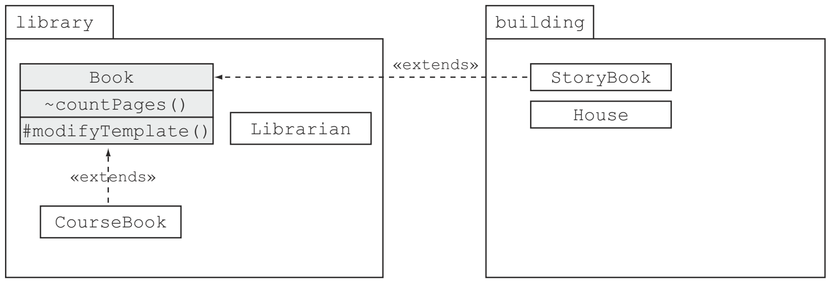


Figure 1.31 Understanding the `private` access modifier

Examine the following definition of the class `Book`:

```
package library;
class Book {
    private void countPages() {}          // private method
    protected void modifyTemplate() {
        countPages();                     // Only Book can access its own private method countPages.
    }
}
```

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 `CourseBook` because both of these classes are defined in the same package, and `CourseBook` extends the class `Book`. Here's the code of `CourseBook`:

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

CourseBook extends Book.

CourseBook can't access private method countPages.

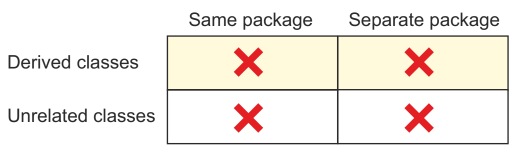
Because the class `CourseBook` tries to access private members of the class `Book`, it won't compile. Similarly, if any of the other classes (`StoryBook`, `Librarian`, `House`, or `CourseBook`) tries to access the `private` method `countPages()` of class `Book`, it won't compile.

Here's an interesting situation: do you think a `Book` instance can access its private members using a reference variable? The following code won't compile—even though variable `b1` is of type `Book`, it's trying to access its private method `countPages` outside `Book`:

```
class TestBook {
    public static void main(String args[]) {
        Book b1 = new Book();
        b1.countPages();
    }
}
```

Won't compile

Figure 1.32 shows the classes that can access the `private` members of a class.



	Same package	Separate package
Derived classes	✗	✗
Unrelated classes	✗	✗

Figure 1.32 No classes can access `private` members of another class

NOTE For your real projects, it *is* possible to access private members of a class outside them, using *Java reflection*. But Java reflection isn't on the exam. So don't consider it when answering questions on the accessibility of private members.

1.4.6 Access modifiers and Java entities

Can every access modifier be applied to all the Java entities? The simple answer is *no*. Table 1.3 lists the Java entities and the access modifiers that can be used with them.

Table 1.3 Java entities and the access modifiers that can be applied to them

Entity name	`public`	`protected`	`private`
Top-level class, interface, enum	✓	✗	✗
Class variables and methods	✓	✓	✓
Instance variables and methods	✓	✓	✓
Method parameter and local variables	✗	✗	✗

What happens if you try to code the combinations for an X in table 1.3? None of these combinations will compile. Here's the code:

```
protected class MyTopLevelClass {}
private class MyTopLevelClass {}
protected interface TopLevelInterface {}
```

Won't compile—top-level class and interfaces can't be defined with protected and private access.

```
void myMethod(private int param) {}
void myMethod(int param) {
    public int localVariable = 10;
}
```

Won't compile—method parameters and local variables can't be defined using any explicit access modifiers.

Watch out for these combinations on the exam. It's simple to insert these small and invalid combinations in any code snippet and still make you believe that you're being tested on a rather complex topic like threads or concurrency.

EXAM TIP Watch out for invalid combinations of a Java entity and an access modifier. Such code won't compile.

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

[7.5] 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 behavior of a Java class and its members.

For example, if you add the keyword `abstract` to the definition of a class, it can't be instantiated. 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 8 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. 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 can't 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. 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` can be used with the declaration of a class, variable, or method. It can't be used with the declaration of an interface.

FINAL CLASS

A class that's marked `final` can't be extended by another class. The class `Professor` won't 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 can't 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, which might change the state of the object that it refers to. 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 (that modify its state):

```
class Person {
    final StringBuilder name = new StringBuilder("Sh");
    Person() {
        name.append("reya");          <-- Can call methods on a final variable that change its state
        name = new StringBuilder();   <-- 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() {                     <-- Won't compile
        System.out.println("Alpha.. beta.. gamma");
    }
}
```

If a method in a derived class has the same method signature as its base class's method, it's 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` can be applied to the declarations of variables, methods, classes, and interfaces. We'll examine each of them in following sections.

STATIC VARIABLES

`static` variables belong to a class. They're 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 the objects of a class.

NOTE A class and an interface can declare `static` variables. This section covers declaration and usage of `static` variables that are defined in a class. Chapter 6 covers interfaces and their `static` variables in detail.

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 1.33.

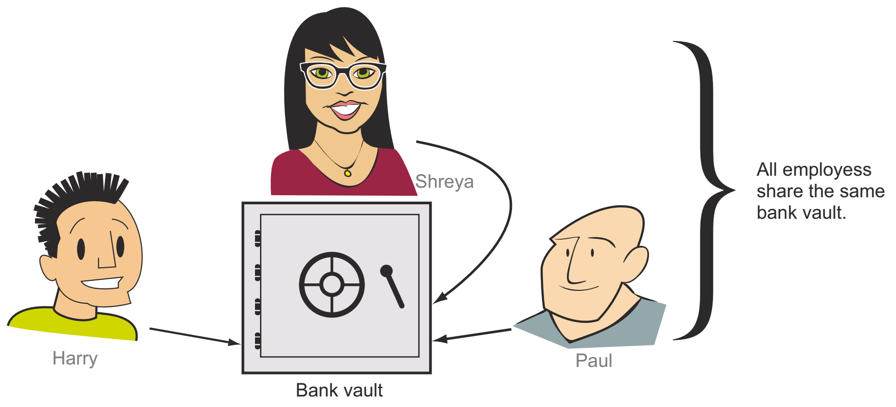


Figure 1.33 Comparing a shared bank vault with a `static` variable

Figure 1.34 defines a class `Emp` that defines a non-`static` variable `name` and a `static` variable `bankVault`.

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

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

Figure 1.34 Definition of the class `Emp` with a `static` variable `bankVault` and non-`static` variable `name`

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 1.34) and modifies the value of the variable `bankVault` using these separate objects:

```
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 emp1 is assigned a value of 10.

Variable bankVault of variable emp2 is assigned a value of 20.

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.bank-Vault` all refer to the *same* `static` attribute: `bankVault`.

> **EXAM TIP** 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 preferred way to access them is by using the class name. The `static` and `final` nonaccess modifiers can be used together 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

Although you can define a constant as a non-`static` member, it's common practice to define constants as `static` members, because 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.

It's a common practice to 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;
}
```

The following utility (`static`) method doesn't define input 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 nonprivate `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.

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

static variable referencing a static method

Non-static method using static method

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

Table 1.4 summarizes the access capabilities of `static` and non-`static` members.

Table 1.4 Access capabilities of `static` and non-`static` members

Member type	Can access `static` attribute or method?	Can access non-`static` attribute or method?
`static`	Yes	No
Non-`static`	Yes	Yes

ACCESSING STATIC MEMBERS FROM A NULL REFERENCE

Because `static` variables and methods belong to a class and not to an instance, you can access them using variables, which are initialized to `null`. Watch out for such questions in the exam. Such code won't throw a runtime exception (`NullPointerException` to be precise). In the following example, the reference variable `emp` is initialized to `null`:

```
class Emp {
    String name;
    static int bankVault;
    static int getBankVaultValue() {
        return bankVault;
    }
}
class Office {
    public static void main(String[] args) {
        Emp emp = null;
        System.out.println(emp.bankVault);
        System.out.println(emp.getBankVaultValue());
    }
}
```

Outputs 0

EXAM TIP You can access `static` variables and methods using a `null` reference.

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 8 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 {}                 <-- Also known as a
    static interface MyInterface {}             static nested class
}
```

The next section covers features of Java that led to its popularity two decades ago, and which still hold strong.

1.6 Features and components of Java

[1.5] Compare and contrast the features and components of Java such as: platform independence, object orientation, encapsulation, etc.

The Java programming language was released in 1995. It was developed mainly to work with consumer appliances. But it soon became very popular with web browsers, to deliver dynamic content (using applets), which didn't require it to be recompiled for separate platforms. Let's get started with the distinctive features and components of Java, which still make it a popular programming language.

NOTE The exam will question you on the features and components of Java that are relevant or irrelevant to it.

1.6.1 Valid features and components of Java

Java offers multiple advantages over other languages and platforms.

PLATFORM INDEPENDENCE

This feature is one of main reasons of Java's phenomenal rise since its release. It's also referred to as "write once, run anywhere" (WORA)—a slogan created by Sun Microsystems^{TM} to highlight Java's platform independence.

Java code can be executed on multiple systems without recompilation. Java code is compiled into *bytecode*, to be executed by a *virtual machine*—the Java Virtual Machine (JVM). A JVM is installed on platforms with different OSs like Windows, Mac, or Linux. A JVM interprets bytecodes to machine-specific instructions for execution. The implementation details of a JVM are machine-dependent and might differ across platforms, but all of them interpret the same bytecode in a similar manner. Bytecode generated by a Java compiler is supported by all platforms with a JVM.

Other popular programming languages like C and C++ compile their code to a host system. So the code must be recompiled for separate platforms.

Object orientation

Java emulates real-life object definition and behavior. In real life, state and behavior are tied to an object. Similarly, all Java code is defined within classes, interfaces, or enums. You need to create their objects to use them.

Abstraction

Java lets you abstract objects and include only the required properties and behavior in your code. For example, if you're developing an application that tracks the population of a country, you'll record a person's name, address, and contact details. But for a health-tracking system, you might want to include health-related details and behavior as well.

Encapsulation

With Java classes, you can encapsulate the state and behavior of an object. The state or the fields of a class are protected from unwanted access and manipulation. You can control the level of access and modifications to your objects.

Inheritance

Java enables its classes to inherit other classes and implement interfaces. The interfaces can inherit other interfaces. This saves you from redefining common code.

Polymorphism

The literal meaning of polymorphism is "many forms." Java enables instances of its classes to exhibit multiple behaviors for the same method calls. You'll learn about this in detail in chapter 6.

Type safety

In Java, you must declare a variable with its data type *before* you can use it. This means that you have compile-time checks that ensure you never assign to a variable a value of the wrong type.

Automatic memory management

Unlike other programming languages like C or C++, Java uses garbage collectors for automatic memory management. They reclaim memory from objects that are no longer in use. This frees developers from explicitly managing the memory themselves. It also prevents memory leaks.

MULTITHREADING AND CONCURRENCY

Java has supported multithreading and concurrency since it was first released—supported by classes and interfaces defined in its core API.

SECURITY

Java includes multiple built-in security features (though not all are covered in this exam) to control access to your resources and execution of your programs.

Java is type safe and includes garbage collection. It provides secure class loading, and verification ensures execution of legitimate Java code.

The Java platform defines multiple APIs, including cryptography and public key infrastructure. Java applications that execute under a security manager control access to your resources, like reading or writing to file. Access to a resource can be controlled using a policy file. Java enables you to define digital signatures, certificates, and keystores to secure code and file exchanges. Signed code is distributed for execution.

With features like encapsulation and data hiding, Java secures the state of its objects. Java applets execute in browsers and don't allow code to be downloaded to a system, thus enabling security for browsers and the systems that run them.

1.6.2 *Irrelevant features and components of Java*

The exam might also include some terms that are irrelevant.

SINGLE-THREADED

Java supports multithreading programming with inbuilt classes and interfaces. You can create and use single threads, but the Java language isn't single-threaded. Even when you create single threads of execution, Java executes its own processes like garbage collection in separate threads. Java isn't a single-threaded language.

RELATED TO JAVASCRIPT

Java isn't related to JavaScript (except for the similarity in their name). JavaScript is a programming language used in web pages to make them interactive.

1.7 *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 8 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 a package.

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.

Finally, we covered the features and components of Java that make it a popular choice.

1.8 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 8 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.
- This 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 allows usage of simple names, nonqualified names of classes, and interfaces.

- 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.
- If present, an `import` statement must be placed before any class or interface definition.
- 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 or a blank line. 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 *overloaded* versions 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 dot.
- 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 levels: `public`, `protected`, default, and `private`.
- Java defines three access modifiers: `public`, `protected`, 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're 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 nonaccess modifiers covered by this exam are `abstract`, `final`, 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 can't be instantiated.
- An interface is implicitly `abstract`. 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. When a non-abstract class extends a class with an abstract method, it must implement the 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 as both `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 independently 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 this 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.

Features and components of Java:

- *Object orientation*—Java emulates real-life object definition and behavior. It uses classes, interfaces, or enums to define all its code.
- *Abstraction*—Java lets you abstract objects and include only the required properties and behavior in your code.
- *Encapsulation*—The state or the fields of a class are protected from unwanted access and manipulation.
- *Inheritance*—Java enables its classes to inherit other classes and implement interfaces. The interfaces can inherit other interfaces.
- *Polymorphism*—Java enables instances of its classes to exhibit multiple behaviors for the same method calls.
- *Type safety*—In Java, you must declare a variable with its data type before you can use it.
- *Automatic memory management*—Java uses garbage collectors for automatic memory management. They reclaim memory from objects that are no longer in use.
- *Multithreading and concurrency*—Java defines classes and interfaces to enable developers to develop multithreaded code.
- Java isn't a single-threaded language.

1.9 Sample exam questions

Q1-1. Given:

```
class EJava {
    //..code
}
```

Which of the following options will compile?

a
```
package java.oca.associate;
class Guru {
    EJava eJava = new EJava();
}
```

b
```
package java.oca;
import EJava;
class Guru {
    EJava eJava;
}
```

c
```
package java.oca.*;
import java.default.*;
class Guru {
    EJava eJava;
}
```

d

```
package java.oca.associate;
import default.*;
class Guru {
    default.EJava eJava;
}
```

e None of the above

Q1-2. The following numbered list of Java class components is not in any particular order. Select the acceptable order of their occurrence in any 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 defines a 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;
}
```

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 `EJavaGuru`, 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. What is the meaning of "write once, run anywhere"? Select the correct options:

- **a** Java code can be written by one team member and executed by other team members.
- **b** It is for marketing purposes only.
- **c** It enables Java programs to be compiled once and can be executed by any JVM without recompilation.
- **d** Old Java code doesn't need recompilation when newer versions of JVMs are released.

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** 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.

Q1-10. 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.

1.10 Answers to sample exam questions

Q1-1. Given:

```
class EJava {
    //..code
}
```

Which of the following options will compile?

a
```
package java.oca.associate;
class Guru {
    EJava eJava = new EJava();
}
```

b
```
package java.oca;
import EJava;
class Guru {
    EJava eJava;
}
```

c
```
package java.oca.*;
import java.default.*;
class Guru {
    EJava eJava;
}
```

d
```
package java.oca.associate;
import default.*;
class Guru {
    default.EJava eJava;
}
```

e **None of the above**

Answer: e

Explanation: A class that isn't defined in a package gets implicitly defined in Java's default package. But such classes can't be accessed by classes or interfaces, which are explicitly defined in a package.

Option a is incorrect. The `EJava` class isn't defined in a package, so it can't be accessed by the `Guru` class, which is defined in the `java.oca.associate` package.

Options b, c, and d won't compile. Option b uses invalid syntax in the `import` statement. Options c and d try to import classes from nonexistent packages—*java.default* and *default*.

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 defines a 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: Option 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`) can't contain spaces. Also, `java virtual machine` isn't a valid package name to be imported in a class. The package name to be imported can't contain spaces.

Option c is incorrect because a `package` statement (if present) must 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 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 following 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. What is the meaning of "write once, run anywhere"? Select the correct options:

- a Java code can be written by one team member and executed by other team members.
- b It is for marketing purposes only.
- c **It enables Java programs to be compiled once and can be executed by any JVM without recompilation.**
- d Old Java code doesn't need recompilation when newer versions of JVMs are released.

Answer: c

Explanation: Platform independence, or "write once, run anywhere," enables Java code to be compiled once and run on any system with a JVM. It isn't for marketing purposes only.

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 base directory, mycode, in which package `com.ejavaguru` is defined, must not be included in the `import` statement.

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** 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.

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 nonpublic class can't be accessed outside a package in which it's defined. The class `Course` therefore can't be accessed from within the class `EJavaGuru`, even if it's 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 8
Programmer I Certification Guide

Mala Gupta

To earn the OCA Java SE 8 Programmer I Certification, you have to know your Java inside and out, and to pass the exam you need 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 8 Programmer I Certification Guide prepares Java developers for the 1Z0-808 with thorough coverage of Java topics typically found on the exam. Each chapter starts with a list of exam objectives mapped to section numbers, followed by sample questions and exercises that reinforce key concepts. You'll learn techniques and concepts in multiple ways, including memorable analogies, diagrams, flowcharts, and lots of well-commented code. You'll also get the scoop on common exam mistakes and ways to avoid traps and pitfalls.

What's Inside

- Covers all exam topics
- Hands-on coding exercises
- Flowcharts, UML diagrams, and other visual aids
- How to avoid built-in traps and pitfalls
- Complete coverage of the OCA Java SE 8 Programmer I exam (1Z0-808)

Written for developers with a working knowledge of Java who want to earn the OCA Java SE 8 Programmer I Certification.

Mala Gupta is a Java coach and trainer who holds multiple Java certifications. Since 2006 she has been actively supporting Java certification as a path to career advancement.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/oca-java-se-8-programmer-i-certification-guide

Free eBook
SEE INSERT

"Guides you through all the tricks and pitfalls you need to master in order to pass the exam."
—Jean-François Morin
Laval University

"Mala Gupta is a master of her art—she wrote the definitive guide to this exam!"
—Marty Henderson, Anthem Inc.

"Offers a thorough and well-structured preparation for the OCA exam."
—-Ursin Strauss, Swiss Post

"Clear, concise, correct, and complete."
—Travis Nelson
Software Technology Group

ISBN-13: 978-1-61729-325-2
ISBN-10: 1-61729-325-3
55999
9 781617 293252

MANNING $59.99 / Can $68.99 [INCLUDING eBOOK]