3 Methods and encapsulation

Exam objectives covered in this chapter	What you need to know
[1.1] Define the scope of variables.	Variables can have multiple scopes: class, instance, method, and local. Accessibility of a variable in a given scope.
[2.4] Explain an object's life cycle.	Difference between when an object is declared, initialized, accessible, and eligible to be collected by Java's garbage collection. Garbage collection in Java.
[6.1] Create methods with arguments and return values.	Creation of methods with correct return types and method argument lists.
[6.3] Create an overloaded method.	Creation of methods with the same names, but a different set of argument lists.
[6.4] Differentiate between default and user-defined constructors.	A default constructor isn't the same as a no-argument constructor. Java defines a no-argument constructor when no user-defined constructors are created. User-defined constructors can be overloaded.
[6.5] Create and overload constructors.	Like regular methods, constructors can be overloaded.
[2.3] Read or write to object fields.	Object fields can be read and written to by using both instance variables and methods. Access modifiers determine the variables and methods that can be accessed to read from or write to object fields.

Exam objectives covered in this chapter	What you need to know
[2.5] Call methods on objects.	The correct notation to call methods on an object. Access modifiers affect the methods that can be called using a reference variable. Methods may or may not change the value of instance variables. Nonstatic methods can't be called on uninitialized objects.
[6.7] Apply encapsulation principles to a class.	Need for and benefits of encapsulation. Definition of classes that correctly implement the encapsulation principle.
[6.8] Determine the effect upon object references and primitive values when they're passed into methods that change the values.	Object references and primitives are treated in a different manner when passed into methods. Unlike reference variables, the values of primitives are never changed in the calling method when they're passed to methods.

Take a look around, and you'll find multiple examples of *well-encapsulated objects*. For instance, most of us use the services of a bank, which applies a set of well-defined processes that enable us to secure our money and valuables (a bank vault). The bank may require input from us to execute some of its processes, such as depositing money into our accounts. But the bank may or may not inform us about the results of other processes; for example, it may inform us about an account balance after a transaction, but it likely won't inform us about its recruitment plans for new employees.

In Java, you can compare a bank to a well-encapsulated class and the bank processes to Java methods. In this analogy, your money and valuables are like object fields in Java. You can also compare inputs that a bank process requires to Java's method parameters, and the bank process result to a Java method's return value. Finally, you can compare the set of steps that a bank executes when it opens a bank account to constructors in Java.

In the exam, you must answer questions about methods and encapsulation. This chapter will help you get the correct answers by covering the following:

- Defining the scope of variables
- Explaining an object's life cycle
- Creating methods with primitive and object arguments and return values
- Creating overloaded methods and constructors
- Reading and writing to object fields
- Calling methods on objects
- Applying encapsulation principles to a class

We'll start the chapter by defining the scope of the variables; we'll then cover the class constructors, including the differences between user-defined and default constructors and the ways in which they're useful.

3.1 Scope of variables

[1.1] Define the scope of variables

The scope of a variable specifies its life span. In this section, we'll cover the scopes of the variables and the domains in which they're accessible. Here are the available scopes of variables:

- Local variables (also known as method-local variables)
- Method parameters (also known as method arguments)
- Instance variables (also known as attributes, fields, and nonstatic variables)
- Class variables (also known as static variables)

Let's get started by defining local variables.

3.1.1 Local variables

Local variables are defined within a method. They may or may not be defined within code constructs such as `if-else` constructs, looping constructs, or switch statements. Typically, you'd use local variables to store the intermediate results of a calculation. Compared to the other three previously listed variable scopes, they have the shortest scope (life span).

Look at the following code, in which a local variable `avg` is defined within the method `getAverage()`:

```
class Student {
    private double marks1, marks2, marks3;
    private double maxMarks = 100;
    public double getAverage() {
        double avg = 0;
        avg = ((marks1 + marks2 + marks3) / (maxMarks*3)) * 100;
        return avg;
    }
    public void setAverage(double val) {
        avg = val;
    }
}
```

Instance variables

Local variable avg

This code won't compile because avg is inaccessible outside the method getAverage

As you can see, the variable `avg`, defined locally in the method `getAverage`, can't be accessed outside of it in the method `setAverage`. The scope of this local variable, `avg`, is depicted in figure 3.1. The unshaded area marks where `avg` is accessible and the shaded area is where it won't be available.

NOTE The life span of a variable is determined by its scope. If the scope of a variable is limited to a method, its life span is also limited to that method. You may notice that these terms are used interchangeably.

Let's define another variable, `avg`, local to the `if` block of an `if` statement (code that executes when the `if` condition evaluates to `true`):

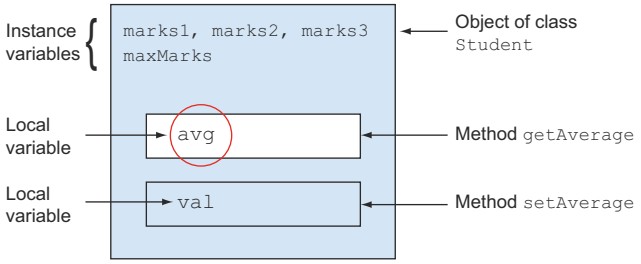


Figure 3.1 You can access the local variable `avg` only within the method `getAverage`.

```
public double getAverage() {
    if (maxMarks > 0) {
        double avg = 0;
        avg = (marks1 + marks2 + marks3)/(maxMarks*3) * 100;
        return avg;
    }
    else {
        avg = 0;
        return avg;
    }
}
```

Variable avg is local to if block

Variable avg can't be accessed because it's local to the if block. Variables local to the if block can't be accessed in the else block

In this case, the scope of the local variable `avg` is reduced to the `if` block of the `if-else` statement defined within the `getAverage` method. The scope of this local variable `avg` is depicted in figure 3.2, where the unshaded area marks where `avg` is accessible, and the shaded part marks the area where it won't be available.

Similarly, loop variables aren't accessible outside of the loop body:

```
public void localVariableInLoop() {
    for (int ctr = 0; ctr < 5; ++ctr) {
        System.out.println(ctr);
    }
    System.out.println(ctr);
}
```

Variable ctr is defined within the for loop

Variable ctr isn't accessible outside the for loop. This line won't compile.

EXAM TIP The local variables topic is a favorite of OCA Java SE 7 Programmer I exam authors. You're likely to be asked a question that seems to be about a rather complex topic, such as inheritance or exception handling, but in fact it'll be testing your knowledge on the scope of a local variable.

The scope of a local variable depends on the location of its declaration within a method. The scope of local variables defined within a loop, `if-else`, or `switch` construct or within a code block (marked with {}) is limited to these constructs.

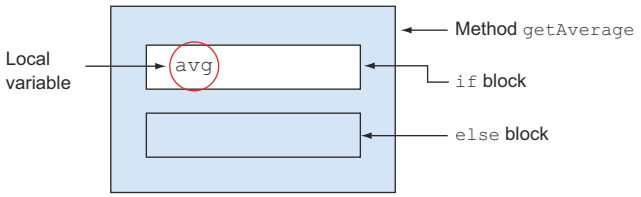


Figure 3.2 The scope of local variable `avg` is part of the `if` statement.

Local variables defined outside any of these constructs are accessible across the complete method.

The next section discusses the scope of method parameters.

3.1.2 Method parameters

The variables that accept values in a method are called *method parameters*. They're accessible only in the method that defines them. In the following example, a method parameter `val` is defined for the method `setTested`:

```
class Phone {
    private boolean tested;
    public void setTested(boolean val) {
        tested = val;
    }
    public boolean isTested() {
        val = false;
        return tested;
    }
}
```

Method parameter val is accessible only in method setTested

Variable val can't be accessed in method isTested

This line of code won't compile

In the previous code, you can access the method parameter `val` only within the method `setTested`. It can't be accessed in any other method.

The scope of the method parameter `val` is depicted in figure 3.3. The unshaded area marks where the variable is accessible, and the shaded part marks where it won't be available.

The scope of a method parameter may be as long as that of a local variable, or longer, but it can never be shorter. The following method, `isPrime`, defines a method parameter, `num`, and two local variables, `result` and `ctr`:

```
boolean isPrime(int num) {
    if (num <= 1) return false;
    boolean result = true;
    for (int ctr = num-1; ctr > 1; ctr--) {
         if (num%ctr == 0) result = false;
    }
    return result;
}
```

Method parameter num

Local variable result

Local variable ctr

The scope of the method parameter `num` is as long as the scope of the local variable `result`. Because the scope of the local variable `ctr` is limited to the `for` block, it's

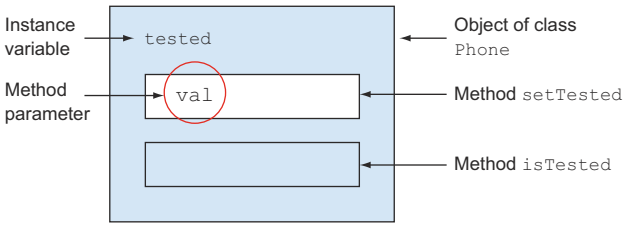


Figure 3.3 The scope of the method parameter `val`, which is defined in method `setTested`

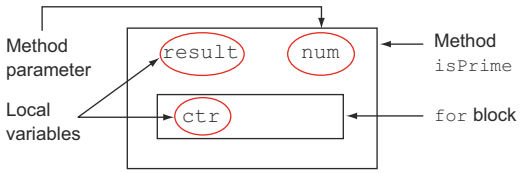


Figure 3.4 Comparison of the scope of method parameters and local variables

shorter than the method parameter `num`. The comparison of the scope of all of these three variables is shown in figure 3.4, where the scope of each variable (defined in an oval) is shown by the rectangle enclosing it.

Let's move on to instance variables, which have a larger scope than method parameters.

3.1.3 Instance variables

Instance is another name for an object. Hence, an *instance variable* is available for the life of an object. An instance variable is declared within a class, outside of all of the methods. It's accessible to all the nonstatic methods defined in a class.

In the following example, the variable `tested` is an instance variable—it's defined within the class `Phone`, outside of all of the methods. It can be accessed by all of the methods of class `Phone`:

```
class Phone {
    private boolean tested;
    public void setTested(boolean val) {
        tested = val;
    }
    public boolean isTested() {
        return tested;
    }
}
```

Instance variable tested

Variable tested is accessible in method setTested

Variable tested is also accessible in method isTested

The scope of the instance variable `tested` is depicted in figure 3.5. As you can see, the variable `tested` is accessible across the object of class `Phone`, represented by the unshaded area. It's accessible in the methods `setTested` and `isTested`.

EXAM TIP The scope of an instance variable is longer than that of a local variable or a method parameter.

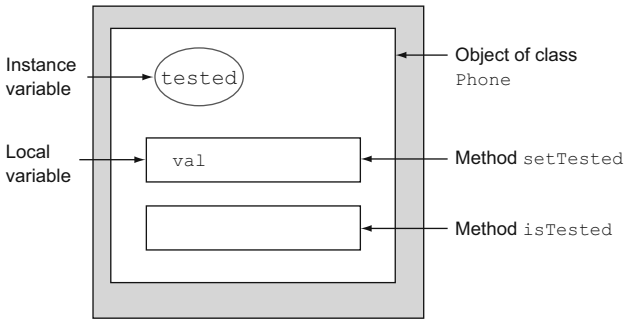


Figure 3.5 The instance variable `tested` is accessible across the object of class `Phone`

Class variables, covered in the next section, have the largest scope among all types of variables.

3.1.4 Class variables

A *class variable* is defined by using the keyword `static`. A class variable belongs to a class, not to individual objects of the class. A class variable is shared across all objects—objects don't have a separate copy of the class variables.

You don't even need an object to access a class variable. It can be accessed by using the name of the class in which it's defined:

```
package com.mobile;
class Phone {
    static boolean softKeyboard = true;
}
```

Class variable softKeyboard

Let's try to access this variable in another class:

```
package com.mobile;
class TestPhone {
    public static void main(String[] args) {
        Phone.softKeyboard = false;

        Phone p1 = new Phone();
        Phone p2 = new Phone();

        System.out.println(p1.softKeyboard);
        System.out.println(p2.softKeyboard);

        p1.softKeyboard = true;

        System.out.println(p1.softKeyboard);
        System.out.println(p2.softKeyboard);
    }
}
```

Accesses the class variable by using the name of the class. It can be accessed even before any of the class's objects exist.

Prints false. A class variable can be read by using objects of the class.

Prints false.

A change in the value of this variable will be reflected when the variable is accessed via objects or class name.

Prints true.

Prints true.

As you can see in the previous code, the class variable `softKeyboard` is accessible using all of the following:

- `Phone.softKeyboard`
- `p1.softKeyboard`
- `p2.softKeyboard`

It doesn't matter whether you use the name of the class (`Phone`) or an object (`p1`) to access a class variable. You can change the value of a class variable using either of them because they all refer to a single shared copy.

The scope of the class variable `softKeyboard` is depicted in figure 3.6. As you can see, a single copy of this variable is accessible to all the objects of class `Phone`. The variable `softKeyboard` isn't defined within any object of class `Phone`, so it's accessible even without the existence of an object of class `Phone`. The class variable `softKeyboard` is made accessible by the JVM when it loads the `Phone` class into memory. The scope of the class variable `softKeyboard` depends on its access modifier and that of the `Phone`

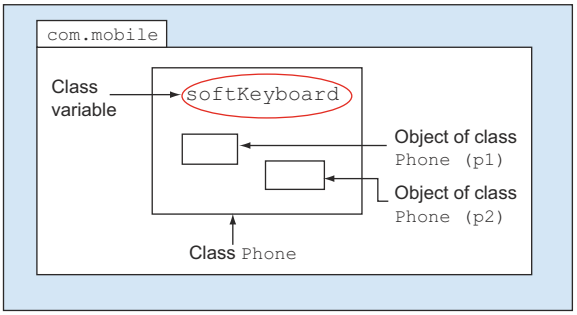


Figure 3.6 The scope of the class variable `softKeyboard` is limited to the package `com.mobile` because it's defined in class `Phone`, which is defined with default access. The class variable `softKeyboard` is shared and accessible across all objects of class `Phone`.

class. Because the class `Phone` and the class variable `softKeyboard` are defined using default access, they're accessible only within the package `com.mobile`.

COMPARING THE USE OF VARIABLES IN DIFFERENT SCOPES

Here is a quick comparison of the use of the local variables, method parameters, instance variables, and class variables:

- Local variables are defined within a method and are normally used to store the intermediate results of a calculation.
- Method parameters are used to pass values to a method. These values can be manipulated and may also be stored as the state of an object by assigning them to instance variables.
- Instance variables are used to store the state of an object. These are the values that need to be accessed by multiple methods.
- Class variables are used to store values that should be shared by all the objects of a class.

3.1.5 Overlapping variable scopes

In the previous sections on local variables, method parameters, instance variables, and class variables, did you notice that some of the variables are accessible in multiple places within an object? For example, all four variables will be accessible in a loop within a method.

This overlapping scope is shown in figure 3.7. The variables are defined in ovals and are accessible within all methods and blocks, as illustrated by their enclosing rectangles.

As shown in figure 3.7, an individual copy of `classVariable` can be accessed and shared by multiple objects (`object1` and `object2`) of a class. Both `object1` and `object2` have their own copy of the instance variable `instanceVariable1`, so `instanceVariable1` is accessible across all the methods of `object1`. The methods `method1` and `method2` have their own copies of `localVariable` and `methodParameter` when used with `object1` and `object2`.

The scope of `instanceVariable1` overlaps with the scope of `localVariable` and `methodParameter`, defined in `method1`. Hence, all three of these variables (`instanceVariable1`, `localVariable`, and `methodParameter`) can access each other in this

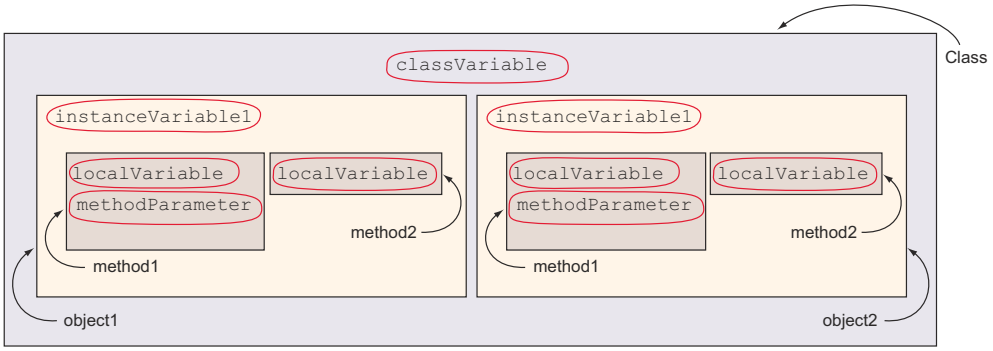


Figure 3.7 The scopes of variables can overlap

overlapped area. But `instanceVariable1` can't access `localVariable` and `methodParameter` outside `method1`.

COMPARING THE SCOPE OF VARIABLES

Figure 3.8 compares the life spans of local variables, method parameters, instance variables, and class variables.

As you can see in figure 3.8, local variables have the shortest scope or life span, and class variables have the longest scope or life span.

EXAM TIP Different local variables can have different scopes. The scope of local variables may be shorter than or as long as the scope of method parameters. The scope of local variables is less than the duration of a method if they're declared in a sub-block (within braces {}) in a method. This sub-block can be an `if` statement, a `switch` construct, a loop, or a `try-catch` block (discussed in chapter 7).

VARIABLES WITH THE SAME NAME IN DIFFERENT SCOPES

The fact that the scopes of variables overlap results in interesting combinations of variables within different scopes but with the same names. Some rules are necessary to

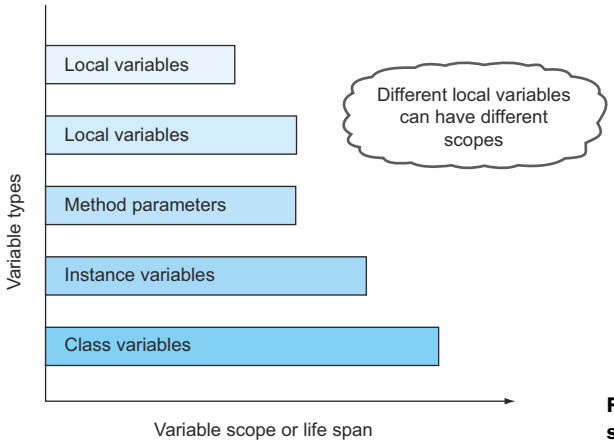


Figure 3.8 Comparing the scope, or life span, of all four variables

prevent conflicts. In particular, you can't define a `static` variable and an instance variable with the same name in a class:

```
class MyPhone {
    static boolean softKeyboard = true;
    boolean softKeyboard = true;
}
```

Won't compile. Class variable and instance variable can't be defined using the same name in a class

Similarly, local variables and method parameters can't be defined with the same name. The following code defines a method parameter and a local variable with the same name, so it won't compile:

```
void myMethod(int weight) {
    int weight = 10;
}
```

Won't compile. Method parameter and local variable can't be defined using the same name in a method

A class can define local variables with the same name as the instance or class variables. The following code defines a class variable and a local variable, `softKeyboard`, with the same name, and an instance variable and a local variable, `phoneNumber`, with the same name, which is acceptable:

```
class MyPhone {
    static boolean softKeyboard = true;
    String phoneNumber;

    void myMethod() {
        boolean softKeyboard = true;
        String phoneNumber;
    }
}
```

Class variable softKeyboard

Instance variable phoneNumber

Local variable softKeyboard can coexist with class variable softKeyboard

Local variable phoneNumber can coexist with instance variable phoneNumber

What happens when you assign a value to a local variable that has the same name as an instance variable? Does the instance variable reflect this modified value? This question provides the food for thought in this chapter's first Twist in the Tale exercise. It should help you remember what happens when you assign a value to a local variable when an instance variable already exists with the same name in the class (answer in the appendix).

Twist in the Tale 3.1

The class `Phone` defines a local variable and an instance variable, `phoneNumber`, with the same name. Examine the definition of the method `setNumber`. Execute the class on your system and select the correct output of the class `TestPhone` from the given options:

```
class Phone {
    String phoneNumber = "123456789";
    void setNumber () {
        String phoneNumber;
        phoneNumber = "987654321";
    }
}
class TestPhone {
```

```
    public static void main(String[] args) {
        Phone p1 = new Phone();
        p1.setNumber();
        System.out.println (p1.phoneNumber);
    }
}
```

a 123456789
b 987654321
c No output
d The class `Phone` will not compile.

In this section, you worked with variables in different scopes. When variables go out of scope, they're no longer accessible by the remaining code. In the next section, you'll see how an object is created and made accessible and then inaccessible.

3.2 Object's life cycle

[2.4] Explain an object's life cycle

The OCA Java SE 7 Programmer I exam will test your understanding of how to determine when an object is or isn't accessible. The exam also tests your ability to determine the total number of objects that are accessible at a particular line of code. Primitives aren't objects, so they're not relevant in this section.

Unlike some other programming languages, such as C, Java doesn't allow you to allocate or deallocate memory yourself when you create or destroy objects. Java manages memory for allocating objects and reclaiming the memory occupied by unused objects.

The task of reclaiming unused memory is taken care of by Java's garbage collector, which is a low-priority thread. It runs periodically and frees up space occupied by unused objects.

Java also provides a method called `finalize`, which is accessible to all of the classes. The method `finalize` is defined in the class `java.lang.Object`, which is the base class of all Java classes. All Java classes can override the method `finalize`, which executes just before an object is garbage collected. In theory, you can use this method to free up resources being used by an object, although doing so isn't recommended because its execution is not guaranteed to happen.

An object's life cycle starts when it's created and lasts until it goes out of its scope or is no longer referenced by a variable. When an object is accessible, it can be referenced by a variable and other classes can use it by calling its methods and accessing its variables. I'll discuss these stages in detail in the following subsections.

3.2.1 An object is born

An object comes into the picture when you use the keyword operator `new`. You can initialize a reference variable with this object. Note the difference between declaring a

variable and initializing it. The following is an example of a class `Person` and another class `ObjectLifeCycle`:

```
class Person {}
class ObjectLifeCycle {
    Person person;
}
```

Class Person

Declaring a reference variable of type Person

In the previous code, no objects of class `Person` are created in the class `ObjectLifeCycle`; it declares only a variable of type `Person`. An object is created when a reference variable is initialized:

```
class ObjectLifeCycle2 {
    Person person = new Person();
}
```

Declaring and initializing a variable of type Person

The difference in variable declaration and object creation is illustrated in figure 3.9, where you can compare a baby name to a reference variable and a real baby to an object. The left box in figure 3.9 represents variable declaration, because the baby hasn't been born yet. The right box in figure 3.9 represents object creation.

Syntactically, an object comes into being by using the `new` operator. Because `Strings` can also be initialized using the `=` operator, the following code is a valid example of `String` objects being created:

```
class ObjectLifeCycle3 {
    String obj1 = new String("eJava");
    String obj2 = "Guru";
}
```

This class creates two objects of the class String

String object referenced by obj1

Another String object referenced by obj2

What happens when you create a new object without assigning it to any reference variable? Let's create a new object of class `Person` in class `ObjectLifeCycle2` without assigning it to any reference variable (modifications in **`bold`**):

```
class ObjectLifeCycle2 {
    Person person = new Person();
    ObjectLifeCycle2() {
          new Person();
    }
}
```

An unreferenced object

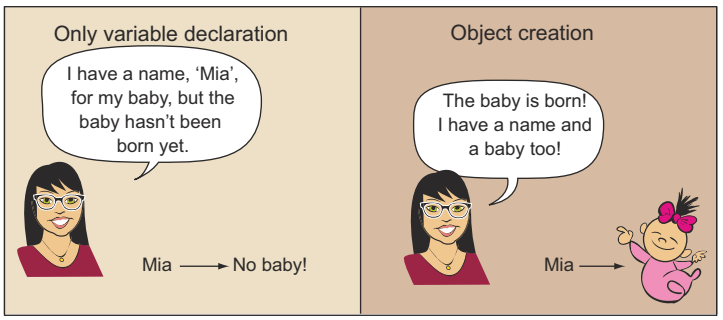


Figure 3.9 The difference between declaring a reference variable and initializing a reference variable

In the previous example, an object of class `Person` is created, but it can't be accessed using any reference variable. Creating an object in this manner will execute the relevant constructors of the class.

EXAM TIP Watch out for a count of the total objects created in any given code—the ones that can be accessed using a variable and the ones that can't be accessed using any variable. The exam may question you on the count of objects created.

In the next section, you'll learn what happens after an object is created.

3.2.2 Object is accessible

Once an object is created, it can be accessed using its reference variable. It remains accessible until it goes out of scope or its reference variable is explicitly set to `null`. Also, if you reassign another object to an initialized reference variable, the previous object becomes inaccessible. You can access and use an object within other classes and methods.

Take a look at the following definition of the class `Exam`:

```
class Exam {
    String name;
    public void setName(String newName) {
        name = newName;
    }
}
```

The class `ObjectLife1` declares a variable of type `Exam`, creates its object, calls its method, sets it to `null`, and then reinitializes it:

```
class ObjectLife1 {
    public static void main(String args[]) {
        Exam myExam = new Exam();
        myExam.setName("OCA Java Programmer 1");
        myExam = null;
        myExam = new Exam();
        myExam.setName("PHP");
    }
}
```

❶ Object creation
❷ Access method
❸ Set reference variable to null
❹ Another object creation
❺ Access method

The previous example creates two objects of class `Exam` using the same reference variable `myExam`. Let's walk through what is happening in the example:

- ❶ creates a reference variable `myExam` and initializes it with an object of class `Exam`.
- ❷ calls method `setName` on the object referenced by the variable `myExam`.
- ❸ assigns a value `null` to the reference variable `myExam` such that the object referenced by this variable is no longer accessible.
- ❹ creates a new object of class `Exam` and assigns it to the reference variable `myExam`.
- ❺ calls method `setName`, on the second `Exam` object, created in method `main`.

When ❹ creates another object of class `Exam` and assigns it to the variable `myExam`, what happens to the first object created by ❶? Because the first object can no longer be accessed using any variable, it's considered garbage by Java and deemed eligible to be sent to the garbage bin by Java's garbage collector. As mentioned earlier, the garbage collector is a low-priority thread that reclaims the space used by unused or unreferenced objects in Java.

So what happens when an object become inaccessible? You'll find out in the next section.

3.2.3 Object is inaccessible

In the previous section, you learned that an object can become inaccessible if it can no longer be referenced by any variable. An object can also become inaccessible if it goes out of scope:

```
public void myMethod() {
    int result = 88;
    if (result > 78) {
            Exam myExam1 = new Exam();
            myExam1.setName("Android");
    }
    else {
            Exam myExam2 = new Exam();
            myExam2.setName("MySQL");
    }
}
```

❶ Scope of local variable myExam1

❷ Start of else block

❸ End of else block

The variable `myExam1` is a local variable defined within the `if` block. Its scope starts from the line where it is declared until the end of the `if` block, marked with a closing brace, at ❶. After this closing brace, the object referred by the variable `myExam1` is no longer accessible. It goes out of scope and is marked as eligible for garbage collection by Java's garbage collector. Similarly, the object referred to by the variable `myExam2` becomes inaccessible at the end of the `else` block, marked with a closing brace, at ❸.

EXAM TIP An object is marked as eligible to be garbage collected when it can no longer be accessed, which can happen when the object goes out of scope. It can also happen when an object's reference variable is assigned an explicit `null` value or is reinitialized.

In the OCA Java SE 7 Programmer I exam, you're likely to answer questions on garbage collection for code that has multiple variable declarations and initializations. The exam may query you on the total number of objects that are eligible for garbage collection after a particular line of code.

As previously mentioned, the garbage collector is a low-priority thread that marks the objects eligible for garbage collection in the JVM and then clears the memory of these objects. You can determine only which objects are *eligible* to be garbage collected. You can *never* determine when a particular object *will* be garbage collected. A user can't control or determine the execution of a garbage collector. It's controlled by the JVM.

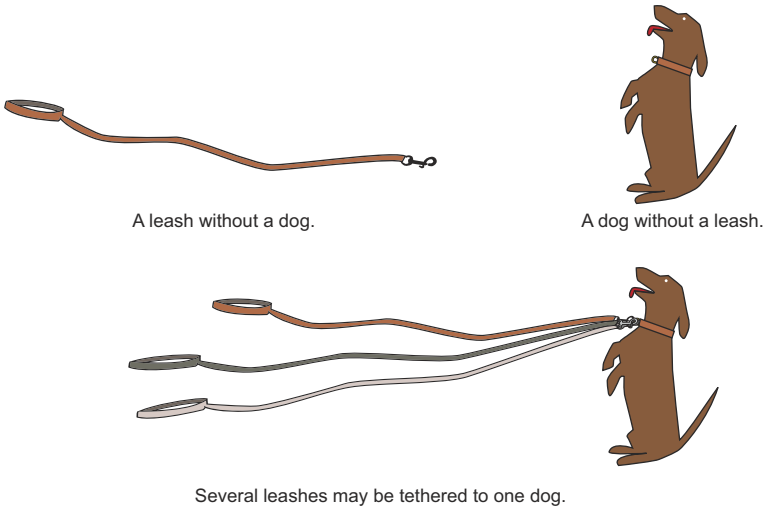


Figure 3.10 Comparing object reference variables and objects to dog leashes and leashed and unleashed dogs

EXAM TIP You can be sure only about which objects are marked for garbage collection. You can never be sure exactly when the object will be garbage collected. Watch for questions with wordings such as "which objects are sure to be collected during the next GC cycle," for which the real answer can never be known.

Let's revisit the dog leash and dog analogy I used in chapter 2 to define object reference variables. In figure 3.10, you can compare an object reference variable with a dog leash and a dog with an object. Review the following comparisons, which will help you to understand the life cycle of an object:

- An uninitialized reference variable can be compared to a dog leash without a dog.
- An initialized reference variable can be compared to a leashed dog.
- An unreferenced object can be compared to an unleashed dog.

You can compare Java's garbage collector to animal control. The way animal control picks up untethered dogs is like how Java's garbage collector reclaims the memory used by unreferenced objects.

Now that you're familiar with an object's life cycle, you can create methods that accept primitive data types and objects as method arguments; these methods return a value, which can be either a primitive data type or an object.

3.3 Create methods with arguments and return values

[6.1] Create methods with arguments and return values

In this section, you'll work with the definitions of methods, which may or may not accept input parameters and may or may not return any values.

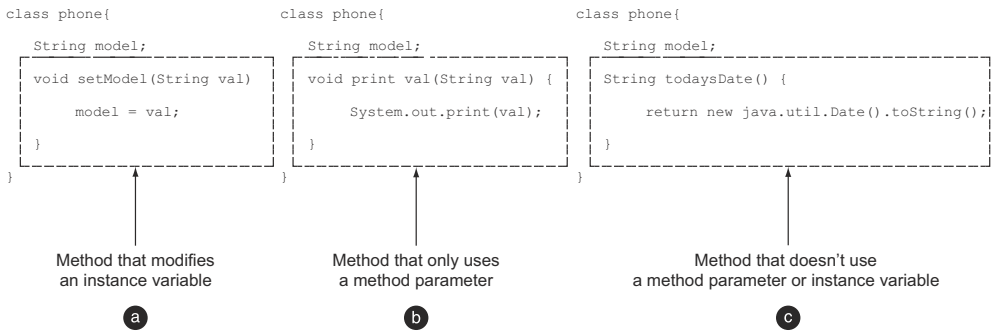


Figure 3.11 Different types of methods

A method is a group of statements identified with a name. Methods are used to define the behavior of an object. A method can perform different functions, as shown in figure 3.11:

- **a** The method `setModel` accesses and modifies the state of the object `Phone`.
- **b** The method `printVal` uses only the method parameter passed to it.
- **c** The method `todaysDate` accesses another class (`java.util.Date`) from the Java API and returns its `String` presentation.

In the following subsections, you'll learn about the components of a method:

- Return type
- Method parameters
- `return` statement
- Access modifiers (covered in chapter 1)
- Nonaccess modifiers (covered in chapter 1)

Figure 3.12 shows the code of a method accepting method parameters and defining a return type and a `return` statement.

Let's get started with a discussion of the return type of a method.

3.3.1 Return type of a method

In this section, you'll work with the return type of a method. The return type of a method states the type of value that a method will return.

A method may or may not return a value. One that doesn't return a value has a return type of `void`. A method can return a primitive value or an object of any class. The name of the return type can be any of the eight primitive types defined in Java, the name of any class, or an interface.

In the following code, the method `setWeight` doesn't return any value, and the method `getWeight` returns a value:

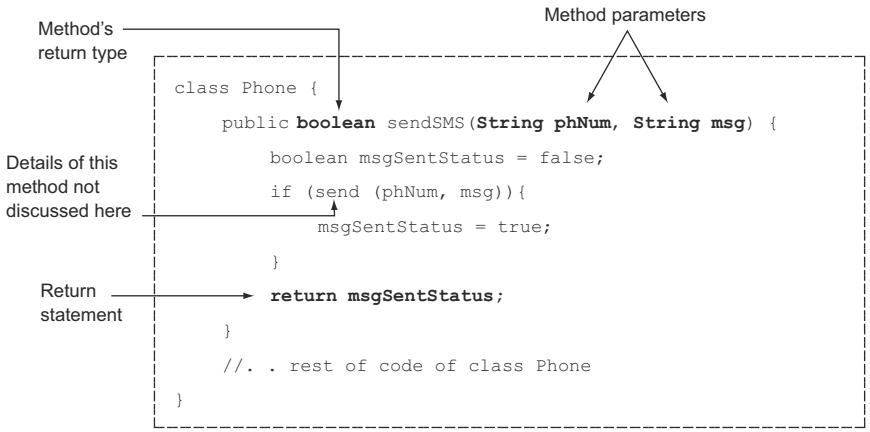


Figure 3.12 An example of a method that accepts method parameters and defines a return type and a `return` statement

```
class Phone {
    double weight;
    void setWeight(double val) {
            weight =val;
    }
    double getWeight() {
            return weight;
    }
}
```

Method with return type void

Method with return type double

If a method doesn't return a value, you can't assign the result of that method to a variable. What do you think is the output of the following class `EJavaTestMethods`, which uses the previous class `Phone`?

```
class EJavaTestMethods {
    public static void main(String args[]) {
            Phone p = new Phone();
            double newWeight = p.setWeight(20.0);
    }
}
```

Because the method setWeight doesn't return any value, this line won't compile.

The previous code won't compile because the method `setWeight` doesn't return a value. Its return type is `void`. Because the method `setWeight` doesn't return a value, there's nothing to be assigned to the variable `newWeight`, so the code fails to compile.

If a method returns a value, the calling method may or may not bother to store the returned value from a method in a variable. Look at the following code:

```
class EJavaTestMethods2 {
    public static void main(String args[]) {
            Phone p = new Phone();
            p.getWeight();
    }
}
```

Method getWeight returns a double value, but this value isn't assigned to any variable

In the previous example, the value returned by the method `getWeight` isn't assigned to any variable, which isn't an issue for the Java compiler. The compiler will happily compile the code for you.

EXAM TIP You can optionally assign the value returned by a method to a variable. If you don't assign the returned value from a method, it's neither a compilation error nor a runtime exception.

The variable you use to accept the returned value from a method must be compatible with the returned value. Consider this example:

```
class EJavaTestMethods2 {
    public static void main(String args[]) {
            Phone p = new Phone();
            double newWeight = p.getWeight();
            int newWeight2 = p.getWeight();
    }
}
```

❶ Will compile

❷ Won't compile

In the preceding code, ❶ will compile successfully because the return type of the method `getWeight` is `double` and the type of the variable `newWeight` is also `double`. But ❷ won't compile because the `double` value returned from method `getWeight` can't be assigned to variable `newWeight2`, which is of type `int`.

We've discussed how to transfer a value out from a method. To transfer value into a method, you can use method arguments.

3.3.2 Method parameters

Method parameters are the variables that appear in the definition of a method and specify the type and number of values that a method can accept. In figure 3.13, the variables `phNum` and `msg` are the method parameters.

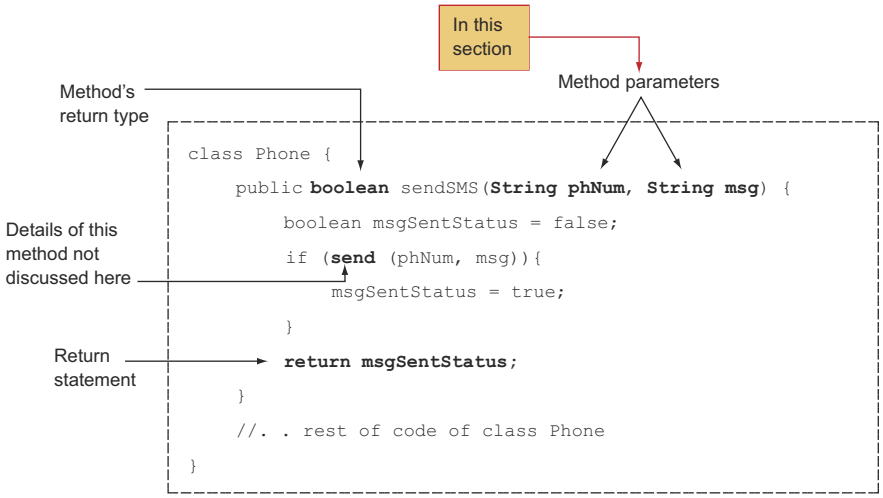


Figure 3.13 An example of a method that accepts method parameters and defines a return type and a `return` statement.

You can pass multiple values to a method as input. Theoretically, no limit exists on the number of method parameters that can be defined by a method, but practically it's not a good idea to define more than five or six method parameters. It's cumbersome to use a method with too many method parameters because you have to cross-check their types and purposes multiple times to ensure that you're passing the right values at the right positions.

NOTE Though the terms *method parameters* and *method arguments* are not the same, you may have noticed that they're used interchangeably by many programmers. *Method parameters* are the variables that appear in the definition of a method. *Method arguments* are the actual values that are passed to a method while executing it. In figure 3.13, variables `phNum` and `msg` are method parameters. If you execute this method as `sendMsg("123456", "Hello")`, then the `String` values `"123456"` and `"Hello"` are method arguments. As you know, you can pass literal values or variables to a method. Thus, method arguments can be literal values or variables.

A method may accept zero or multiple method arguments. The following example accepts two `int` values and returns their average as a `double` value:

```
double calcAverage(int marks1, int marks2) {
    double avg = 0;
    avg = (marks1 + marks2)/2.0;
    return avg;
}
```

Multiple method parameters: marks1 and marks2

The following example shows a method that doesn't accept any method parameters:

```
void printHello() {
    System.out.println("Hello");
}
```

If a method doesn't accept any parameters, the parentheses that follow the name of the method are empty. Because the keyword `void` is used to specify that a method doesn't return a value, you may think it's correct to use the keyword `void` to specify that a method doesn't accept any method parameters, but this is incorrect. The following is an invalid definition of a method that accepts no parameters:

```
void printHello(void) {
    System.out.println("Hello");
}
```

Won't compile

You can define a parameter that can accept variable arguments (varargs) in your methods. Following is an example of class `Employee`, which defines a method `daysOffWork` that accepts variable arguments:

```
class Employee {
    public int daysOffWork(int... days) {
          int daysOff = 0;
```

```
            for (int i = 0; i < days.length; i++)
                daysOff += days[i];
            return daysOff;
    }
}
```

The ellipsis (...) that follows the data type indicates that the method parameter `days` may be passed an array or multiple comma-separated values. Re-examine the preceding code example and note the usage of the variable `days` in the method `daysOffWork`—it works like an array. When you define a variable-length argument for a method, Java creates an array behind the scenes to implement it.

You can define only one variable argument in a parameter list, and it should be the last variable in the parameter list. If you don't comply with these two rules, your code won't compile:

```
class Employee {
    public int daysOffWork(String... months, int... days) {
            int daysOff = 0;
            for (int i = 0; i < days.length; i++)
                daysOff += days[i];
            return daysOff;
    }
}
```

Won't compile. You can't define multiple variables that can accept variable arguments.

If your method defines multiple method parameters, the variable that accepts variable arguments must be the last one in the parameter list:

```
class Employee {
    public int daysOffWork(int... days, String year) {
            int daysOff = 0;
            for (int i = 0; i < days.length; i++)
                daysOff += days[i];
            return daysOff;
    }
}
```

Won't compile. If multiple parameters are defined, the variable argument must be the last in the list.

EXAM TIP In the OCA exam, you may be questioned on the valid return types for a method that doesn't accept any method parameters. Note that there are no valid or invalid combinations of the number and type of method parameters that can be passed to a method and the value that it can return. They're independent of each other.

You can pass any type and number of parameters to a method, including primitives, objects of a class, or objects referenced by an interface.

RULES TO REMEMBER

Points to note with respect to defining method parameters:

- You can define multiple parameters for a method.
- The method parameter can be a primitive type or objects referenced by a class or referenced by an interface.

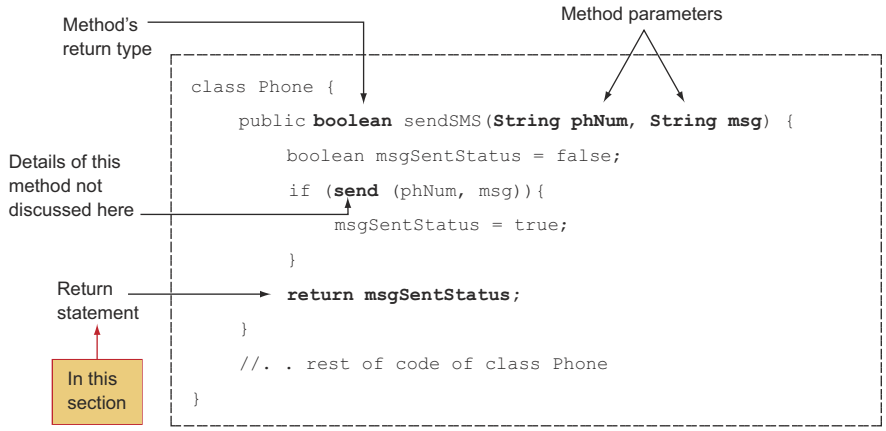


Figure 3.14 An example of a method that accepts method parameters and defines a return type and a `return` statement

- The method's parameters are separated by commas.
- Each method parameter is preceded by the name of its type. Each method parameter must have an explicit type declared with its name. You can't declare the type once and then list them separated by commas, as you can for variables.

3.3.3 Return statement

A `return` statement is used to exit from a method, with or without a value. For methods that define a return type, the `return` statement must be immediately followed by a return value. For methods that don't return a value, the `return` statement can be used without a return value to exit a method. Figure 3.14 illustrates the use of a `return` statement.

In this example, we'll revisit the previous example of method `calcAverage`, which returns a value of type `double`, using a `return` statement:

```
double calcAverage(int marks1, int marks2) {
    double avg = 0;
    avg = (marks1 + marks2)/2.0;
    return avg;
}
```

return statement

The methods that don't return a value (return type is `void`) are not required to define a `return` statement:

```
void setWeight(double val) {
    weight = val;
}
```

return statement not required for methods with return type void

But you can use the `return` statement in a method even if it doesn't return a value. Usually this statement is used to define an early exit from a method:

```
void setWeight(double val) {
    if (val < -1) return;
    weight = val;
}
```

This code compiles successfully. Control exits the method if this condition is true.

Method with return type void can use return statement.

Also, the `return` statement must be the last statement to *execute* in a method, if present. The `return` statement transfers control out of the method, which means that there's no point in defining any code after it. The compiler will fail to compile such code:

```
void setWeight(double val) {
    return;
    weight = val;
}
```

The return statement must be the last statement to execute in a method

This code can't execute due to the presence of the return statement before it

Note that there's a difference in the `return` statement being the last statement in a method and being the last statement to execute in a method. The `return` statement need not be the *last statement* in a method, but it must be the *last statement to execute* in a method:

```
void setWeight(double val) {
    if (val < 0)
        return;                    <-1
    else
        weight = val;
}
```

In the preceding example, the `return` statement ❶ isn't the last statement in this method. But it's the last statement to execute for method parameter values of less than zero.

RULES TO REMEMBER WHEN DEFINING A RETURN STATEMENT

Here are some items to note when defining a `return` statement:

- For a method that returns a value, the `return` statement must be followed immediately by a value.
- For a method that doesn't return a value (return type is `void`), the `return` statement must *not* be followed by a return value.
- If the compiler determines that a `return` statement isn't the last statement to *execute* in a method, the method will fail to compile.

RULES TO REMEMBER FOR DEFINING METHODS WITH ARGUMENTS AND RETURN TYPES

The previous list of rules will help you define a `return` statement in a method. Following is a set of rules to remember for the complete exam objective 6.1, "Create methods with arguments and return values":

- A method may or may not accept method arguments.
- A method may or may not return a value.

- A method returns a value by using the keyword `return`, followed by the name of a variable or an expression whose value is passed back to the calling method.
- The returned value from a method may or may not be assigned to a variable. If the value is assigned to a variable, the variable type must be compatible with the type of the return value.
- A `return` statement must be the last statement in a method. Statements placed after the `return` statements aren't reachable and fail to compile.
- A method can be an instance method (nonstatic) or a class method (static).
- A method can take zero or more parameters but can return only zero or one values.

Do you think we've covered all the rules for defining a method? Not yet! Do you think you can define multiple methods in a class with the same name? You can, but you need to be aware of some additional rules, which are discussed in the next section.

3.4 Create an overloaded method

[6.3] Create an overloaded method

Overloaded methods are methods with the same name but different method parameter lists. In this section, you'll learn how to create and use overloaded methods.

Imagine that you're delivering a lecture and need to instruct the audience to take notes using paper, a smart phone, or a laptop—whichever is available to them for the day. One way to do this is give the audience a list of instructions as follows:

- Take notes using paper.
- Take notes using smartphones.
- Take notes using laptops.

Another method is to instruct them to "take notes" and then provide them with the paper, a smartphone, or a laptop they're supposed to use. Apart from the simplicity of the latter method, it also gives you the flexibility to add other media on which to take notes (such as one's hand, some cloth, or the wall) without needing to remember the list of all the instructions.

This second approach, providing one set of instructions (with the same name) but a different set of input values can be compared to overloaded methods in Java, as shown in figure 3.15.

Again, overloaded methods are methods that are defined in the same class with the same name but with different method argument lists. As shown in figure 3.15, overloaded methods make it easier to add methods with similar functionality that work with different sets of input values.

Let's work with an example from the Java API classes that we all use frequently: `System.out.println()`. The `println` method accepts multiple types of method parameters:

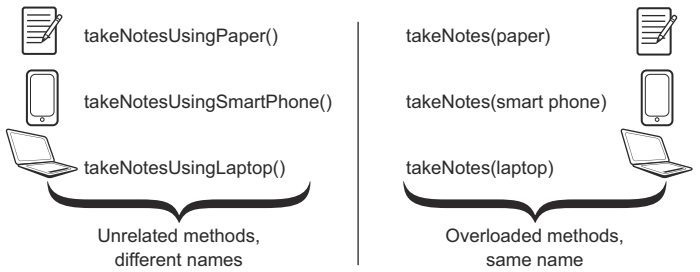


Figure 3.15 Real-life examples of overloaded methods

```
int intVal = 10;
boolean boolVal = false;
String name = "eJava";

System.out.println(intVal);
System.out.println(boolVal);
System.out.println(name);
```

Prints an int value

Prints a boolean value

Prints a string value

When you use the method `println`, you know that whatever you pass to it as a method argument will be printed to the console. Wouldn't it be crazy to use methods like `printlnInt`, `printlnBool`, and `printlnString` for the same functionality? I think so, too.

RULES TO REMEMBER FOR DEFINING OVERLOADED METHODS

Here are a few rules for defining overloaded methods:

- Overloaded methods must have different method parameters from one another.
- Overloaded methods may or may not define a different return type.
- Overloaded methods may or may not define different access modifiers.
- Overloaded methods can't be defined by only changing their return type or access modifiers.

Next, I'll describe in detail the method parameters that are passed to overloaded methods, their return types, and access modifiers.

3.4.1 Argument list

Overloaded methods accept different lists of arguments. The argument lists can differ in terms of any of the following:

- Change in the number of parameters that are accepted
- Change in the types of parameters that are accepted
- Change in the positions of the parameters that are accepted (based on parameter type, not variable names)

Following is an example of the overloaded method `calcAverage`, which accepts different numbers of method parameters:

```
double calcAverage(int marks1, double marks2) {
    return (marks1 + marks2)/2.0;
}
```

Two method arguments

```
double calcAverage(int marks1, int marks2, int marks3) {
    return (marks1 + marks2 + marks3)/3.0;
}
```

Three method arguments

The previous code is an example of the simplest flavor of overloaded methods. You can also define overloaded methods in which the difference in the argument list is in the types of the parameters that are accepted:

```
double calcAverage(int marks1, double marks2) {
    return (marks1 + marks2)/2.0;
}
double calcAverage(char marks1, char marks2) {
    return (marks1 + marks2)/2.0;
}
```

Arguments: int, double

Arguments: char, char

The methods are also correctly overloaded if they change only the positions of the parameters that are passed to them:

```
double calcAverage(double marks1, int marks2) {
    return (marks1 + marks2)/2.0;
}
double calcAverage(int marks1, double marks2) {
    return (marks1 + marks2)/2.0;
}
```

Arguments: double, int

Arguments: int, double

Although you might argue that the arguments being accepted are one and the same, with only their positions differing, the Java compiler treats them as different argument lists. Hence, the previous code is a valid example of overloaded methods.

But an issue arises when you try to execute this method using values that can be passed to both versions of the overloaded methods. In this case, the code will fail to compile:

```
class MyClass {
    double calcAverage(double marks1, int marks2) {
            return (marks1 + marks2)/2.0;
    }
    double calcAverage(int marks1, double marks2) {
            return (marks1 + marks2)/2.0;
    }
    public static void main(String args[]) {
            MyClass myClass = new MyClass();
            myClass.calcAverage(2, 3);
    }
}
```

❶ **Method parameters: double and int**

❷ **Method parameters: int and double**

❸ **Compiler can't determine which overloaded method calcAverage should be called**

In the previous code, ❶ defines the method `calcAverage`, which accepts two method parameters: a `double` and an `int`. ❷ defines the overloaded method `calcAverage`, which accepts two method parameters: an `int` and a `double`. Because an `int` literal value can be passed to a variable of type `double`, literal values `2` and `3` can be passed to

both the overloaded methods declared at ❶ and ❷. Because this method call is dubious, ❸ fails to compile.

3.4.2 Return type

Methods can't be defined as overloaded methods if they differ only in their return types:

```
double calcAverage(int marks1, int marks2) {
    return (marks1 + marks2)/2.0;
}
int calcAverage(int marks1, int marks2) {
    return (marks1 + marks2)/2.0;
}
```

Return type of method calcAverage is double

Return type is int

The previous methods can't be termed overloaded methods. If they're defined within the same class, they won't compile. The code also won't compile if one of the methods is defined in a subclass or derived class.

3.4.3 Access modifier

Methods can't be defined as overloaded methods if they only differ in their access modifiers:

```
public double calcAverage(int marks1, int marks2) {
    return (marks1 + marks2)/2.0;
}
private double calcAverage(int marks1, int marks2) {
    return (marks1 + marks2)/2.0;
}
```

Access—public

Access—private

If you define overloaded `calcAverage` methods as shown in the preceding code, the code won't compile.

How to remember the rules of defining overloaded methods

An interesting way to remember the preceding rules for overloaded methods is illustrated in figure 3.16 with faces that have varying shapes. These faces also differ in the shapes of their noses and lips, and the number and shapes of their hairs. These faces are like methods and their parts are like the different parts of a method:

- Shape of face (triangle, oval, square) = method name
- Shape of nose = method's return type
- Shape of lips = method's access modifier
- Hair = method parameters

Your task is to encircle all the methods (faces) that you think overload the method (face) in the center. To get you started, I have circled one of the faces.

In the next section, you'll create special methods called constructors, which are used to create objects of a class.

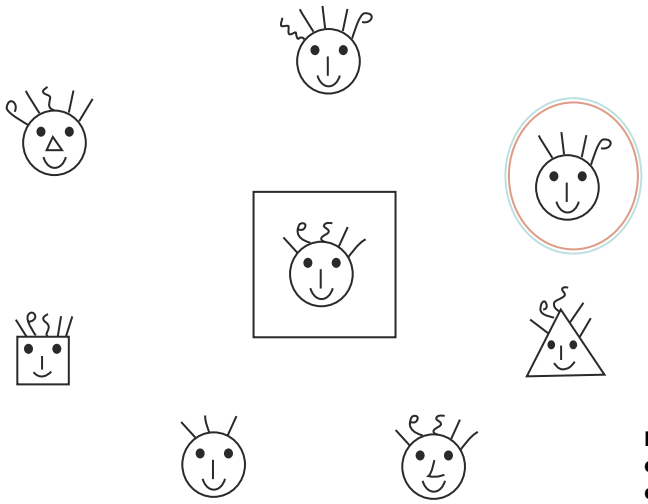


Figure 3.16 An interesting exercise to remember method-overloading rules

3.5 Constructors of a class

[6.4] Differentiate between default and user-defined constructors

[6.5] Create and overload constructors

In this section, you'll create constructors, learn the differences between default and user-defined constructors, and create overloaded constructors.

What happens when you open a new bank account? Depending on the services your bank provides, you may be assigned a new bank account number, provided with a checkbook, and given access to a new online account the bank has created for you. These details are created and returned to you as part of setting up your new bank account.

Compare these steps with what a constructor does in Java, as illustrated in figure 3.17.

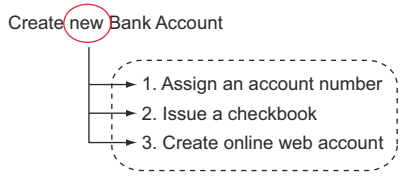


Figure 3.17 The series of steps that may be executed when you create a new bank account. These steps can be compared with what a constructor does in Java.

Constructors are special methods that create and return an object of the class in which they're defined. Constructors have the same name as the name of the class in which they're defined, and they don't specify a return type—not even `void`.

A constructor can accomplish the following tasks:

- Call the base class's constructor; this can be an implicit or explicit call.
- Initialize all of the instance variables of a class with their default values.

Constructors come in two flavors: user-defined constructors and default constructors, which we'll cover in detail in the next sections.

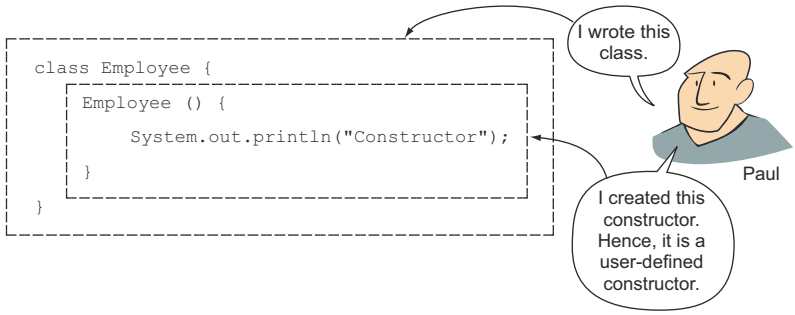


Figure 3.18 A class, `Employee`, with a constructor defined by the user Paul

3.5.1 User-defined constructors

The author of a class has full control over the definition of the class. An author may or may not define a constructor in a class. If the author does define a constructor in a class, it's known as a *user-defined constructor*. Here the word "user" doesn't refer to another person or class that uses this class, but instead refers to the person who created the class. It's called "user-defined" because it's not created by the Java compiler.

Figure 3.18 shows a class `Employee` that defines a constructor.

Here is a class, `Office`, which creates an object of class `Employee`:

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

❶ **Constructor is called on object creation**

In the previous example, ❶ creates an object of class `Employee` using the keyword `new`, which triggers the execution of the `Employee` class constructor. The output of the class `Office` is as follows:

```
Constructor
```

Because a constructor is called as soon as an object is created, you can use it to assign default values to the instance variable of your class, as follows (modified and additional code is highlighted in bold):

```
class Employee {
    String name;
    int age;
    Employee() {
            age = 20;
            System.out.println("Constructor");
    }
}
```

Instance variable

Initialize age

Let's create an object of class `Employee` in class `Office` and see if there's any difference:

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

```
            Employee emp = new Employee();
            System.out.println(emp.age);
    }
}
```

Access and print the value of variable age

The output of the previous code is as follows:

```
Constructor
20
```

Because a constructor is a method, you can also pass method parameters to it, as follows (changes are highlighted in bold):

```
class Employee {
    String name;
    int age;
    Employee(int newAge, String newName) {
            name = newName;
            age = newAge;
            System.out.println("Constructor");
    }
}
```

You can use this constructor in the class `Office` by passing to it the required method arguments, as follows:

```
class Office {
    public static void main(String args[]) {
            Employee emp = new Employee(30, "Pavni Gupta");
    }
}
```

Revisit the use and declaration of the previously mentioned constructors. Note that a constructor is called when you create an object of a class. A constructor does have an implicit return type, which is the class in which it's defined. It creates and returns an object of its class, which is why you can't define a return type for a constructor. Also note that you can define constructors using any of the four access modifiers.

EXAM TIP You can define a constructor using all four access modifiers: `public`, `protected`, default, and `private`.

What happens if you define a return type for a constructor? Java will treat it as another method, not a constructor, which also implies that it won't be called implicitly when you create an object of its class:

```
class Employee {
    void Employee() {
            System.out.println("Constructor");
    }
}
class Office {
    public static void main(String args[]) {
            Employee emp = new Employee();
    }
}
```

1 Doesn't call method Employee with return type void

In the previous example, ❶ won't call the method `Employee` with the return type `void` defined in the class `Employee`. Because the method `Employee` defines its return type as `void`, it's no longer treated as a constructor.

If the class `Employee` defines the return type of the method `Employee` as `void`, how can Java use it to create an object? The method (with the return type `void`) is reduced to the state of another method in the class `Employee`. This logic applies to all of the other data types: if you define the return type of a constructor to be any data type—such as `char`, `int`, `String`, `long`, `double`, or any other class—it'll no longer be treated as a constructor.

How do you execute such a method? By calling it explicitly, as in the following code (modified code is in bold):

```
class Employee {
    void Employee() {
            System.out.println("not a Constructor now");
    }
}
class Office {
    public static void main(String args[]) {
            Employee emp = new Employee();
            emp.Employee();
    }
}
```

Prints "not a Constructor now"

Note that the previous method is called like any other method defined in class `Employee`. It doesn't get called automatically when you create an object of class `Employee`. As you can see in the previous code, it's perfectly fine to define a method that's not a constructor in a class with the same name. Interesting.

But note that the authors of the OCA exam also found this interesting, and you're likely to get a few tricky questions regarding this concept. Don't worry: with the right information under your belt, you're sure to answer them correctly.

EXAM TIP A constructor must not define any return type. Instead, it creates and returns an object of the class in which it's defined. If you define a return type for a constructor, it'll no longer be treated as a constructor. Instead, it'll be treated as a regular method, even though it shares the same name as its class.

INITIALIZER BLOCKS VERSUS CONSTRUCTORS

An *initializer block* is defined within a class, not as a part of a method. It executes for every object that's created for a class. In the following example, the class `Employee` defines an initializer block:

```
class Employee {
    {
        System.out.println("Employee:initializer");
    }
}
```

Initializer block

In the following code, the class `TestEmp` creates an object of class `Employee`:

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

Prints "Employee:initializer"

If you define both an initializer and a constructor for a class, both of these will execute. The initializer block will execute prior to the constructor:

```
class Employee {
    Employee() {
        System.out.println("Employee:constructor");
    }
    {
        System.out.println("Employee:initializer");
    }
}
class TestEmp {
    public static void main(String args[]) {
        Employee e = new Employee();
    }
}
```

Constructor

Initializer block

Creates an object of class Employee; calls both the initializer block and the constructor

The output of the class `TestEmp` is as follows:

```
Employee:initializer
Employee:constructor
```

Do the previous examples leave you wondering why we need both an initializer block and a constructor, if both of these execute upon the creation of an object? Initializer blocks are used to initialize the variables of anonymous classes. An *anonymous class* is a type of inner class. In the absence of a name, anonymous classes can't define a constructor and rely on an initializer block to initialize their variables upon the creation of an object of their class. Because inner classes are not on this exam, I won't discuss how to use an initializer block with an anonymous inner class.

A lot of action can happen within an initializer block: It can create local variables. It can access and assign values to instance and static variables. It can call methods and define loops, conditional statements, and `try-catch-finally` blocks. Unlike constructors, an initializer block can't accept method parameters.

NOTE Loops and conditional statements are covered in chapter 5, and `try-catch-finally` blocks are covered in chapter 7.

3.5.2 Default constructor

In the previous section on user-defined constructors, I discussed how a constructor is used to create an object. What happens if you don't define any constructor in a class?

The following code is an example of the class `Employee` that doesn't define a constructor:

```
class Employee {
    String name;
    int age;
}
```

No constructor is defined in class Employee

You can create objects of this class in another class (`Office`), as follows:

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

Class Employee doesn't define a constructor, but this code compiles successfully

In this case, which method creates the object of the class `Employee`? Figure 3.19 shows what happens when a class (`Employee`) is compiled that doesn't define any constructor. In the absence of a user-defined constructor, Java inserts a *default constructor*. This constructor doesn't accept any method arguments. It calls the constructor of the super (parent) class and assigns default values to all the instance variables.

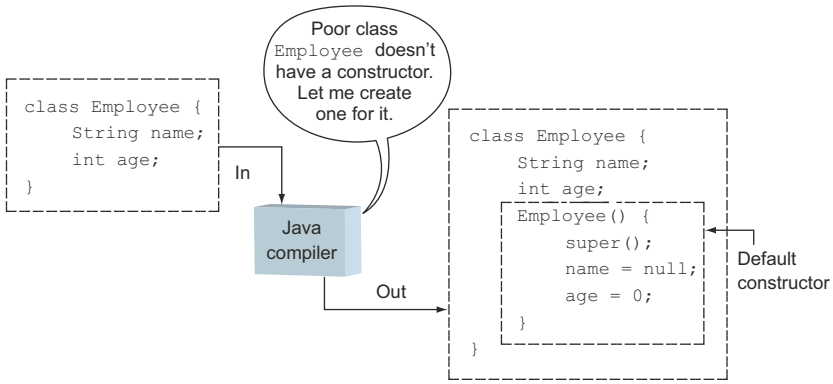


Figure 3.19 When the Java compiler compiles a class that doesn't define a constructor, the compiler creates one for it.

What happens if you add another constructor to the class `Employee`, as in the following example?

```
class Employee {
    String name;
    int age;
    Employee(int newAge, String newName) {
            name = newName;
            age = newAge;
            System.out.println("User defined Constructor");
    }
}
```

User-defined constructor

In this case, upon recompilation, the Java compiler will notice that you've defined a constructor in the class `Employee`. It won't add a default constructor to it, as shown in figure 3.20.

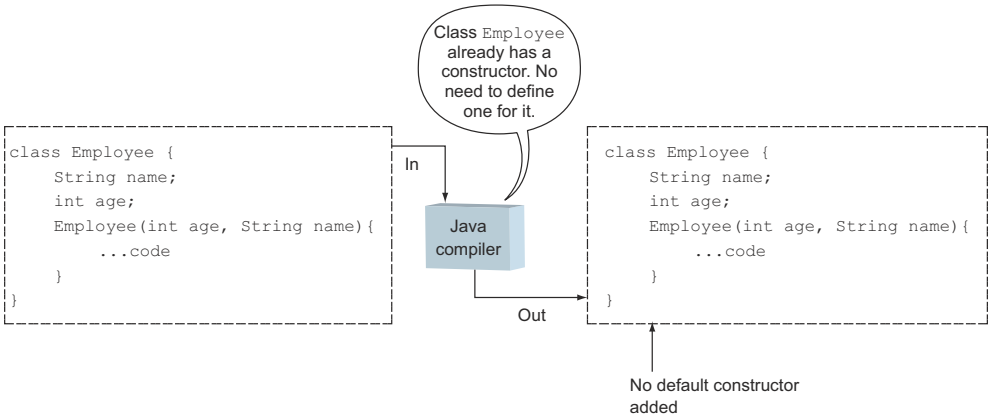


Figure 3.20 When a class with a constructor is compiled, the Java compiler doesn't add a default constructor to it.

In the absence of a no-argument constructor, the following code will fail to compile:

```
class Office {
    public static void main(String args[]) {
            Employee emp = new Employee();        <-- Won't compile
    }
}
```

EXAM TIP Java defines a default constructor if and only if you don't define a constructor. If a class doesn't define a constructor, the compiler will add a default, no-argument constructor to the class. But if you modify the class later by adding a constructor to it, the Java compiler will remove the default, no-argument constructor that it initially added to the class.

3.5.3 *Overloaded constructors*

In the same way in which you can overload methods in a class, you can also overload the constructors in a class. *Overloaded constructors* follow the same rules as discussed in the previous section for overloaded methods. Here's a quick recap:

- Overloaded constructors must be defined using different argument lists.
- Overloaded constructors can't be defined by just a change in the access modifiers.

Because constructors don't define a return type, there's no point to defining invalid overloaded constructors with different return types.

The following is an example of an `Employee` class that defines four overloaded constructors:

```
class Employee {
    String name;
    int age;
    Employee() {                    <-- (1) No-argument constructor
            name = "John";
```

```
        age = 25;
    }
    Employee(String newName) {
        name = newName;
        age = 25;
    }
    Employee(int newAge, String newName) {
        name = newName;
        age = newAge;
    }
    Employee(String newName, int newAge) {
        name = newName;
        age = newAge;
    }
}
```

❷ Constructor with one String argument

❸ Constructor with two arguments—int and String

❹ Constructor with two arguments—String and int

In the previous code, ❶ defines a constructor that doesn't accept any method arguments. ❷ defines another constructor that accepts a single method argument. Note the constructors defined at ❸ and ❹. Both of these accept two method arguments, `String` and `int`. But the placement of these two method arguments is different in ❸ and ❹, which is acceptable and valid for overloaded constructors and methods.

INVOKING AN OVERLOADED CONSTRUCTOR FROM ANOTHER CONSTRUCTOR

It's common to define multiple constructors in a class and reuse their functionality across constructors. Unlike overloaded methods, which can be invoked using the name of a method, overloaded constructors are invoked by using the keyword `this`—an implicit reference that's accessible to all objects that refer to an object itself:

```
class Employee {
    String name;
    int age;
    Employee() {
        this(null, 0);
    }
    Employee(String newName, int newAge) {
        name = newName;
        age = newAge;
    }
}
```

❶ No-argument constructor

❷ Invokes constructor that accepts two method arguments

❸ Constructor that accepts two method arguments

The code at ❶ creates a no-argument constructor. At ❷, this constructor calls the overloaded constructor by passing to it values `null` and `0`. ❸ defines an overloaded constructor that accepts two method arguments.

Because a constructor is defined using the name of its class, it's a common mistake to try to invoke a constructor from another constructor using the class's name:

```
class Employee {
    String name;
    int age;
    Employee() {
        Employee(null, 0);
    }
```

Won't compile—you can't invoke a constructor within a class by using the class's name.

```
    Employee(String newName, int newAge) {
            name = newName;
            age = newAge;
    }
}
```

Also, when you invoke an overloaded constructor using the keyword `this`, it must be the first statement in your constructor:

```
class Employee {
    String name;
    int age;
    Employee() {
            System.out.println("No-argument constructor");
            this(null, 0);
    }
    Employee(String newName, int newAge) {
            name = newName;
            age = newAge;
    }
}
```

Won't compile—the call to the overloaded constructor must be the first statement in a constructor.

That's not all: you can't call a constructor from any other method in your class. None of the other methods of the class `Employee` can invoke its constructor.

RULES TO REMEMBER

Here's a quick list of rules to remember for the exam for defining and using overloaded constructors:

- Overloaded constructors must be defined using different argument lists.
- Overloaded constructors can't be defined by just a change in the access modifiers.
- Overloaded constructors may be defined using different access modifiers.
- A constructor can call another overloaded constructor by using the keyword `this`.
- A constructor can't invoke a constructor by using its class's name.
- If present, the call to another constructor must be the first statement in a constructor.

The next Twist in the Tale exercise hides an important concept within its code, which you can get to know only if you execute the modified code (answer in the appendix).

Twist in the Tale 3.2

Let's modify the definition of the class `Employee` that I used in the section on overloaded constructors, as follows:

```
class Employee {
    String name;
    int age;
    Employee() {
            this ();
    }
    Employee (String newName, int newAge) {
```

```
            name = newName;
            age = newAge;
    }
}
```

What is the output of this modified code, and why?

Now that you've seen how to create methods, constructors, and their overloaded variants, we'll turn to how all of these can be used to access and modify object fields in the next section.

3.6 Accessing object fields

[2.3] Read or write to object fields

[2.5] Call methods on objects

In this section, you'll read, initialize, and modify object fields. You'll also learn the correct notation used to call methods on objects. Access modifiers also determine whether you can call a method on an object.

3.6.1 What is an object field?

An *object field* is another name for an instance variable defined in a class. I've often seen certification aspirants who are confused over whether the object fields are the same as instance variables of a class.

Here's an example of the class `Star`:

```
class Star {
    double starAge;
    public void setAge(double newAge) {
            starAge = newAge;
    }
    public double getAge() {
            return starAge;
    }
}
```

❶ Instance variable—starAge

❷ Setter method—setAge

❸ Getter method—getAge

In the previous example, ❶ defines an instance variable, `starAge`. ❷ defines a *setter* method, `setAge`. A *setter* method is used to set the value of a variable. ❸ defines a *getter* method, `getAge`. A *getter* method is used to retrieve the value of a variable. In this example, the object field is `starAge`, not age or `newAge`. The name of an object field is not determined by the name of its *getter* or *setter* methods.

3.6.2 Read and write object fields

The OCA Java SE 7 Programmer I exam will test you on how to read values from and write them to fields of an object, which can be accomplished by any of following:

JavaBeans properties and object fields

The reason for the confusion over the name of the object field is that Java classes can also be used to define visual components called *JavaBeans*, which are used in visual environments. These classes are supposed to define getter and setter methods to retrieve and set the properties of the visual components. If a visual JavaBean component defines a property such as `age`, then the name of its getter and setter methods would be `getAge` and `setAge`. For a JavaBean, you don't have to worry about the name of the variable that's used to store the value of this property. In a JavaBean, an object field `thisIsMyAge` can be used to store the value of its *property* `age`.

Note that the JavaBeans I mentioned aren't Enterprise JavaBeans. Enterprise JavaBeans are used in enterprise applications written in Java, which run on servers.

- Using methods to read and write object fields
- Using constructors to write values to object fields
- Directly accessing instance variables to read and write object fields

This exam objective (2.3) will also test your understanding of how to assign different values to the same object fields for multiple objects. Let's start with an example:

```
class Employee {
    String name;
    int age;
    Employee() {
            age = 22;
}
    public void setName(String val) {
            name = val;
    }
    public void printEmp() {
            System.out.println("name = " + name + " age = " + age);
    }
}
```

❶ Object fields

❷ Assign value to age

❸ Assign val to name

In class `Employee`, ❶ defines two object fields: `name` and `age`. It defines a (no-argument) constructor. And ❷ assigns a value of `22` to its field `age`. This class also defines a method `setName` where ❸ assigns the value passed to it to the object field `name`. The method `printEmp` is used to print the values of object fields `name` and `age`.

The following is the definition of a class, `Office`, which creates two instances, `e1` and `e2`, of the class `Employee` and assigns values to its fields. Let's look at the output of the class `Office`:

```
class Office {
    public static void main(String args[]) {
            Employee e1 = new Employee();
            Employee e2 = new Employee();
            e1.name = "Selvan";
            e2.setName("Harry");
```

```
            e1.printEmp();
            e2.printEmp();
    }
}
```

This is the output of the previous code:

```
name = Selvan age = 22
name = Harry age = 22
```

Figure 3.21 defines object diagrams (a diagram with the name and type of an object, the name of the object's fields, and their corresponding values), which will help you to better understand the previous output.

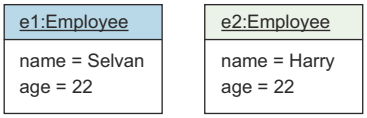


Figure 3.21 Two objects of class `Employee`

You can access the object field name of the object of class `Employee` either by using its variable `name` or by using the method `setName`. The following line of code assigns a value `Selvan` to the field `name` of object `e1`:

```
e1.name = "Selvan";
```

The following line of code uses the method `setName` to assign a value of `Harry` to the field `name` of object `e2`:

```
e2.setName("Harry");
```

Because the constructor of the class `Employee` assigns a value of `22` to the variable `age`, objects `e1` and `e2` both contain the same value, `22`.

What happens if you don't assign any value to an object field and try to print out its value? All the instance variables (object fields) are assigned their default values if you try to access or read their values before writing any values to them:

```
class Employee {
    String name;
    int age;
    public void printEmp() {
            System.out.println("name = " + name + " age = " + age);
    }
}
class Office {
    public static void main(String args[]) {
            Employee e1 = new Employee();
            e1.printEmp();
    }
}
```

Object field: name

Object field: age

The output of the previous code is as follows (the default value of an object is `null` and `int` is `0`):

```
name = null age = 0
```

What happens if you change the access modifier of the variable `name` to `private`, as shown here (modified code in bold)?

```
class Employee {
    private String name;
    int age;
    Employee() {
             age = 22;
    }
    public void setName(String val) {
            name = val;
    }
    public void printEmp() {
            System.out.println("name = " + name + " age = " + age);
    }
}
```

Object field with private access

Nonprivate object field

Assign value to age

Assign val to name

You won't be able to set the value of the object field name as follows:

```
e1.name = "Selvan";
```

This line of code won't compile. Instead, it complains that the variable `name` has private access in the class `Employee` and can't be accessed from any other class:

```
Office.java:6:  name has private access in Employee
            e1.name = "Selvan";
```

When you answer questions on reading values from and writing them to an object field, watch out for the following points, which will help you escape traps laid by the authors of the OCA Java SE 7 Programmer I exam:

- Access modifier of the object field
- Access modifiers of methods used to read and write value of the object field
- Constructors that assign values to object fields

3.6.3 Calling methods on objects

You can call methods defined in a class using an object reference variable. In this exam objective, the OCA Java SE 7 Programmer I exam will specifically test you on the following:

- The correct notation used to call a method on an object reference variable
- The right number of method parameters that must be passed to a method
- The return value of a method that's assigned to a variable

Java uses the dot notation (`.`) to execute a method on a reference variable. Suppose the class `Employee` is defined as follows:

```
class Employee {
    private String name;
    public void setName(String val) {
            name = val;
    }
}
```

Class Employee

Method setName

You can create an object of class `Employee` and call the method `setName` on it like this:

```
Employee e1 = new Employee();
e1.setName("Java");
```

The following method invocations aren't valid in Java:

```
e1->setName("Java");
e1->.setName("Java");
e1-setName("Java");
```

Invalid method invocations

When you call a method, you must pass to it the exact number of method parameters that are defined by it. In the previous definition of the `Employee` class, the method `setName` defines a method parameter of type `String`. You can pass a literal value or a variable to a method, as a method parameter. The following code invocations are correct:

```
Employee e1 = new Employee();
String anotherVal = "Harry";
e1.setName("Shreya");
e1.setName(anotherVal);
```

Passing literal value as method parameter

Passing variable as method parameter

If the parameter list of the called method defines a variable argument at the rightmost position, you can call the method with a variable number of arguments. Let's add a method `daysOffWork` in the class `Employee` that accepts a variable list of arguments (modifications in bold):

```
class Employee {
    private String name;
    public void setName(String val) {
            name = val;
    }
    public int daysOffWork(int... days) {
          int daysOff = 0;
          for (int i = 0; i < days.length; i++)
              daysOff += days[i];
          return daysOff;
    }
```

You can call this method using a variable list of arguments:

```
Class Test {
    public static void main(String args[]) {
        Employee e = new Employee();
        System.out.println(e.daysOffWork(1, 2, 3, 4));
        System.out.println(e.daysOffWork(1, 2, 3));
    }
}
```

Call method daysOffWork with four method arguments

Call method daysOffWork with three method arguments

The output of the previous code is as follows:

```
10
6
```

Let's add a method `getName` to the class `Employee` that returns a `String` value (changes in bold):

```
class Employee {
    private String name;
    public void setName(String val) {
            name = val;
    }
```

```
    public String getName() {
            return name;
    }
}
```

You can assign the `String` value returned from the method `getName` to a `String` variable or pass it on to another method, as follows:

```
Employee e1 = new Employee();
Employee e2 = new Employee();
String name = e1.getName();
e2.setName(e1.getName());
```

Assign method's return value to a variable

Pass the method's return value to another method

In the previous code, the return type of method `setName` is `void`; therefore, you can't use it to assign a value to a variable:

```
Employee e1 = new Employee();
String name = e1.setName();
```

Won't compile

Also, you can't assign a return value of a method to an incompatible variable, as follows:

```
Employee e1 = new Employee();
int val = e1.getName();
```

You can't assign the String returned from method getName to an int variable

You can read and write object fields either by using methods or by directly accessing the instance variables of a class. But it's not a good idea to enable access to the instance variables outside a class.

In the next section, you'll see the risks of exposing instance variables outside a class and the benefits of a well-encapsulated class.

3.7 Apply encapsulation principles to a class

[6.7] Apply encapsulation principles to a class

As the heading of this section suggests, we'll apply the encapsulation principle to a class. A well-encapsulated object doesn't expose its internal parts to the outside world. It defines a set of methods that enable the users of the class to interact with it.

As an example from the real world, you can compare a bank to a well-encapsulated class. A bank doesn't expose its internal parts—for example, its vaults and bank accounts—to the outside world, just as a well-encapsulated class in Java shouldn't expose the variables that it uses to store the state of an object outside of that object. The way a bank defines a set of procedures (such as key access to vaults and verification before money withdrawals) to protect its internal parts is much like the way a well-encapsulated class defines methods to access its variables.

3.7.1 Need for encapsulation

The private members of a class—its variables and methods—are used to hide information about a class. Why would you need to hide information in a class? Compare a class with yourself. Do you want anyone else to know about all of your weaknesses? Do you

want anyone else to be able to control your mind? The same applies to a class that you define in Java. A class may need a number of variables and methods to store an object's state and define its behavior. But it wouldn't like all the other classes to know about it.

Let's work with an example to help you get the hang of this concept. Here's the definition of a class `Phone`:

```
class Phone {
    String model;
    String company;
    double weight;
    void makeCall(String number) {

    }
    void receiveCall() {

    }
}
```

Instance variables that store the state of an object of Phone

Methods; details not relevant at this point

Because the variable `weight` isn't defined as a `private` member, any other class can access it and write any value to it, as follows:

```
class Home {
    public static void main() {
            Phone ph = new Phone();
            ph.weight = -12.23;
    }
}
```

Assign a negative weight to Phone

3.7.2 Apply encapsulation

In the previous section, you might have noticed that the object fields of a class that isn't well encapsulated are exposed outside of the class. This approach enables the users of the class to assign arbitrary values to the object fields.

Should this be allowed? For example, going back to the example of the `Phone` class discussed in the previous section (3.7.1), how can the weight of a phone be a negative value?

Let's resolve this issue by defining the variable `weight` as a `private` variable in class `Phone`, as follows (irrelevant changes have been omitted):

```
class Phone {
    private double weight;

}
```

But now this variable won't be accessible in class `Home`. Let's define methods using this variable, which can be accessible outside the class `Phone` (changes in bold):

```
class Phone {
    private double weight;
    public void setWeight(double val) {
            if (val > 0 && val < 1000) {
                weight =val;
            }
    }
```

Negative and weight over 1,000 not allowed

```
    public double getWeight() {
            return weight;
    }
}
```

The method `setWeight` doesn't assign the value passed to it as a method parameter to the instance variable `weight` if it's a negative value or a value greater than 1,000. This behavior is known as exposing object functionality using public methods.

Let's see how this method is used to assign a value to the variable `weight` in the class `Home`:

```
class Home {
    public static void main(String[] args) {
            Phone ph = new Phone();
            ph.setWeight(-12.23);
Prints 0.0 ⟶ System.out.println(ph.getWeight());

            ph.setWeight(77712.23);
Prints 0.0 ⟶ System.out.println(ph.getWeight());

            ph.setWeight(12.23);
            System.out.println(ph.getWeight());
    }
}
```

Assign a negative weight to Phone object

Assign weight > 1,000 to Phone object

Assign weight in allowed range

Prints 12.23

Note that when the class `Home` tries to set the value of the variable to `-12.23` or `77712.23` (out-of-range values), those values aren't assigned to the `Phone`'s private variable `weight`. It accepts the value `12.23`, which is within the defined range.

On the OCA Java SE 7 Programmer I exam, you may also find the term "information hiding." *Encapsulation* is the concept of defining variables and the methods together in a class. *Information hiding* originated from the application and purpose of the concept of encapsulation. These terms are also used interchangeably.

EXAM TIP The terms *encapsulation* and *information hiding* are used interchangeably. By exposing object functionality only through methods, you can prevent your private variables from being assigned any values that don't fit your requirements. One of the best ways to create a well-encapsulated class is to define its instance variables as private variables and allow access to these variables using public methods.

The next Twist in the Tale exercise has a little hidden trick about determining a correctly encapsulated class. Let's see if you can find it (answer in the appendix).

Twist in the Tale 3.3

Let's modify the definition of the class `Phone` that I previously used to demonstrate the encapsulation principle in this section. Given the following definition of class `Phone`, which of the options, when replacing the code on lines 1–3, makes it a well-encapsulated class?

```
class Phone {
    public String model;
    double weight;                                      //LINE1
    public void setWeight(double w) {weight = w;}      //LINE2
    public double getWeight() {return weight;}         //LINE3
}
```

a
```
public double weight;
private void setWeight(double w) { weight = w; }
private double getWeight() {    return weight; }
```

b
```
public double weight;
void setWeight(double w) { weight = w; }
double getWeight() {    return weight; }
```

c
```
public double weight;
protected void setWeight(double w) { weight = w; }
protected double getWeight() {    return weight; }
```

d
```
public double weight;
public void setWeight(double w) { weight = w; }
public double getWeight() {    return weight; }
```

e None of the above.

Well-encapsulated classes don't expose their instance variables outside their class. What happens when the methods of these classes modify the state of the method arguments that are passed to them? Is this acceptable behavior? I'll discuss what happens in the next section.

3.8 Passing objects and primitives to methods

[6.8] Determine the effect upon object references and primitive values when they are passed into methods that change the values

In this section, you'll learn the difference between passing object references and primitives to a method. You'll determine the effect upon object references and primitive values when they're passed into methods that change the values.

Object references and primitives behave in a different manner when they're passed to a method because of the differences in how these two data types are internally stored by Java. Let's start with passing primitives to methods.

3.8.1 Passing primitives to methods

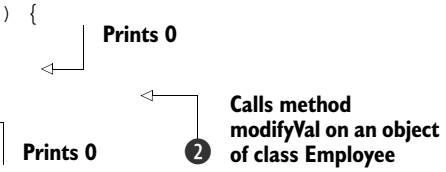
The value of a primitive data type is copied and passed on to a method. Hence, the variable whose value was copied doesn't change:

```
class Employee {
    int age;
    void modifyVal(int a) {
        a = a + 1;
        System.out.println(a);
```

❶ **Method modifyVal accepts method argument of type int**

```
    }
}

class Office {
    public static void main(String args[]) {
        Employee e = new Employee();
        System.out.println(e.age);
        e.modifyVal(e.age);
        System.out.println(e.age);
    }
}
```



The output of the previous code is as follows:

```
0
1
0
```

The method `modifyVal` ❶ accepts a method argument `a` of type `int`. In this method, the variable `a` is a method parameter and holds a copy of the value that is passed to it. The method increments the value of the method parameter `a` and prints its value.

When the class `Office` calls the method `modifyVal` ❷, it passes a copy of the value of the object field `age` to it. The method `modifyVal` never accesses the object field `age`. Hence, after the execution of this method, the value of the method field `age` prints as `0` again.

What happens if the definition of the class `Employee` is modified as follows (modifications in bold):

```
class Employee {
    int age;
    void modifyVal(int age) {
            age = age + 1;
            System.out.println(age);
    }
}
```

The class `Office` will still print the same answer because the method `modifyVal` defines a method parameter with the name `age`. Note the following important points related to passing a method parameter to a method:

- It's okay to define a method parameter with the same name as an instance variable (or object field).
- Within a method, a method parameter takes precedence over an object field. When the method `modifyVal` refers to the variable `age`, it refers to the method parameter `age`, not the instance variable `age`. To access the instance variable `age` within the method `modifyVal`, the variable name `age` needs to be prefixed with the keyword `this` (`this` is a keyword that refers to the object itself).

The keyword `this` is discussed in detail in chapter 6.

EXAM TIP When you pass a primitive variable to a method, its value remains the same after the execution of the method. The value doesn't change, regardless of whether the method reassigns the primitive to another variable or modifies it.

3.8.2 Passing object references to methods

There are two main cases:

- When a method reassigns the object reference passed to it to another variable
- When a method modifies the state of the object reference passed to it

WHEN METHODS REASSIGN THE OBJECT REFERENCES PASSED TO THEM

When you pass an object reference to a method, the method can assign it to another variable. In this case, the state of the object, which was passed on to the method, remains intact.

The following code example explains this concept. Suppose you have the following definition of class `Person`:

```
class Person {
    private String name;
    Person(String newName) {
            name = newName;
    }
    public String getName() {
            return name;
    }
    public void setName(String val) {
            name = val;
    }
}9
```

What do you think is the output of the following code?

```
class Test {
    public static void swap(Person p1, Person p2) {
            Person temp = p1;
            p1 = p2;
            p2 = temp;
    }
    public static void main(String args[]) {
            Person person1 = new Person("John");
            Person person2 = new Person("Paul");
            System.out.println(person1.getName()
                                + ":" + person2.getName());

            swap(person1, person2);

            System.out.println(person1.getName()
                                + ":" + person2.getName());
    }
}
```

Method to swap two object references

1 Creates object

2 Prints John:Paul before passing objects referred by variable personl and person2 to method swap

Executes method swap

3 Prints John:Paul after method swap completes execution

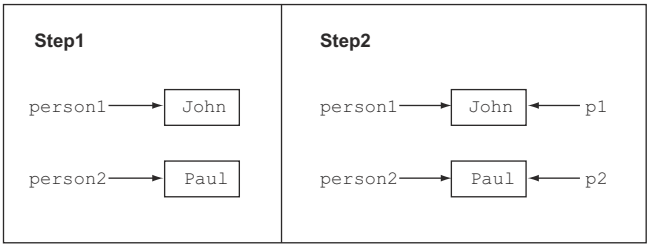


Figure 3.22 Objects of class `Person`, referred to by variables `person1`, `person2`, `p1`, and `p2`

In the previous code, ❶ creates two object references, `person1` and `person2`, illustrated in step 1 of figure 3.22. The boxed values represent objects of class `Person`. ❷ prints `John:Paul`—the value of `person1.name` and `person2.name`.

The code then calls the method `swap` and passes to it the objects referred to by `person1` and `person2`. When these objects are passed as arguments to the method `swap`, the method arguments `p1` and `p2` also refer to these objects. This behavior is illustrated in step 2 in figure 3.22.

The method `swap` defines three lines of code:

- `Person temp = p1;` makes `temp` refer to the object referred to by `p1`.
- `p1 = p2;` makes `p1` refer to the object referred to by `p2`.
- `p2 = temp;` makes `p2` refer to the object referred to by `temp`.

These three steps are represented in figure 3.23.

As you can see in figure 3.23, the reference variables `person1` and `person2` are still referring to the objects that they passed to the method `swap`. Because no change was made to the values of the objects referred to by variables `person1` and `person2`, ❸ prints `John:Paul` again.

The output of the previous code is:

```
John:Paul
John:Paul
```

WHEN METHODS MODIFY THE STATE OF THE OBJECT REFERENCES PASSED TO THEM

Let's see how a method can change the state of an object so that the modified state is accessible in the calling method. Assume the same definition of the class `Person`, listed again for your convenience:

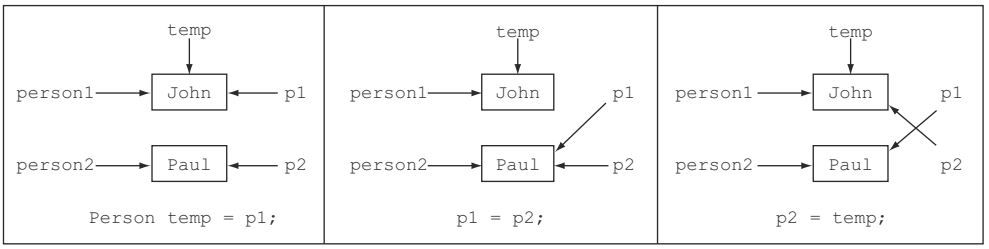


Figure 3.23 The change in the objects referred to by variables during the execution of the method `swap`

```
class Person {
    private String name;
    Person(String newName) {
            name = newName;
    }
    public String getName() {
            return name;
    }
    public void setName(String val) {
            name = val;
    }
}
```

What's the output of the following code?

```
class Test {
    public static void resetValueOfMemberVariable(Person p1) {
            p1.setName("Rodrigue");
    }
    public static void main(String args[]) {
            Person person1 = new Person("John");

            System.out.println(person1.getName());

            resetValueOfMemberVariable(person1);

            System.out.println(person1.getName());
    }
}
```

Create an object reference person1

Print person1.name before passing it to resetValueOfMemberVariable

Pass person1 to method resetValueOfMemberVariable

Print person1.name after passing it to resetValueOfMemberVariable

The output of the previous code is as follows:

```
John
Rodrigue
```

The method `resetValueOfMemberVariable` accepts the object referred to by `person1` and assigns it to the method parameter `p1`. Now both the variables `person1` and `p1` refer to the same object. `p1.setName("Rodrigue")` modifies the value of the object referred to by variable `p1`. Because the variable `person1` also refers to the same object, `person1.getName()` returns the new name, `Rodrigue`, in the method `main`. This sequence of actions is represented in figure 3.24.

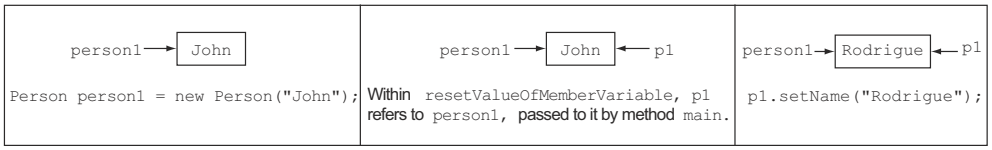


Figure 3.24 Modification of the state of an object passed to the method `resetValueOfMemberVariable`

3.9 Summary

I started this chapter by discussing the scope of these variables: local, method parameter, instance, and class. Often these variables' scopes overlap with each other. I also covered the constructors of a class: the user-defined and default constructors. Java inserts a default constructor in a class that doesn't define any constructor. You can modify the source code of such a class, add a constructor, and recompile the class. Upon recompilation, the Java compiler removes the automatically generated constructor.

I then covered the subobjective of reading from and writing to object fields. The terms *object fields* and *instance variables* have the same meaning and are used interchangeably. You can read from and write to object fields by directly accessing them or by using accessor methods. I also showed you how to apply encapsulation principles to a class and explained why doing so is useful.

Finally, I explained the effect upon references and primitives when they're passed into methods that change their values. When you pass a primitive value to a method, its value never changes for the calling method. When you pass an object reference variable to a method, a change in its value may be reflected in the calling method—if the called method modifies an object field of the object passed to it. If the called method assigns a new object reference to the method argument before modifying the value of its fields, these changes aren't visible in the calling method.

3.10 Review notes

This section lists the main points covered in this chapter.

Scope of variables:

- Variables can have multiple scopes: class, instance, local, and method parameters.
- Local variables are defined within a method. Loop variables are local to the loop within which they're defined.
- The scope of local variables is less than the scope of a method if they're declared in a sub-block (within braces, {}) in a method. This sub-block can be an `if` statement, a `switch` construct, a loop, or a `try-catch` block (discussed in chapter 7).
- Local variables can't be accessed outside the method in which they're defined.
- Instance variables are defined and accessible within an object. They're accessible to all the instance methods of a class.
- Class variables are shared by all of the objects of a class—they can be accessed even if there are no objects of the class.
- Method parameters are used to accept arguments in a method. Their scope is limited to the method where they're defined.
- A method parameter and a local variable can't be defined using the same name.
- Class and instance variables can't be defined using the same name.

- Local and instance variables can be defined using the same name. In a method, if a local variable exists with the same name as an instance variable, the local variable takes precedence.

Object's lifecycle:

- An object's lifecycle starts when it's initialized and lasts until it goes out of scope or is no longer referenced by a variable.
- When an object is alive, it can be referenced by a variable and other classes can use it by calling its methods and accessing its variables.
- Declaring a reference object variable isn't the same as creating an object.
- An object is created using the operator `new`. Strings have special shorthand built into the compiler. Strings can be created by using double quotes, as in `"Hello"`.
- An object is marked as eligible for garbage collection when it can no longer be accessed.
- An object can become inaccessible if it can no longer be referenced by any variable, which happens when a reference variable is explicitly set to `null` or when it goes out of scope.
- You can be sure only about whether objects are marked for garbage collection. You can never be sure about whether an object has been garbage collected.

Create methods with arguments and return values:

- The return type of a method states the type of value that a method will return.
- You can define multiple method parameters for a method.
- The method parameter can be of a primitive type or objects of a class or interface.
- The method parameters are separated by commas.
- Each method parameter is preceded by the name of its type. You can't define the type of a method once, even when they're of the same type (the way you can when declaring multiple variables of same type).
- You can define only one variable argument in a parameter list, and it should be the final variable in the parameter list. If these two rules aren't followed, your code won't compile.
- For a method that returns a value, the `return` statement must be followed immediately by a value.
- For a method that doesn't return a value (return type is `void`), the `return` statement must not be followed by a return value.
- If there is code that can be executed only after a `return` statement, the class will fail to compile.
- A method can optionally accept method arguments.
- A method may optionally return a value.

- A method returns a value by using the keyword `return` followed by the name of a variable, whose value is passed back to the calling method.
- The returned value from a method may or may not be assigned to a variable. If the value is assigned to a variable, the variable type should be compatible with the type of the return value.
- A `return` statement should be the last statement in a method. Statements placed after the `return` statement aren't accessible and fail to compile.

Create an overloaded method:

- Overloaded methods accept different lists of arguments. The argument lists can differ by
  - Changes in the number of parameters that are accepted
  - Changes in the types of parameters that are accepted
  - Changes in the positions of parameters that are accepted
- Methods can't be defined as overloaded methods if they differ only in their return types or access modifiers.

Constructors of a class:

- Constructors are special methods defined in a class that create and return an object of the class in which they're defined.
- Constructors have the same name as the class, and they don't specify a return type—not even `void`.
- User-defined constructors are defined by the developer.
- Default constructors are defined by Java, but only if the developer doesn't define any constructor in a class.
- You can define a constructor using the four access modifiers: `public`, `protected`, default, and `private`.
- If you define a return type for a constructor, it'll no longer be treated like a constructor. It'll be treated like a regular method, even though it shares the same name as its class.
- An *initializer block* is defined within a class, not as a part of a method. It executes for every object that's created for a class.
- If you define both an initializer and a constructor for a class, both of these will execute. The initializer block will execute prior to the constructor.
- Unlike constructors, an initializer block can't accept method parameters.
- An initializer block can create local variables. It can access and assign values to instance and static variables. It can call methods and define loops, conditional statements, and `try-catch-finally` blocks.

Overloaded constructors:

- A class can also define overloaded constructors.
- Overloaded constructors should be defined using different argument lists.

- Overloaded constructors can't be defined by just a change in the access modifiers.
- Overloaded constructors may be defined using different access modifiers.
- A constructor can call another overloaded constructor by using the keyword `this`.
- A constructor can't invoke a constructor by using its class's name.
- If present, a call to another constructor should be the first statement in a constructor.

Accessing object fields:

- An object field is another name for an instance variable defined in a class.
- An object field can be read by either directly accessing the variable (if its access modifier permits) or by using a method that returns its value.
- An object field can be written by either directly accessing the variable (if its access modifier permits) or by using constructors and methods that accept a value and assign it to the instance variable.
- You can call methods defined in a class using an object reference variable.
- When calling a method, it must be passed the correct number and type of method arguments.

Applying encapsulation principles to a class:

- A well-encapsulated object doesn't expose the internal parts of an object outside it. It defines a set of well-defined interfaces (methods), which enables the users of the class to interact with it.
- A class that isn't well encapsulated is at risk of being assigned undesired values for its variables by the callers of the class, which can make the state of an object unstable.
- By exposing object functionality only through methods, you can prevent private variables from being assigned values that don't fit your requirements.
- The terms *encapsulation* and *information hiding* are also used interchangeably.
- One of the best ways to define a well-encapsulated class is to define its instance variables as private variables and allow access to these variables using methods.

Passing objects and primitives to methods:

- Objects and primitives behave in different manners when they're passed to a method, because of differences in the way these two data types are internally stored by Java.
- When you pass a primitive variable to a method, its value remains the same after the execution of the method. This doesn't change, regardless of whether the method reassigns the primitive to another variable or modifies it.
- When you pass an object to a method, the method can modify the object's state by executing its methods. In this case, the modified state of the object is reflected in the calling method.

3.11 Sample exam questions

Q3-1. How can you include encapsulation in your class design?

- **a** Define instance variables as private members.
- **b** Define public methods to access and modify the instance variables.
- **c** Define some of the instance variables as public members.
- **d** All of the above.

Q3-2. Examine the following code and select the correct option(s):

```
public class Person {
    public int height;
    public void setHeight(int newHeight) {
            if (newHeight <= 300)
                height = newHeight;
    }
}
```

- **a** The `height` of a `Person` can never be set to more than 300.
- **b** The previous code is an example of a well-encapsulated class.
- **c** The class would be better encapsulated if the height validation weren't set to 300.
- **d** Even though the class isn't well encapsulated, it can be inherited by other classes.

Q3-3. Which of the following methods correctly accepts three whole numbers as method arguments and returns their sum as a decimal number?

- **a**
```
public void addNumbers(byte arg1, int arg2, int arg3) {
    double sum = arg1 + arg2 + arg3;
}
```
- **b**
```
public double subtractNumbers(byte arg1, int arg2, int arg3) {
    double sum = arg1 + arg2 + arg3;
    return sum;
}
```
- **c**
```
public double numbers(long arg1, byte arg2, double arg3) {
    return arg1 + arg2 + arg3;
}
```
- **d**
```
public float wakaWakaAfrica(long a1, long a2, short a977) {
    double sum = a1 + a2 + a977;
    return (float)sum;
}
```

Q3-4. Which of the following statements are true?

- **a** If the return type of a method is `int`, the method can return a value of type `byte`.
- **b** A method may or may not return a value.
- **c** If the return type of a method is `void`, it can define a `return` statement without a value, as follows:

```
return;
```

d A method may or may not accept any method arguments.

e A method must accept at least one method argument or define its return type.

f A method whose return type is `String` can't return `null`.

Q3-5. Given the following definition of class `Person`,

```
class Person {
    public String name;
    public int height;
}
```

what is the output of the following code?

```
class EJavaGuruPassObjects1 {
    public static void main(String args[]) {
            Person p = new Person();
            p.name = "EJava";

            anotherMethod(p);
            System.out.println(p.name);

            someMethod(p);
            System.out.println(p.name);
    }
    static void someMethod(Person p) {
            p.name = "someMethod";
            System.out.println(p.name);
    }
    static void anotherMethod(Person p) {
            p = new Person();
            p.name = "anotherMethod";
            System.out.println(p.name);
    }
}
```

a
```
anotherMethod
anotherMethod
someMethod
someMethod
```

b
```
anotherMethod
EJava
someMethod
someMethod
```

c
```
anotherMethod
EJava
someMethod
EJava
```

d Compilation error.

Q3-6. What is the output of the following code?

```
class EJavaGuruPassPrim {
    public static void main(String args[]) {
            int ejg = 10;
            anotherMethod(ejg);
            System.out.println(ejg);
```

```
            someMethod(ejg);
            System.out.println(ejg);
    }
    static void someMethod(int val) {
            ++val;
            System.out.println(val);
    }
    static void anotherMethod(int val) {
            val = 20;
            System.out.println(val);
    }
}
```

a
```
20
10
11
11
```

b
```
20
20
11
10
```

c
```
20
10
11
10
```

d Compilation error

Q3-7. Given the following signature of method `eJava`, choose the options that correctly overload this method:

```
public String eJava(int age, String name, double duration)
```

a `private String eJava(int val, String firstName, double dur)`

b `public void eJava(int val1, String val2, double val3)`

c `String eJava(String name, int age, double duration)`

d `float eJava(double name, String age, byte duration)`

e `ArrayList<String> eJava()`

f `char[] eJava(double numbers)`

g `String eJava()`

Q3-8. Given the following code,

```
class Course {
    void enroll(long duration) {
            System.out.println("long");
    }
    void enroll(int duration) {
            System.out.println("int");
    }
    void enroll(String s) {
            System.out.println("String");
    }
```

```
    void enroll(Object o) {
            System.out.println("Object");
    }
}
```

what is the output of the following code?

```
class EJavaGuru {
    public static void main(String args[]) {
            Course course = new Course();
            char c = 10;
            course.enroll(c);
            course.enroll("Object");
    }
}
```

a Compilation error

b Runtime exception

c
```
int
String
```

d
```
long
Object
```

Q3-9. Examine the following code and select the correct options:

```
class EJava {
    public EJava() {
            this(7);
            System.out.println("public");
    }
    private EJava(int val) {
            this("Sunday");
            System.out.println("private");
    }
    protected EJava(String val) {
            System.out.println("protected");
    }
}

class TestEJava {
    public static void main(String[] args) {
            EJava eJava = new EJava();
    }
}
```

a The class `EJava` defines three overloaded constructors.

b The class `EJava` defines two overloaded constructors. The private constructor isn't counted as an overloaded constructor.

c Constructors with different access modifiers can't call each other.

d The code prints the following:

```
protected
private
public
```

e The code prints the following:

```
public
private
protected
```

Q3-10. Select the incorrect options:

a If a user defines a `private` constructor for a `public` class, Java creates a `public` default constructor for the class.
b A class that gets a default constructor doesn't have overloaded constructors.
c A user can overload the default constructor of a class.
d The following class is eligible for a default constructor:

```
class EJava {}
```

e The following class is also eligible for a default constructor:

```
class EJava {
        void EJava() {}
}
```

3.12 Answers to sample exam questions

Q3-1. How can you include encapsulation in your class design?

a **Define instance variables as private members.**
b **Define public methods to access and modify the instance variables.**
c Define some of the instance variables as public members.
d All of the previous.

Answer: a, b

Explanation: A well-encapsulated class should be like a capsule, hiding its instance variables from the outside world. The only way you should access and modify instance variables is through the public methods of a class to ensure that the outside world can access only the variables the class allows it to. By defining methods to assign values to its instance variables, a class can control the range of values that can be assigned to them.

Q3-2. Examine the following code and select the correct option(s):

```
public class Person {
    public int height;
    public void setHeight(int newHeight) {
            if (newHeight <= 300)
                height = newHeight;
    }
}
```

a The `height` of a `Person` can never be set to more than 300.
b The previous code is an example of a well-encapsulated class.

c The class would be better encapsulated if the height validation weren't set to 300.

d **Even though the class isn't well encapsulated, it can be inherited by other classes.**

Answer: d

Explanation: This class isn't well encapsulated because its instance variable `height` is defined as a `public` member. Because the instance variable can be directly accessed by other classes, the variable doesn't always use the method `setHeight` to set its `height`. The class `Person` can't control the values that can be assigned to its public variable `height`.

Q3-3. Which of the following methods correctly accepts three whole numbers as method arguments and returns their sum as a decimal number?

a
```
public void addNumbers(byte arg1, int arg2, int arg3) {
    double sum = arg1 + arg2 + arg3;
}
```

b
```
public double subtractNumbers(byte arg1, int arg2, int arg3) {
    double sum = arg1 + arg2 + arg3;
    return sum;
}
```

c
```
public double numbers(long arg1, byte arg2, double arg3) {
    return arg1 + arg2 + arg3;
}
```

d
```
public float wakaWakaAfrica(long a1, long a2, short a977) {
    double sum = a1 + a2 + a977;
    return (float)sum;
}
```

Answer: b, d

Explanation: Option (a) is incorrect. The question specifies the method should return a decimal number (type `double` or `float`), but this method doesn't return any value.

Option (b) is correct. This method accepts three integer values: `byte`, `int`, and `int`. It computes the sum of these integer values and returns it as a decimal number (data type `double`). Note that the name of the method is `subtractNumbers`, which doesn't make it an invalid option. Practically, one wouldn't name a method `subtractNumbers` if it's adding them. But syntactically and technically, this option meets the question's requirements and is a correct option.

Option (c) is incorrect. This method doesn't accept integers as the method arguments. The type of the method argument `arg3` is `double`, which isn't an integer.

Option (d) is correct. Even though the name of the method seems weird, it accepts the correct argument list (all integers) and returns the result in the correct data type (`float`).

Q3-4. Which of the following statements are true?

a **If the return type of a method is `int`, the method can return a value of type `byte`.**

b **A method may or may not return a value.**

c **If the return type of a method is `void`, it can define a return statement without a value, as follows:**

```
return;
```

d **A method may or may not accept any method arguments.**

e A method should accept at least one method argument or define its return type.

f A method whose return type is `String` can't return `null`.

Answer: a, b, c, d

Explanation: Option (e) is incorrect. There is no constraint on the number of arguments that can be passed on to a method, regardless of whether the method returns a value.

Option (f) is incorrect. You can't return the value `null` for methods that return primitive data types. You can return `null` for methods that return objects (`String` is a class and not a primitive data type).

Q3-5. Given the following definition of class `Person`,

```
class Person {
    public String name;
    public int height;
}
```

what is the output of the following code?

```
class EJavaGuruPassObjects1 {
    public static void main(String args[]) {
            Person p = new Person();
            p.name = "EJava";

            anotherMethod(p);
            System.out.println(p.name);

            someMethod(p);
            System.out.println(p.name);
    }
    static void someMethod(Person p) {
            p.name = "someMethod";
            System.out.println(p.name);
    }
    static void anotherMethod(Person p) {
            p = new Person();
            p.name = "anotherMethod";
            System.out.println(p.name);
    }
}
```

a
```
anotherMethod
anotherMethod
someMethod
someMethod
```

b
```
anotherMethod
EJava
someMethod
someMethod
```

c
```
anotherMethod
EJava
someMethod
EJava
```

d Compilation error.

Answer: b

Explanation: The class `EJavaGuruPassObject1` defines two methods, `someMethod` and `anotherMethod`. The method `someMethod` modifies the value of the object parameter passed to it. Hence, the changes are visible within this method and in the calling method (method `main`). But the method `anotherMethod` reassigns the reference variable passed to it. Changes to any of the values of this object are limited to this method. They aren't reflected in the calling method (the `main` method).

Q3-6. What is the output of the following code?

```
class EJavaGuruPassPrim {
    public static void main(String args[]) {
            int ejg = 10;
            anotherMethod(ejg);
            System.out.println(ejg);
            someMethod(ejg);
            System.out.println(ejg);
    }
    static void someMethod(int val) {
            ++val;
            System.out.println(val);
    }
    static void anotherMethod(int val) {
            val = 20;
            System.out.println(val);
    }
}
```

a
```
20
10
11
11
```

b
```
20
20
11
10
```

c
```
20
10
11
10
```

d Compilation error

Answer: c

Explanation: When primitive data types are passed to a method, the values of the variables in the calling method remain the same. This behavior doesn't depend on whether the primitive values are reassigned other values or modified by addition, subtraction, or multiplication—or any other operation.

Q3-7. Given the following signature of method `eJava`, choose the options that correctly overload this method:

```
public String eJava(int age, String name, double duration)
```

a `private String eJava(int val, String firstName, double dur)`
b `public void eJava(int val1, String val2, double val3)`
c **`String eJava(String name, int age, double duration)`**
d **`float eJava(double name, String age, byte duration)`**
e **`ArrayList<String> eJava()`**
f **`char[] eJava(double numbers)`**
g **`String eJava()`**

Answer: c, d, e, f, g

Explanation: Option (a) is incorrect. Overloaded methods can change the access modifiers, but changing the access modifier alone won't make it an overloaded method. This option also changes the names of the method parameters, but that doesn't make any difference to a method signature.

Option (b) is incorrect. Overloaded methods can change the return type of the method, but changing the return type won't make it an overloaded method.

Option (c) is correct. Changing the placement of the types of the method parameters overloads it.

Option (d) is correct. Changing the return type of a method and the placement of the types of the method parameters overloads it.

Option (e) is correct. Changing the return type of a method and making a change in the parameter list overload it.

Option (f) is correct. Changing the return type of a method and making a change in the parameter list overload it.

Option (g) is correct. Changing the parameter list also overloads a method.

Q3-8. Given the following code,

```
class Course {
    void enroll(long duration) {
            System.out.println("long");
    }
    void enroll(int duration) {
            System.out.println("int");
    }
    void enroll(String s) {
            System.out.println("String");
```

```
    }
    void enroll(Object o) {
            System.out.println("Object");
    }
}
```

what is the output of the following code?

```
class EJavaGuru {
    public static void main(String args[]) {
            Course course = new Course();
            char c = 10;
            course.enroll(c);
            course.enroll("Object");
    }
}
```

a Compilation error

b Runtime exception

c **`int`**
`String`

d `long`
`Object`

Answer: c

Explanation: No compilation issues exist with the code. You can overload methods by changing the type of the method arguments in the list. Using method arguments with data types having a base-derived class relationship (`Object` and `String` classes) is acceptable. Using method arguments with data types for which one can be automatically converted to the other (`int` and `long`) is also acceptable.

When the code executes `course.enroll(c)`, `char` can be passed to two overloaded `enroll` methods that accept `int` and `long`. The `char` gets expanded to its nearest type—`int`—so `course.enroll(c)` calls the overloaded method that accepts `int`, printing `int`. The code `course.enroll("Object")` is passed a `String` value. Although `String` is also an `Object`, this method calls the specific (not general) type of the argument passed to it. So `course.enroll("Object")` calls the overloaded method that accepts `String`, printing `String`.

Q3-9. Examine the following code and select the correct options:

```
class EJava {
    public EJava() {
            this(7);
            System.out.println("public");
    }
    private EJava(int val) {
            this("Sunday");
            System.out.println("private");
    }
```

```
    protected EJava(String val) {
            System.out.println("protected");
    }
}

class TestEJava {
    public static void main(String[] args) {
            EJava eJava = new EJava();
    }
}
```

a **The class EJava defines three overloaded constructors.**

b The class `EJava` defines two overloaded constructors. The private constructor isn't counted as an overloaded constructor.

c Constructors with different access modifiers can't call each other.

d **The code prints the following:**

```
protected
private
public
```

e The code prints the following:

```
public
private
protected
```

Answer: a, d

Explanation: You can define overloaded constructors with different access modifiers in the same way that you define overloaded methods with different access modifiers. But a change in only the access modifier can't be used to define overloaded methods or constructors. `private` methods and constructors are also counted as overloaded methods.

The following line of code calls `EJava`'s constructor, which doesn't accept any method argument:

```
EJava eJava = new EJava();
```

The no-argument constructor of this class calls the constructor that accepts an `int` argument, which in turn calls the constructor with the `String` argument. Because the constructor with the `String` constructor doesn't call any other methods, it prints `protected` and returns control to the constructor that accepts an `int` argument. This constructor prints `private` and returns control back to the constructor that doesn't accept any method argument. This constructor prints `public` and returns control to the `main` method.

Q 3-10. Select the incorrect options:

a **If a user defines a `private` constructor for a `public` class, Java creates a `public` default constructor for the class.**

b A class that gets a default constructor doesn't have overloaded constructors.

c **A user can overload the default constructor of a class.**

d The following class is eligible for default constructor:

```
class EJava {}
```

e The following class is also eligible for a default constructor:

```
class EJava {
            void EJava() {}
}
```

Answer: a, c

Explanation: Option (a) is incorrect. If a user defines a constructor for a class with any access modifier, it's no longer an eligible candidate to be provided with a default constructor.

Option (b) is correct. A class gets a default constructor only when it doesn't have any constructor. A default or an automatic constructor can't exist with other constructors.

Option (c) is incorrect. A default constructor can't coexist with other constructors. A default constructor is automatically created by the Java compiler if the user doesn't define any constructor in a class. If the user reopens the source code file and adds a constructor to the class, upon recompilation no default constructor will be created for the class.

Option (d) is correct. Because this class doesn't have a constructor, Java will create a default constructor for it.

Option (e) is also correct. This class also doesn't have a constructor, so it's eligible for the creation of a default constructor. The following isn't a constructor because the return type of a constructor isn't void:

```
void EJava() {}
```

It's a regular and valid method, with the name same as its class.