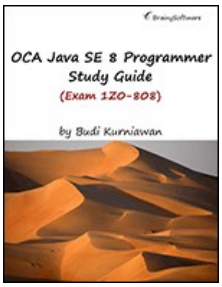


# OCA Java SE 8 Programmer Study Guide (Exam 1Z0-808)

by Budi Kurniawan



---

# Chapter 3: Objects and Classes

## Overview

Object-oriented programming (OOP) works by modeling applications on real-world objects. The benefits of OOP are real, which explains why OOP is the paradigm of choice today and why OOP languages like Java are popular. This chapter introduces you to objects and classes. If you are new to OOP, you may want to read this chapter carefully. A good understanding of OOP is key to writing quality programs.

This chapter starts by explaining what an object is and what constitutes a class. It then teaches you how to create an object with the **new** keyword, how objects are stored in memory, how classes can be organized into packages, how to use access control to achieve encapsulation, how the Java Virtual Machine (JVM) loads and links objects, and how Java manages unused objects. In addition, method overloading and static class members are explained.

## 3.1 What Is An Object?

When developing an application in an OOP language, you create a model that resembles a real-life situation to solve your problem. Take for example a payroll application, which calculates an employee's income tax and take home pay. An application like this would have a **Company** object to represent the company using the application, **Employee** objects that represent the employees in the company, **Tax** objects to represent the tax details of each employee, and so on. Before you can start programming such applications, however, you need to understand what Java objects are and how to create them.

Let's begin with a look at objects in life. Objects are everywhere, living (persons, pets, etc) and otherwise (cars, houses, streets, etc); concrete (books, televisions, etc) and abstract (love, knowledge, tax rate, regulations, and so forth). Every object has two features: the attributes and the actions the object can perform. For example, the following are some of a car's attributes:

- color
- number of doors
- plate number

Additionally, a car can perform these actions:

- run
- brake

As another example, a dog has the following attributes: color, age, type, weight, etc. And it can bark, run, urinate, sniff, etc.

A Java object also has attribute(s) and can perform action(s). In Java, attributes are called fields and actions are called methods. In other programming languages these may be known by other names. For example, methods are often called functions.

Both fields and methods are optional, meaning some Java objects may not have fields but have methods and some others may have fields but not methods. Some, of course, have both attributes and methods and some have neither.

How do you create Java objects? This is the same as asking, "How do you make cars?" Cars are expensive objects that need careful design that takes into account many things, such as safety and cost-effectiveness. You need a good blueprint to make good cars. To create Java objects, you need similar blueprints: classes.

## 3.2 Java Classes

A class is a blueprint or template to create objects of identical type. If you have an **Employee** class, you can create any number of **Employee** objects. To create **Street** objects, you need a **Street** class. A class determines what kind of object you get. For example, if you create an **Employee** class that has **age** and **position** fields, all **Employee** objects created out of this **Employee** class will have **age** and **position** fields as well. No more no less. The class determines the object.

In summary, classes are an OOP tool that enable programmers to create the abstraction of a problem. In OOP, abstraction is the act of using programming objects to represent real-world objects. As such, programming objects do not need to have the details of real-world objects. For instance, if an **Employee** object in a payroll application needs only be able to work and

receive a salary, then the **Employee** class would only need two methods, **work** and **receiveSalary**. OOP abstraction ignores the fact that a real-world employee can do many other things including eat, run, kiss and kick.

Classes are the fundamental building blocks of a Java program. All program elements in Java must reside in a class, even if you are writing a simple program that does not require Java's object-oriented features. A Java beginner needs to consider three things when writing a class:

- the class name
- the fields
- the methods

There are other things that can be present in a class, but they will be discussed later.

A class declaration must use the keyword **class** followed by a class name. Also, a class has a body within braces. Here is a general syntax for a class:

```
class className {
    [class body]
}
```

For example, Listing 3.1 shows a Java class named **Employee**, where the lines in bold are the class body.

Listing 3.1: The Employee class

```
class Employee {
    int age;
    double salary;
}
```

Note By convention, class names capitalize the initial of each word. For example, here are some names that follow the convention: **Employee, Boss, DateUtility, PostOffice, RegularRateCalculator**. This type of naming convention is known as Pascal naming convention. The other convention, the camel naming convention, capitalize the initial of each word, except the first word. Method and field names use the camel naming convention.

A public class definition must be saved in a file that has the same name as the class name, even though this restriction does not apply to non-public classes. The file name must have **java** extension.

Note In UML class diagrams, a class is represented by a rectangle that consists of three parts: the topmost part is the class name, the middle part is the list of fields, and the bottom part is the list of methods. (See Figure 3.1) The fields and methods can be hidden if showing them is not important.

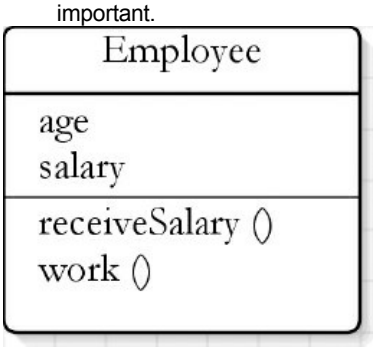


Figure 3.1: The Employee class in the UML class diagram

## Fields

Fields are variables. They can be primitives or references to objects. For example, the **Employee** class in Listing 3.1 has two fields, **age** and **salary**. In Chapter 1, "Language Fundamentals" you learned how to declare and initialize variables of primitive types.

However, a field can also refer to another object. For instance, an **Empoyee** class may have an **address** field of type **Address**, which is a class that represents a street address:

```
Address address;
```

In other words, an object can contain other objects, that is if the class of the former contains variables that reference to the latter.

Field names should follow the camel naming convention. The initial of each word in the field, except for the first word, is written with a capital letter. For example, here are some "good" field names: **age, maxAge, address, validAddress, numberOfRows**.

## Methods

A methods defines an action that a class's objects (or instances) can perform. A method has a declaration part and a body. The declaration part consists of a return value, the method name and a list of arguments. The body contains code that performs the action.

To declare a method, use the following syntax:

```
returnType methodName (listOfArguments)
```

The return type of a method can be a primitive, an object or void. The return type **void** means that the method returns nothing. The declaration part of a method is also called the signature of the method.

For example, here is a method named **getSalary** that returns a **double**.

```
double getSalary()
```

The **getSalary** method does not accept arguments.

As another example, here is a method that returns an **Address** object.

```
Address getAddress()
```

And, here is a method that accepts an argument:

```
int negate(int number)
```

If a method takes more than one argument, two arguments are separated by a comma. For example, the following **add** method takes two **int**s and return an **int**.

```
int add(int a, int b)
```

## The Method main

A special method called **main** provides the entry point to an application. An application normally has many classes and only one of the classes needs to have a **main** method. This method allows the containing class to be invoked.

The signature of the **main** method is as follows.

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

If you wonder why there is "public static void" before **main**, you will get the answer towards the end of this chapter.

You can pass arguments to **main** when using **java** to run a class. To pass arguments, type them after the class name. Two arguments are separated by a space.

```
java className arg1 arg2 arg3 ...
```

All arguments must be passed as strings. For instance, to pass two arguments, "1" and "safeMode" when running a **Test** class, type this:

```
java Test 1 safeMode
```

**String**s are discussed in Chapter 4, "Core Classes."

## Constructors

Every class must have at least one constructor. Otherwise, no objects could be created out of it and the class would be

useless. As such, if your class does not explicitly define a constructor, the compiler adds one for you.

A constructor is used to construct an object. A constructor looks like a method and is sometimes called a constructor method. However, unlike a method, a constructor does not have a return value, not even **void**. Additionally, a constructor must have the same name as the class.

The syntax for a constructor is as follows.

```
constructorName (listOfArguments) {
    [constructor body]
}
```

A constructor may have zero argument, in which case it is called a no-argument (or no-arg, for short) or default constructor. Constructor arguments can be used to initialize the fields in an object.

If the Java compiler adds a no-arg constructor to a class because the class contains no constructor, the addition will be implicit, i.e. it will not be displayed in the source file. However, if there is a constructor in a class definition, regardless of the number of arguments it accepts, no constructor will be added to the class by the compiler.

As an example, Listing 3.2 adds two constructors to the **Employee** class in Listing 3.1.

Listing 3.2: The Employee class with constructors

```
public class Employee {
    public int age;
    public double salary;
    public Employee() {
    }
    public Employee(int ageValue, double salaryValue) {
        age = ageValue;
        salary = salaryValue;
    }
}
```

The second constructor is particularly useful. Without it, to assign values to age and position, you would need to write extra lines of code to initialize the fields:

```
employee.age = 20;
employee.salary = 90000.00;
```

With the second constructor, you can pass the values at the same time you create an object.

```
new Employee(20, 90000.00);
```

The **new** keyword is new to you, but you will learn how to use it later in this chapter.

## Varargs

Varargs is a Java feature that allows methods to have a variable length of argument list. Here is an example of a method called **average** that accepts any number of **int**s and calculates their average.

```
public double average(int... args)
```

The ellipsis says that there is zero or more arguments of this type. For example, the following code calls **average** with two and three **int**s.

```
double avg1 = average(100, 1010);
double avg2 = average(10, 100, 1000);
```

If an argument list contains both fixed arguments (arguments that must exist) and variable arguments, the variable arguments must come last.

You should be able to implement methods that accept varargs after you read about arrays in Chapter 5, "Arrays." Basically, you receive a vararg as an array.

## Class Members in UML Class Diagrams

Figure 3.1 depicts a class in a UML class diagram. The diagram provides a quick summary of all fields and methods. You could do more in UML. UML allows you to include field types and method signatures. For example, Figure 3.2 presents a **Book** class with five fields and one method.

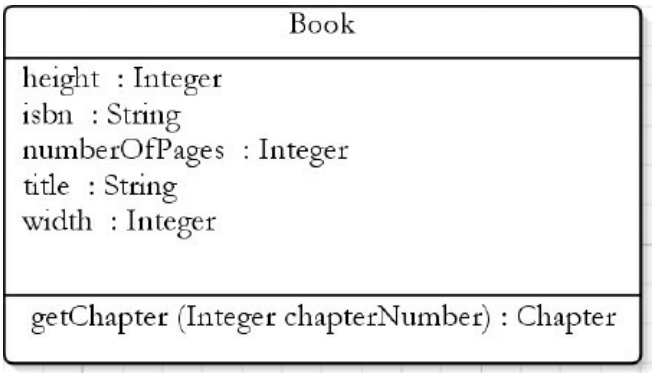


Figure 3.2: Including class member information in a class diagram

Note that in a UML class diagram a field and its type is separated by a colon. A method's argument list is presented in parentheses and its return type is written after a colon.

## 3.3 Creating An Object

Now that you know how to write a class, it is time to learn how to create an object from a class. An object is also called an instance. The word construct is often used in lieu of create, thus constructing an **Employee** object. Another term commonly used is *instantiate*. Instantiating the **Employee** class is the same as creating an instance of **Employee**.

There are a number of ways to create an object, but the most common one is by using the **new** keyword. **new** is always followed by the constructor of the class to be instantiated. For example, to create an **Employee** object, you write:

```
new Employee();
```

Most of the time, you will want to assign the created object to an object variable (or a reference variable), so that you can manipulate the object later. To achieve this, you need to declare an object reference with the same type as the object. For instance:

```
Employee employee = new Employee();
```

Here, **employee** is an object reference of type **Employee**.

Once you have an object, you can call its methods and access its fields, by using the object reference that was assigned the object. You use a period (.) to call a method or a field. For example:

```
objectReference.methodName
objectReference.fieldName
```

The following code, for instance, creates an **Employee** object and assigns values to its **age** and **salary** fields:

```
Employee employee = new Employee();
employee.age = 24;
employee.salary = 50000;
```

## 3.4 The null Keyword

A reference variable refers to an object. There are times, however, when a reference variable does not have a value (it is not referencing an object). Such a reference variable is said to have a null value. For example, the following class level reference variable is of type **Book** but has not been assigned a value;

```
Book book; // book is null
```

If you declare a local reference variable within a method but do not assign an object to it, you will need to assign null to it to satisfy the compiler:

```
Book book = null;
```

Class-level reference variables will be initialized when an instance is created, therefore you do not need to assign **null** to them.

Trying to access the field or method of a null variable reference raises an error, such as in the following code:

```
Book book = null;
System.out.println(book.title); // error because book is null
```

You can test if a reference variable is **null** by using the **==** operator. For instance.

```
if (book == null) {
    book = new Book();
}
System.out.println(book.title);
```

## 3.5 Memory Allocation for Objects

When you declare a variable in your class, either in the class level or in the method level, you allocate memory space for data that will be assigned to the variable. For primitives, it is easy to calculate the amount of memory taken. For example, declaring an **int** costs you four bytes and declaring a **long** sets you back eight bytes. However, calculation for reference variables is different.

When a program runs, some memory space is allocated for data. This data space is logically divided into two, the stack and the heap. Primitives are allocated in the stack and Java objects reside in the heap.

When you declare a primitive, several bytes are allocated in the stack. When you declare a reference variable, some bytes are also set aside in the stack, but the memory does not contain the object's data, it contains the address of the object in the heap. In other words, when you declare

```
Book book;
```

Some bytes are set aside for the reference variable **book**. The initial value of **book** is **null** because there is not yet an object assigned to it. When you write

```
Book book = new Book();
```

you create an instance of **Book**, which is stored in the heap, and assign the address of the instance to the reference variable **book**. A Java reference variable is like a C++ pointer except that you cannot manipulate a reference variable. In Java, a reference variable is used to access the member of the object it is referring to. Therefore, if the **Book** class has a public **review** method, you can call the method by using this syntax:

```
book.review();
```

An object can be referenced by more than one reference variable. For example,

```
Book myBook = new Book();
Book yourBook = myBook;
```

The second line copies the value of **myBook** to **yourBook**. As a result, **yourBook** is now referencing the same **Book** object as **myBook**.

Figure 3.3 illustrates memory allocation for a **Book** object referenced by **myBook** and **yourBook**.

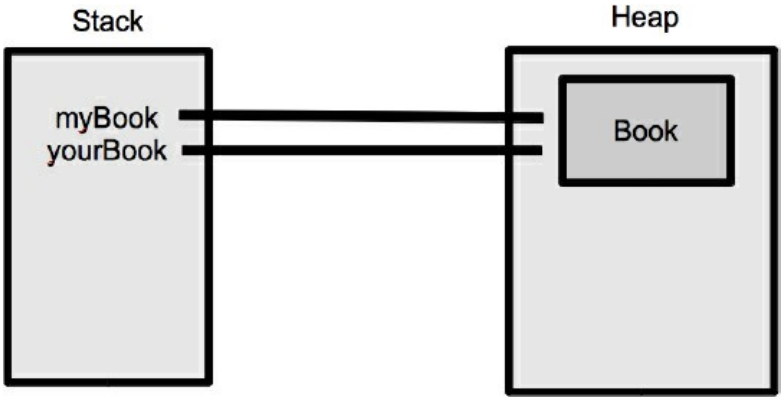


Figure 3.3: An object referenced by two variables

On the other hand, the following code creates two different **Book** objects:

```
Book myBook = new Book();
Book yourBook = new Book();
```

The memory allocation for this code is illustrated in Figure 3.4.

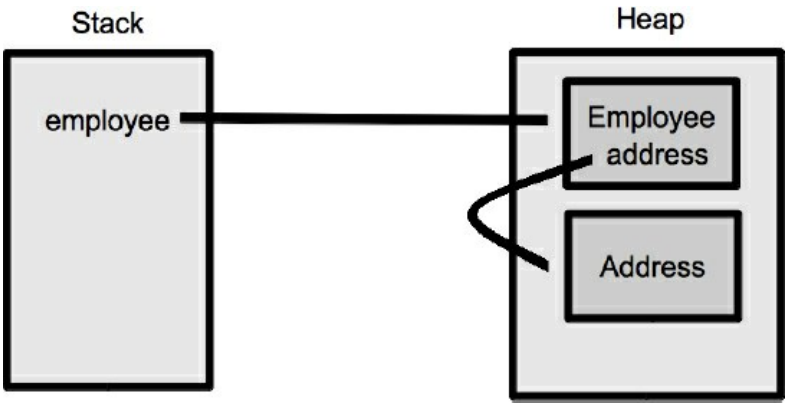


Figure 3.4: Two objects referenced by two variables

Now, how about an object that contains another object? For example, consider the code in Listing 3.3 that shows an **Employee** class that contains an **Address** class.

Listing 3.3: An Employee class that contains another class

```
public class Employee {
    Address address = new Address();
}
```

When you create an **Employee** object using the following code, an **Address** object is also created.

```
Employee employee = new Employee();
```

Figure 3.5 depicts the position of each object in the heap.

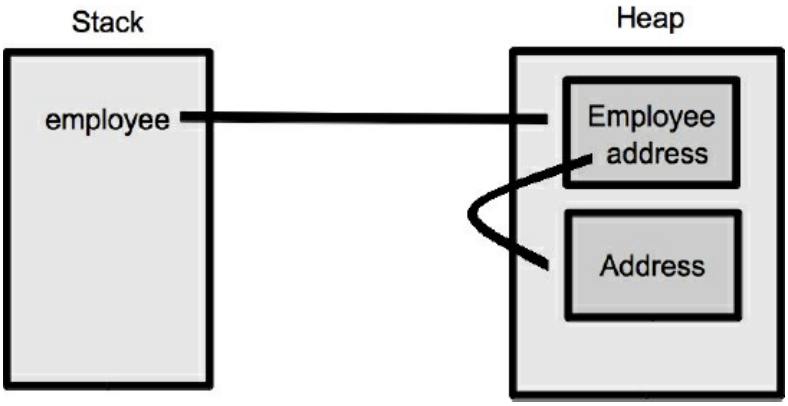


Figure 3.5: An object "within" another object

It turns out that the **Address** object is not really inside the **Employee** object. However, the **address** field within the **Employee** object has a reference to the **Address** object, thus allowing the **Employee** object to manipulate the **Address** object. Because in Java there is no way of accessing an object except through a reference variable assigned the object's address, no one else can access the **Address** object 'within' the **Employee** object.

## 3.6 Java Packages

If you are developing an application that consists of different parts, you may want to organize your classes to retain maintainability. With Java, you can group related classes or classes with similar functionality in packages. For example, standard Java classes come in packages. Java core classes are in the **java.lang** package. All classes for performing input and output operations are members of the **java.io** package, and so on. If a package needs to be organized in more detail, you can

create packages that share part of the name of the former. For example, the Java class library comes with the **java.lang.annotation** and **java.lang.reflect** packages. However, mind you that sharing part of the name does not make two packages related. The **java.lang** package and the **java.lang.reflect** package are different packages.

Package names that start with **java** are reserved for the core libraries. Consequently, you cannot create a package that starts with the word **java**. You can compile classes that belong to such a package, but you cannot run them.

In addition, packages starting with **javax** are meant for extension libraries that accompany the core libraries. You should not create packages that start with **javax** either.

In addition to class organization, packaging can avoid naming conflict. For example, an application may use the **MathUtil** class from company A and an identically named class from another company if both classes belong to different packages. For this purpose, by convention your package names should be based on your domain name in reverse. Therefore, Sun's package names start with **com.sun**. My domain name is brainysoftware.com, so it's appropriate for me to start my package name with **com.brainysoftware**. For example, I would place all my applets in a **com.brainysoftware.applet** package and my servlets in **com.brainysoftware.servlet**.

A package is not a physical object, and therefore you do not need to create one. To group a class in a package, use the keyword **package** followed by the package name. For example, the following **MathUtil** class is part of the **com.brainysoftware.common** package:

```
package com.brainysoftware.common;
public class MathUtil {
    ...
}
```

Java also introduces the term *fully qualified name*, which refers to a class name that carries with it its package name. The fully qualified name of a class is its package name followed by a period and the class name. Therefore, the fully qualified name of a **Launcher** class that belongs to package **com.example** is **com.example.Launcher**.

A class that has no package declaration is said to belong to the default package. For example, the **Employee** class in Listing 3.1 belongs to the default package. You should always use a package because types in the default package cannot be used by other types outside the default package (except when using a technique called reflection). It is a bad idea for a class to not have a package.

Even though a package is not a physical object, package names have a bearing on the physical location of their class source files. A package name represents a directory structure in which a period in a package name indicates a subfolder. For example, all Java source files in the **com.brainysoftware.common** package must reside in the **common** directory that is a subdirectory of the **brainysoftware** directory. In turn, the latter must be a subdirectory of the **com** directory. Figure 3.6 depicts the folder structure for a **com.brainysoftware.common.MathUtil** class.

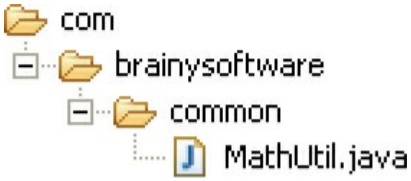


Figure 3.6: The physical location of a class in a package

Compiling a class in a non-default package presents a challenge for beginners. To compile such a class, you need to include the package name, replacing the dot (.) with /. For example, to compile the **com.brainysoftware.common.MathUtil** class, change directory to the working directory (the directory which is the parent directory of **com)** and type

```
javac com/brainysoftware/common/MathUtil.java
```

By default, **javac** will place the result in the same directory structure as the source. In this case, a **MathUtil.class** file will be created in the **com/brainysoftware/common** directory.

Running a class that belongs to a package follows a similar rule: you must include the package name, replacing . with /. For example, to run the **com.brainysoftware.common.MathUtil** class, type the following from your working directory.

```
java com/brainysoftware/common/MathUtil
```

The packaging of your classes also affects the visibility of your classes, as you will witness in the next section.

## 3.7 Encapsulation and Access Control

An OOP principle, encapsulation is a mechanism that protects parts of an object that need to be secure and exposes only parts that are safe to be exposed. A television is a good example of encapsulation. Inside it are thousands of electronic components that together form the parts that can receive signals and decode them into images and sound. These components are not to be accessible to the user, however, so Sony and other manufacturers wrap them in a strong metallic cover that does not break easily. For a television to be easy to use, it exposes buttons that the user can touch to turn on and off the set, adjust brightness, turn up and down the volume, and so on.

Back to encapsulation in OOP, let's take as an example a class that can encode and decode messages. The class exposes two methods called **encode** and **decode**, that users of the class can access. Internally, there are dozens of variables used to store temporary values and other methods that perform supporting tasks. The author of the class hides these variables and other methods because allowing access to them may compromise the security of the encoding/decoding algorithms. Besides, exposing too many things makes the class harder to use. As you can see later, encapsulation is a powerful feature.

Java supports encapsulation through access control. Access control is governed by access control modifiers. There are four access control modifiers in Java: **public, protected, private** and the default access level. Access control modifiers can be applied to classes or class members. They are explained in the following subsections.

### Class Access Control Modifiers

In an application with many classes, a class may be instantiated and used from another class that is a member of the same package or a different package. You can control from which packages your class can be "seen" by employing an access control modifier at the beginning of the class declaration.

A class can have either the public or the default access control level. You make a class public by using the **public** access control modifier. A class whose declaration bears no access control modifier has default access. A public class is visible from anywhere. Listing 3.4 shows a public class named **Book**.

Listing 3.4: The public class Book

```
package app03;
public class Book {
    String isbn;
    String title;
    int width;
    int height;
    int numberOfPages;
}
```

The **Book** class is a member of the **app03** package and has five fields. Since **Book** is public, it can be instantiated from any other classes. In fact, the majority of the classes in the Java core libraries are public classes. For example, here is the declaration of the **java.lang.Runtime** class:

```
public class Runtime
```

A public class must be saved in a file that has the same name as the class, and the extension must be **java**. The **Book** class in Listing 3.4 must be saved in a **Book.java** file. Also, because **Book** belongs to package **app03**, the **Book.java** file must reside inside an **app03** directory.

Note A Java source file can only contain one public class. However, it can contain multiple classes that are not public.

When there is no access control modifier preceding a class declaration, the class has the default access level. For example, Listing 3.5 presents the **Chapter** class that has the default access level.

Listing 3.5: The Chapter class, with the default access level

```
package app03;
class Chapter {
    String title;
    int numberOfPages;

    public void review() {
        Page page = new Page();
        int sentenceCount = page.numberOfSentences;
```

```
        int pageNumber = page.getPageNumber();
    }
}
```

Classes with the default access level can only be used by other classes that belong to the same package. For instance, the **Chapter** class can be instantiated from inside the **Book** class because **Book** belongs to the same package as **Chapter**. However, **Chapter** is not visible from other packages.

For example, you can add the following **getChapter** method inside the **Book** class:

```
Chapter getChapter() {
    return new Chapter();
}
```

On the other hand, if you try to add the same **getChapter** method to a class that does not belong to the **app03** package, a compile error will be raised.

## Class Member Access Control Modifiers

Class members (methods, fields, constructors, etc) can have one of four access control levels: public, protected, private and default access. The access control modifier **public** is used to make a class member public, the **protected** modifier to make a class member protected, and the **private** modifier to make a class member private. Without an access control modifier, a class member will have the default access level.

Table 3.1 shows the visibility of each access level.

Table 3.1: Class member access levels

| Access Level | From classes in other packages | From classes in the same package | From child classes | From the same class |
|---|---|---|---|---|
| public | yes | yes | yes | yes |
| protected | no | yes | yes | yes |
| default | no | yes | no | yes |
| private | no | no | no | yes |

Note The default access is sometimes called package private. To avoid confusion, this book will only use the term default access.

A public class member can be accessed by any other classes that can access the class containing the class member. For example, the **toString** method of the **java.lang.Object** class is public. Here is the method signature:

```
public String toString()
```

Once you construct an **Object** object, you can call its **toString** method because **toString** is public.

```
Object obj = new Object();
obj.toString();
```

Recall that you access a class member by using this syntax:

```
referenceVariable.memberName
```

In the preceding code, **obj** is a reference variable to an instance of **java.lang.Object** and **toString** is the method defined in the **java.lang.Object** class.

A protected class member has a more restricted access level. It can be accessed only from

- any class in the same package as the class containing the member
- a child class of the class containing the member

Note A child class is a class that extends another class. Chapter 6, "Inheritance" explains this concept.

For instance, consider the public class **Page** in Listing 3.6.

Listing 3.6: The Page class

```
package app03;
public class Page {
```

```
    int numberOfSentences = 10;
    private int pageNumber = 5;
    protected int getPageNumber() {
        return pageNumber;
    }
}
```

---

**Page** has two fields (**numberOfSentences** and **pageNumber**) and one method (**getPageNumber**). First of all, because **Page** is public, it can be instantiated from any other class. However, even if you can instantiate it, there is no guarantee you can access its members. It depends on from which class you are accessing the **Page** class's members.

Its **getPageNumber** method is protected, so it can be accessed from any classes that belong to **app03**, the package that houses the **Page** class. For example, consider the **review** method in the **Chapter** class (given in Listing 3.5).

```
public void review() {
    Page page = new Page();
    int sentenceCount = page.numberOfSentences;
    int pageNumber = page.getPageNumber();
}
```

The **Chapter** class can access the **getPageNumber** method because **Chapter** belongs to the same package as the **Page** class. Therefore, **Chapter** can access all protected members of the **Page** class.

The default access allows classes in the same package access a class member. For instance, the **Chapter** class can access the **Page** class's **numberOfSentences** field because the **Page** and **Chapter** classes belong to the same package. However, **numberOfSentences** is not accessible from a subclass of **Page** if the subclass belongs to a different package. This differentiates the protected and default access levels and will be explained in detail in Chapter 6, "Inheritance."

The private members of a class can only be accessed from inside the same class. For example, there is no way you can access the **Page** class's private field **pageNumber** from anywhere other than the **Page** class itself. However, look at the following code from the **Page** class definition.

```
private int pageNumber = 5;
protected int getPageNumber() {
    return pageNumber;
}
```

The **pageNumber** field is private, so it can be accessed from the **getPageNumber** method, which is defined in the same class. The return value of **getPageNumber** is **pageNumber**, which is private. Beginners are often confused by this kind of code. If **pageNumber** is private, why use it as a return value of a protected method (**getPageNumber**)? Note that access to **pageNumber** is still private, so other classes cannot modify this field. However, using it as a return value of a non-private method is allowed.

How about constructors? Access levels to constructors are the same as those for fields and methods. Therefore, constructors can have public, protected, default, and private access levels. You may think that all constructors must be public because the intention of having a constructor is to make the class instantiatable. However, to your surprise, this is not the case. Some constructors are made private so that their classes cannot be instantiated from other classes. Private constructors are normally used in singleton classes. If you are interested in this topic, there are articles on this that you can find easily on the Internet.

Note In a UML class diagram, you can include information on the class member access level. Prefix a public member with a +, a protected member with a # and a private member with a -. Members with no prefix are regarded as having the default access level. Figure 3.7 shows the **Manager** class with members having various access levels.

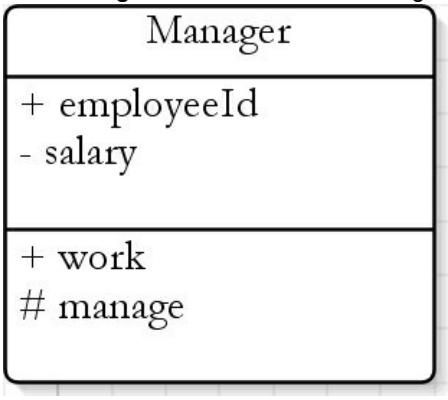


Figure 3.7: Including class member access level in a UML class diagram

## 3.8 The this Keyword

You use the **this** keyword from any method or constructor to refer to the current object. For example, if you have a class-level field with the same name as a local variable, you can use this syntax to refer to the former:

```
this.field
```

A common use is in the constructor that accepts values used to initialize fields. Consider the **Box** class in Listing 3.7.

Listing 3.7: The Box class

```
package app03;
public class Box {
    int length;
    int width;
    int height;
    public Box(int length, int width, int height) {
        this.length = length;
        this.width = width;
        this.height = height;
    }
}
```

The **Box** class has three fields, **length, width**, and **height**. Its constructor accepts three arguments used to initialize the fields. It is very convenient to use **length, width**, and **height** as the parameter names because they reflect what they are. Inside the constructor, **length** refers to the **length** argument, not the **length** field. **this.length** refers to the classlevel **length** field.

It is of course possible to change the argument names, such as this.

```
public Box (int lengthArg, int widthArg, int heightArg) {
    length = lengthArg;
    width = widthArg;
    height = heightArg;
}
```

This way, the class-level fields are not shadowed by local variables and you do not need to use the **this** keyword to refer to the class-level fields However, using the **this** keyword spares you from having to think of different names for your method or constructor arguments.

## 3.9 Using Other Classes

It is common to use other classes from the class you are writing. Using classes in the same package as your current class is allowed by default. However, to use classes in other packages, you must first import the packages or the classes you want to use.

Java provides the keyword **import** to indicate that you want to use a package or a class from a package. For example, to use the **java.util.ArrayList** class from your code, you must have the following **import** statement:

```
package app03;
import java.util.ArrayList;

public class Demo {
    ...
}
```

Note that **import** statements must come after the **package** statement but before the class declaration. The **import** keyword can appear multiple times in a class.

```
package app03;
import java.time.Clock;
import java.util.ArrayList;

public class Demo {
    ...
}
```

Sometimes you need many classes in the same package. You can import all classes in the same package by using the wild character *. For example, the following code imports all members of the **java.util** package.

```
package app03;
import java.util.*;
public class Demo {
    ...
}
```

Now, not only can you use the **java.util.ArrayList** class, but you can use other members of the **java.util** package too. However, to make your code more readable, it is recommended that you import a package member one at a time. In other words, if you need to use both the **java.io.PrintWriter** class and the **java.io.FileReader** class, it is better to have two **import** statements like the following than to use the * character.

```
import java.io.PrintWriter;
import java.io.FileReader;
```

Note Members of the **java.lang** package are imported automatically. Thus, to use the **java.lang.String** class, for example, you do not need to explicitly import the class.

The only way to use classes that belong to other packages without importing them is to use the fully qualified names of the classes in your code. For example, the following code declares the **java.io.File** class using its fully qualified name.

```
java.io.File file = new java.io.File(filename);
```

If you import identically-named classes from different packages, you must use the fully qualified names when declaring the classes. For example, the Java core libraries contain the classes **java.sql.Date** and **java.util.Date**. Importing both upsets the compiler. In this case, you must write the fully qualified names of **java.sql.Date** and **java.util.Date** in your class to use them.

Note Java classes can be deployed in a jar file. Appendix A details how to compile a class that uses other classes in a jar file. Appendix B shows how to run a Java class in a jar file. Appendix C provides instructions on the **jar** tool, a program that comes with the JDK to package your Java classes and related resources. A class that uses another class is said to "depend on" the latter. A UML diagram that depicts this dependency is shown in Figure 3.8.

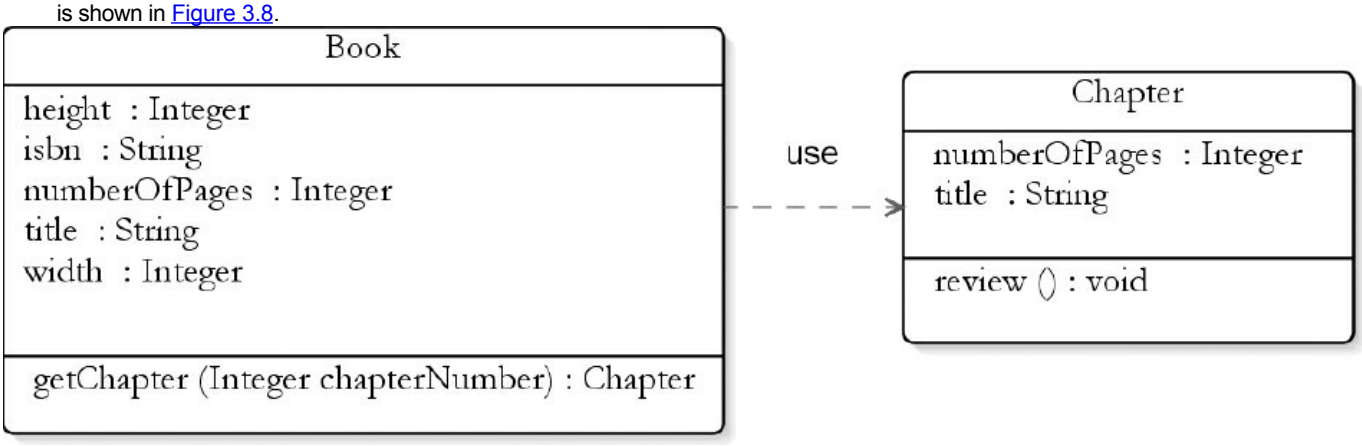


Figure 3.8: Dependency in the UML class diagram

A dependency relationship is represented by a dashed line with an arrow. In Figure 3.8 the **Book** class is dependent on **Chapter** because the **getChapter** method returns a **Chapter** object.

## 3.10 Final Variables

Java does not reserve the keyword constant to create constants. However, in Java you can prefix a variable declaration with the keyword **final** to make its value unchangeable. You can make both local variables and class fields final.

For example, the number of months in a year never changes, so you can write:

```
final int numberOfMonths = 12;
```

As another example, in a class that performs mathematical calculation, you can declare the variable **pi** whose value is equal to 22/7 (the circumference of a circle divided by its diameter, in math represented by the Greek letter π).

```
final float pi = (float) 22 / 7;
```

Once assigned a value, the value cannot change. Attempting to change it will result in a compile error.

Note that the casting **(float)** after **22 / 7** is needed to convert the value of division to **float**. Otherwise, an **int** will be returned

and the **pi** variable will have a value of 3.0, instead of 3.1428.

Also note that since Java uses Unicode characters, you can simply define the variable **pi** as π if you don't think typing it is harder than typing **pi**.

```
final float π = (float) 22 / 7;
```

Note You can also make a method final, thus prohibiting it from being overridden in a subclass. This will be discussed in Chapter 6, "Inheritance."

## 3.11 Static Members

You have learned that to access a public field or method of an object, you use a period after the object reference, such as:

```
// Create an instance of Book
Book book = new Book();
// access the review method
book.review();
```

This implies that you must create an object first before you can access its members. However, in previous chapters, there were examples that used **System.out.print** to print values to the console. You may have noticed that you could call the **out** field without first having to construct a **System** object. How come you did not have to do something like this?

```
System ref = new System();
ref.out;
```

Rather, you use a period after the class name:

```
System.out
```

Java (and many OOP languages) supports the notion of static members, which are class members that can be called without first instantiating the class. The **out** field in **java.lang.System** is static, which explains why you can write **System.out**.

Static members are not tied to class instances. Rather, they can be called without having an instance. In fact, the method **main**, which acts as the entry point to a class, is static because it must be called before any object is created.

To create a static member, you use the keyword **static** in front of a field or method declaration. If there is an access modifier, the **static** keyword may come before or after the access modifier. These two are correct:

```
public static int a;
static public int b;
```

However, the first form is more often used.

For example, Listing 3.8 shows the **MathUtil** class with a static method:

Listing 3.8: The MathUtil class

```
package app03;
public class MathUtil {
    public static int add(int a, int b) {
        return a + b;
    }
}
```

To use the **add** method, you can simply call it this way:

```
MathUtil.add(a, b)
```

The term instance methods/fields are used to refer to non-static methods and fields.

From inside a static method, you cannot call instance methods or instance fields because they only exist after you create an object. From a static method, you can access other static methods or static fields, however.

A common confusion that a beginner often encounters is when they cannot compile their class because they are calling instance members from the **main** method. Listing 3.9 shows such a class.

Listing 3.9: Calling non-static members from a static method

```
package app03;
public class StaticDemo {
    public int b = 8;
    public static void main(String[] args) {
        System.out.println(b);
    }
}
```

---

The line in bold causes a compile error because it attempts to access non-static field **b** from the **main** static method. There are two solutions to this.

1. Make **b** static
2. Create an instance of the class, then access **b** by using the object reference.

Which solution is appropriate depends on the situation. It often takes years of OOP experience to come up with a good decision that you're comfortable with.

Note You can only declare a static variable in a class level. You cannot declare local static variables even if the method is static.

How about static reference variables? You can declare static reference variables. The variable will contain an address, but the object referenced is stored in the heap. For instance

```
static Book book = new Book();
```

Static reference variables provide a good way of exposing the same object that needs to be shared among other different objects.

Note In UML class diagrams, static members are underlined. For example, Figure 3.9 shows the **MathUtil** class with the static method **add**.

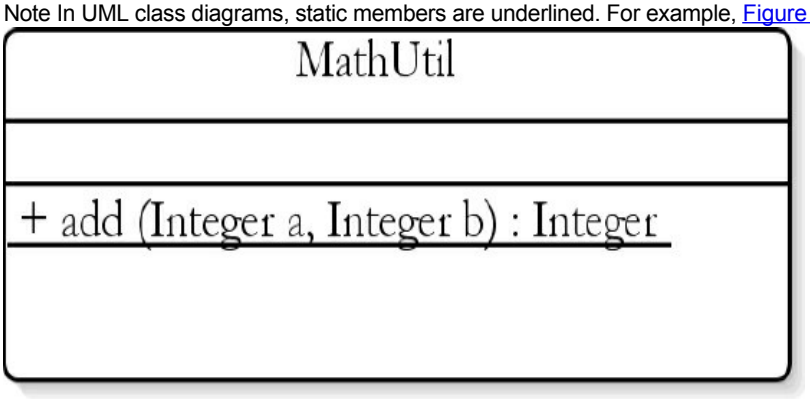


Figure 3.9: Static members in UML class diagrams

## 3.12 Static Final Variables

In the section "Final Variables" earlier in this chapter, you learned that you could create a final variable by using the keyword **final**. However, final variables at a class level or local variables will always have the same value when the program is run. If you have multiple objects of the same class with final variables, the value of the final variables in those objects will have the same values. It is more common (and also more prudent) to make a final variable static too. This way, all objects share the same value.

The naming convention for static final variables is to have them in upper case and separate two words with an underscore. For example

```
static final int NUMBER_OF_MONTHS = 12;
static final float PI = (float) 22 / 7;
```

The positions of **static** and **final** are interchangeable, but it is more common to use "static final" than "final static."

If you want to make a static final variable accessible from outside the class, you can make it public too:

```
public static final int NUMBER_OF_MONTHS = 12;
public static final float PI = (float) 22 / 7;
```

To better organize your constants, sometimes you want to put all your static final variables in a class. This class most often

does not have a method or other fields and is never instantiated.

For example, sometimes you want to represent a month as an **int**, therefore January is 1, February is 2, and so on. Then, you use the word January instead of number 1 because it's more descriptive. Listing 3.10 shows the **Months** class that contains the names of months and its representation.

Listing 3.10: The Months class

```
package app03;
public class Months {
    public static final int JANUARY = 1;
    public static final int FEBRUARY = 2;
    public static final int MARCH = 3;
    public static final int APRIL = 4;
    public static final int MAY = 5;
    public static final int JUNE = 6;
    public static final int JULY = 7;
    public static final int AUGUST = 8;
    public static final int SEPTEMBER = 9;
    public static final int OCTOBER = 10;
    public static final int NOVEMBER = 11;
    public static final int DECEMBER = 12;
}
```

In your code, you can get the representation of January by writing.

```
int thisMonth = Months.JANUARY;
```

Static final reference variables are also possible. However, note that only the variable is final, which means once it is assigned an address to an instance, it cannot be assigned another object of the same type. The fields in the referenced object itself can be changed.

In the following line of code

```
public static final Book book = new Book();
```

**book** always refer to this particular instance of **Book**. Reassigning it to another **Book** object raises a compile error:

```
book = new Book(); // compile error
```

However, you can change the **Book** object's field value.

```
book.title = "No Excuses";  // assuming the title field is public
```

## 3.13 Static import

There are a number of classes in the Java core libraries that contain static final fields. One of them is the **java.util.Calendar** class, that has the static final fields representing days of the week (**MONDAY, TUESDAY**, etc). To use a static final field in the **Calendar** class, you must first import the **Calendar** class.

```
import java.util.Calendar;
```

Then, you can use it by using the notation *className.staticField*.

```
if (today == Calendar.SATURDAY)
```

However, you can also import static fields using the **import static** keywords. For example, you can do

```
import static java.util.Calendar.SATURDAY;
```

Then, to use the imported static field, you do not need the class name:

```
if (today == SATURDAY)
```

## 3.14 Variable Scope

You have seen that you can declare variables in several different places:

- In a class body as class fields. Variables declared here are referred to as class-level variables.
- As parameters of a method or constructor.
- In a method's body or a constructor's body.
- Within a statement block, such as inside a **while** or **for** block.

Now it's time to learn variable scope.

Variable scope refers to the accessibility of a variable. The rule is that variables defined in a block are only accessible from within the block. The scope of the variable is the block in which it is defined. For example, consider the following **for** statement.

```
for (int x = 0; < 5; x++) {
    System.out.println(x);
}
```

The variable **x** is declared within the **for** statement. As a result, **x** is only available from within this **for** block. It is not accessible or visible from anywhere else. When the JVM executes the **for** statement, it creates **x**. When it is finished executing the **for** block, it destroys **x**. After **x** is destroyed, **x** is said to be out of scope.

Rule number 2 is a nested block can access variables declared in the outer block. Consider this code.

```
for (int x = 0; x < 5; x++) {
    for (int y = 0; y < 3; y++) {
        System.out.println(x);
        System.out.println(y);
    }
}
```

The preceding code is valid because the inner **for** block can access **x**, which is declared in the outer **for** block.

Following the rules, variables declared as method parameters can be accessed from within the method body. Also, class-level variables are accessible from anywhere in the class.

If a method declares a local variable that has the same name as a class-level variable, the former will 'shadow' the latter. To access the class-level variable from inside the method body, use the **this** keyword.

## 3.15 Method Overloading

Method names are very important and should reflect what the methods do. In many circumstances, you may want to use the same name for multiple methods because they have similar functionality. For instance, the method **printString** may take a **String** argument and prints the string. However, the same class may also provide a method that prints part of a **String** and accepts two arguments, the **String** to be printed and the character position to start printing from. You want to call the latter method **printString** too because it does print a **String**, but that would be the same as the first **printString** method.

Thankfully, it is okay in Java to have multiple methods having the same name, as long as each method accept different sets of argument types. In other words, in our example, it is legal to have these two methods in the same class.

```
public String printString(String string)
public String printString(String string, int offset)
```

This feature is called method overloading.

The return value of the method is not taken into consideration. As such, these two methods must not exist in the same class:

```
public int countRows(int number);
public String countRows(int number);
```

This is because a method can be called without assigning its return value to a variable. In such situations, having the above **countRows** methods would confuse the compiler as it would not know which method is being called when you write

```
System.out.println(countRows(3));.
```

A trickier situation is depicted in the following methods whose signatures are very similar.

```
public int printNumber(int i) {
    return i*2;
```

```
}

public long printNumber(long l) {
    return l*3;
}
```

It is legal to have these two methods in the same class. However, you might wonder, which method is being called if you write **printNumber(3)?**

The key is to recall from Chapter 1, "Language Fundamentals" that a numeric literal will be translated into an **int** unless it is suffixed **L** or **l**‥ Therefore, **printNumber(3)** will invoke this method:

```
public int printNumber(int i)
```

To call the second, pass a **long:**

```
printNumber(3L);
```

**System.out.print()** (and **System.out.println())** is an excellent example of method overloading. You can pass any primitive or object to the method because there are nine overloads of the method. There is an overload that accepts an **int**, one that accepts a **long**, one that accepts a **String**, and so on.

Note Static methods can also be overloaded.

## 3.16 Static Factory Methods

You've learned to create an object using **new**. However, there are classes in Java class libraries that cannot be instantiated this way. For example, you cannot create an instance of **java.util.LocalDate** with **new** because its constructor is private. Instead, you would use one of its static methods, such as **now:**

```
LocalDate today = LocalDate.now();
```

Such methods are called static factory methods.

You can design your class to use static factory methods. Listing 3.11 shows a class named **Discount** with a private constructor. It is a simple class that contains an **int** that represents a discount rate. The value is either 10 (for small customers) or 12 (for bigger customers). It has a **getValue** method, which returns the value, and two static factory methods, **createSmallCustomerDiscount** and **createBigCustomerDiscount**. Note that the static factory methods can invoke the private constructor to create an object because they are in the same class. Recall that you can access a class private member from within the class. With this design, you restrict a **Discount** object to contain either 10 or 12. Other values are not possible.

Listing 3.11: The Discount classs

```
package app03;
import java.time.LocalDate;

public class Discount {
    private int value;
    private Discount(int value) {
        this.value = value;
    }

    public int getValue() {
        return this.value;
    }

    public static Discount createSmallCustomerDiscount() {
        return new Discount(10);
    }

    public static Discount createBigCustomerDiscount() {
        return new Discount(12);
    }
}
```

You can construct a **Discount** object by calling one of its static factory methods, for example

```
Discount discount = Discount.createBigCustomerDiscount();
System.out.println(discount.getValue());
```

There are also classes that allow you to create an instance through static factory methods and a constructor. In this case, the constructor must be public. Examples of such classes are **java.lang.Integer** and **java.lang.Boolean**.

With static factory methods, you can control what objects can be created out of your class, like you have seen in **Discount**. Also, you might cache an instance and return the same instance every time an instance is needed. Also, unlike constructors, you can name static factory methods to make clear what kind of object will be created.

## 3.17 By Value or By Reference?

You can pass primitive variables or reference variables to a method. Primitive variables are passed by value and reference variables are passed by reference. What this means is when you pass a primitive variable, the JVM will copy the value of the passed-in variable to a new local variable. If you change the value of the local variable, the change will not affect the passed in primitive variable.

If you pass a reference variable, the local variable will refer to the same object as the passed in reference variable. If you change the object referenced within your method, the change will also be reflected in the calling code. Listing 3.12 shows the **ReferencePassingTest** class that demonstrates this.

Listing 3.12: The ReferencePassingTest class

```
package app03;
class Point {
    public int x;
    public int y;
}
public class ReferencePassingTest {
    public static void increment(int x) {
        x++;
    }
    public static void reset(Point point) {
        point.x = 0;
        point.y = 0;
    }
    public static void main(String[] args) {
        int a = 9;
        increment(a);
        System.out.println(a); // prints 9
        Point p = new Point();
        p.x = 400;
        p.y = 600;
        reset(p);
        System.out.println(p.x); // prints 0
    }
}
```

There are two methods in **ReferencePassingTest, increment** and **reset**. The **increment** method takes an **int** and increments it. The **reset** method accepts a **Point** object and resets its **x** and **y** fields.

Now pay attention to the **main** method. We passed **a** (whose value is 9) to the **increment** method. After the method invocation, we printed the value of **a** and you get 9, which means that the value of **a** did not change.

Afterwards, you create a **Point** object and assign the reference to **p**. You then initialize its fields and pass it to the **reset** method. The changes in the **reset** method affects the **Point** object because objects are passed by reference. As a result, when you print the value of **p.x**, you get 0.

## 3.18 Loading, Linking, and Initialization

Now that you've learned how to create classes and objects, let's take a look at what happens when the JVM executes a class.

You run a Java program by using the **java** tool. For example, you use the following command to run the **DemoTest** class.

```
java DemoTest
```

After the JVM is loaded into memory, it starts its job by invoking the **DemoTest** class's **main** method. There are three things the JVM will do next in the specified order: loading, linking, and initialization.

## Loading

The JVM loads the binary representation of the Java class (in this case, the **DemoTest** class) to memory and may cache it in memory, just in case the class is used again in the future. If the specified class is not found, an error will be thrown and the process stops here.

## Linking

There are three things that need to be done in this phase: verification, preparation, and resolution (optional). Verification means the JVM checks that the binary representation complies with the semantic requirements of the Java programming language and the JVM. If, for example, you tamper with a class file created as a result of compilation, the class file may no longer work.

Preparation prepares the specified class for execution. This involves allocating memory space for static variables and other data structured for that class.

Resolution checks if the specified class references other classes/interfaces and if the other classes/interfaces can also be found and loaded. Checks will be done recursively to the referenced classes/interfaces.

For example, if the specified class contains the following code:

```
MathUtil.add(4, 3)
```

the JVM will load, link, and initialize the **MathUtil** class before calling the static **add** method.

Or, if the following code is found in the **DemoTest** class:

```
Book book = new Book();
```

the JVM will load, link, and initialize the **Book** class before an instance of **Book** is created.

Note that a JVM implementation may choose to perform resolution at a later stage, i.e. when the executing code actually requires the use of the referenced class/interface.

## Initialization

In this last step, the JVM initializes static variables with assigned or default values and executes static initializers (code in **static** blocks). Initialization occurs just before the **main** method is executed. However, before the specified class can be initialized, its parent class will have to be initialized. If the parent class has not been loaded and linked, the JVM will first load and link the parent class. Again, when the parent class is about to be initialized, the parent's parent will be treated the same. This process occurs recursively until the initialized class is the topmost class in the hierarchy.

For example, if a class contains the following declaration

```
public static int z = 5;
```

the variable **z** will be assigned the value 5. If no initialization code is found, a static variable is given a default value. Table 3.2 lists default values for Java primitives and reference variables.

Table 3.2: Default values for primitives and references

| Type | Default Value |
|---|---|
| boolean | false |
| byte | 0 |
| short | 0 |
| int | 0 |
| long | 0L |
| char | \u0000 |
| float | 0.0f |
| double | 0.0d |
| object reference | null |

In addition, code in **static** blocks will be executed. For example, Listing 3.13 shows the **StaticCodeTest** class with static code that gets executed when the class is loaded. Like static members, you can only access static members from static code.

Listing 3.13: StaticCodeTest

```
package app03;
public class StaticInitializationTest {
    public static int a = 5;
    public static int b = a * 2;
    static {
        System.out.println("static");
        System.out.println(b);
    }
    public static void main(String[] args) {
        System.out.println("main method");
    }
}
```

If you run this class, you will see the following on your console:

```
static
10
main method
```

## 3.19 Object Creation Initialization

Initialization happens when a class is loaded, as described in the section "Linking, Loading, and Initialization" earlier in this chapter. However, you can also write code that performs initialization every time an instance of a class is created.

When the JVM encounters code that instantiates a class, the JVM does the following.

1. Allocates memory space for a new object, with room for the instance variables declared in the class plus room for instance variables declared in its parent classes.
2. Processes the invoked constructor. If the constructor has parameters, the JVM creates variables for these parameter and assigns them values passed to the constructor.
3. If the invoked constructor begins with a call to another constructor (using the **this** keyword), the JVM processes the called constructor.
4. Performs instance initialization and instance variable initialization for this class. Instance variables that are not assigned a value will be assigned default values (See Table 3.2). Instance initialization applies to code in braces:

   ```
   {
        // code
   }
   ```

5. Executes the rest of the body of the invoked constructor.
6. Returns a reference variable that refers to the new object.

Note that instance initialization is different from static initialization. The latter occurs when a class is loaded and has nothing to do with instantiation. Instance initialization, by contrast, is performed when an object is created. In addition, unlike static initializers, instance initialization may access instance variables.

For example, Listing 3.14 presents a class named **InitTest1** that has the initialization section. There is also some static initialization code to give you the idea of what is being run.

Listing 3.14: The InitTest1 class

```
package app03;

public class InitTest1 {
    int x = 3;
    int y;
    // instance initialization code
    {
        y = x * 2;
```

```
        System.out.println(y);
    }

    // static initialization code
    static {
        System.out.println("Static initialization");
    }
    public static void main(String[] args) {
        InitTest1 test = new InitTest1();
        InitTest1 moreTest = new InitTest1();
    }
}
```

---

When run, the **InitTest** class prints the following on the console:

```
Static initialization
6
6
```

The static initialization is performed first, before any instantiation takes place. This is where the JVM prints the "Static initialization" message. Afterward, the **InitTest1** class is instantiated twice, explaining why you see "6" twice.

The problem with having instance initialization code is this. As your class grows bigger it becomes harder to notice that there exists initialization code.

Another way to write initialization code is in the constructor. In fact, initialization code in a constructor is more noticeable and hence preferable. Listing 3.15 shows the **InitTest2** class that puts initialization code in the constructor.

Listing 3.15: The InitTest2 class

---

```
package app03;
public class InitTest2 {
    int x = 3;
    int y;
    // instance initialization code
    public InitTest2() {
        y = x * 2;
        System.out.println(y);
    }
    // static initialization code
    static {
        System.out.println("Static initialization");
    }
    public static void main(String[] args) {
        InitTest2 test = new InitTest2();
        InitTest2 moreTest = new InitTest2();
    }
}
```

---

The problem with this is when you have more than one constructor and each of them must call the same code. The solution is to wrap the initialization code in a method and let the constructors call them. Listing 3.16 shows this

Listing 3.16: The InitTest3 class

---

```
package app03;
public class InitTest3 {
    int x = 3;
    int y;
    // instance initialization code
    public InitTest3() {
        init();
    }
    public InitTest3(int x) {
        this.x = x;
        init();
    }
    private void init() {
        y = x * 2;
        System.out.println(y);
    }
    // static initialization code
    static {
```

```
        System.out.println("Static initialization");
    }
    public static void main(String[] args) {
        InitTest3 test = new InitTest3();
        InitTest3 moreTest = new InitTest3();
    }
}
```

---

Note that the **InitTest3** class is preferable because the calls to the **init** method from the constructors make the initialization code more obvious than if it is in an initialization block.

## 3.20 The Garbage Collector

In several examples so far, I have shown you how to create objects using the **new** keyword, but you have never seen code that explicitly destroys unused objects to release memory space. If you are a C++ programmer you may have wondered if I had shown flawed code, because in C++ you must destroy objects after use.

Java comes with a garbage collector, which destroys unused objects and frees memory. Unused objects are defined as objects that are no longer referenced or objects whose references are already out of scope.

With this feature, Java becomes much easier than C++ because Java programmers do not need to worry about reclaiming memory space. This, however, does not entail that you may create objects as many as you want because memory is (still) limited and it takes some time for the garbage collector to start. That's right, you can still run out of memory.

## Self Test

**1 Question** ?

What is the correct syntax for a method?

A. *returnType methodName (listOfArguments)*

B. *methodName (listOfArguments)* { return *returnType* }

C. *returnType methodName* [*listOfArguments*]

D. None of the above

**2 Question** ?

Which of the following are valid components of a Java class?

A. constructors

B. methods

C. fields

D. A return type

**3 Question** ?

What keyword indicates that a method returns no value?

A. null

B. nothing

C. void

D. return

**4 Question** ?

Which of the following are valid method names? (Choose all that apply)

A. instanceOf

B. $createTempObject

C. return_old_string

D. implements

E. divide-by-two

**5 Question** ?

Which of the following methods will make a class executable? (Choose all that apply)

A. protected static void main(String[] args)

B. static public void main(String[] args)

C. void public static main(String[] args)

D. public static void main(String[] args)

E. protected static void main(String args)

**6 Question** ?

Which of the following statements about constructors and or methods are true? (Choose all that apply)

A. Constructors are like methods except that constructors have no return value.

B. Constructors with no return value must use void in its signature.

C. Constructors are like methods except that constructors must be public.

D. A method can be private but a constructor cannot.

E. Protected constructors are not allowed.

**7 Question** ?

Given

```
package com.example;
public class Descriptor {
    public static void main(String[] args) {
        Object object;
        System.out.println(object);
    }
}
```

Which of the following statements are false?

A. The code prints **null** on the console.

B. The code will not compile because object has not been initialized.

C. The code will not compile because it does not import java.lang.Object.

D. The code will compile but will raise an error when run.

**8 Question** ?

Consider the following code fragment with named regions R, S, T, U, V and X.

```
// ------- R --------
for (int m = 0; m < 5; m++) {
  // ------- S --------
    for (int n = 0; n < 3; n++) {
      // ------- T --------
        System.out.println(m);
        System.out.println(n);
      // ------- U --------
    }
  // ------- V --------
}
// ------- X --------
```

Which of the following statements are true?

A. *m* and *n* can be used in region R.

B. *m* and *n* can be used in region T.

C. *m* and *n* can be used in region U.

D. *m* and *n* can be used in region V.

E. *m* can be used in regions S, T, U and V.

F. *m* and *n* are out of scope in Region X.

**9 Question** ?

Given the following code

```
class HttpServer {
    public void service() { ... }
    public void service(int port) { ... }
}

class HttpRequest {
    public int getMethod();
    public String getMethod();
}

class HttpResponse {
    public String getParameter(String name) { ... }
    protected String getParameter(String name, boolean all) { ... }
}
```

Which class(es) feature method overloading?

A. HttpServer

B. HttpRequest

C. HttpResponse

D. None of the above.

**10 Question** ?

Given

```
 1.  class Animal {
 2.      public void walk() {
 3.      }
 4.  }
 5.
 6.  public class Printer {
 7.      public static void main(String[] args) {
 8.          Animal animal1 = new Animal();
 9.          Animal animal2 = new Animal();
10.          animal1 = null;
11.          System.out.println(animal1);
12.          animal2 = animal1;
13.          System.out.println(animal2);
14.      }
15.  }
```

Which of the following statements are true?

A. The object referenced by animal1 is eligible for garbage collection on line 11.

B. The object referenced by animal2 is eligible for garbage collection on line 13.

C. Two objects will be garbage collected on line 14.

D. One object will be garbage collected on line 12.

## Answers

**1** **A**.

The signature of a method has a return type followed by the method name and the list of arguments.

**2** **A, B, C**.

A class may contain one or more constructors, zero or more methods and zero or more fields. It does not need a return type.

**3** **C**.

The keyword **void** indicates a method has no return value.

4 **A, B, C**.

**instanceof** is a reserved keyword and cannot be used as a method name, but instanceOf is valid. B and C are legal because a method name may contain $ and _ characters. D is invalid because **implements** is a reserved keyword in Java. E is illegal because it contains hyphens.

5 **B, D**.

The **main** method must be public and static, return no value and take a String array argument. The public and static modifiers are interchangeable.

6 **A**.

Constructors cannot have a return value, not even void. Constructors can be public, protected or private.

7 **A, C, D**.

The class will not compile as object has not been initialized. As a result, nothing can be executed. You can use members of the **java.lang** package without explicitly importing them.

8 **B, C, E, F**.

*m* is visible in Regions S, T, U and V. *n* is visible in Regions T and U.

9 **A, C**.

Method overloading refers to multiple methods having the same name and different sets of arguments. In HttpRequest, both methods have the same argument set and generate a compile error.

10 **A, B**.

An object is eligible for garbage collection if it is no longer referenced by any variable. Assigning null to an reference variable decrements the number of references to the object. Therefore, A and B are correct.

C and D are incorrect because there is no guarantee when or if an object will be garbage-collected. The garbage collector runs on a low-priority thread and an intelligent garbage collector will not start destroying objects unless the heap is close to full.