Module: Core Java

Session 2: Classes, Objects and Constructors

* Introduction
* Class Fundamentals
* The General Form of a Class
* Defining a Simple Class
* Declaring a Class
* Declaring Fields for a Class
* Defining a Constructor
* Creating an Object by Calling a Constructor
* Introducing Methods
* Objects
* Using the ‘this’ Keyword
* Garbage Collector
* The finalize() method

**Objective**

At the end of the chapter, you will be able to:

* Define Classes, fields, objects, Constructors and methods etc.
* Define and declare a Simple Class by creating an object
* Define a Constructor
* Create instance by declaring a Constructor
* Create an Object by Calling a Constructor
* Learn how methods calls each other
* Make out where to use ‘this’ keyword
* Learn the basic of Garbage Collector
* Understand the function of finalize() method

**Introduction**

## There is no doubt that ‘Class’ is at the core of object Oriented Architecture. It is a construct that is used in

## the entire Java language. ‘Class’ in turn implements the Object Oriented Programming features and

## defines the shape and nature of an object. Any concept, code or logic that you wish to implement in a

## Java program must be encapsulated within a “Class”. In this chapter you will be introduced to the basic

## elements of a class. You will learn how a class can be used to create objects. You will also gain

## knowledge about the methods, constructors, Inheritance, Polymorphism and the ‘this’keyword.

**Class Fundamentals**

## A class is nothing but a blueprint or a template for creating different objects, which defines its properties

## and behaviors. Java class objects exhibit the properties and behaviors defined by its class. A class can

## contain fields and methods to describe the behavior of an object.

**The General Form of a Class**

## When you define and create a class, you have to declare its exact attributes & behaviors required. You do

## this by specifying the attributes that it will contain and the behavior in the form of code that operates on

## the same data. While the very simple classes may contain only behavior (methods) or only attributes

## (data), most real-world classes contain both.

## You can declare a class by using the ‘class’ keyword.

## Following is the example of a general form of a class:

class classname {

type instance-variable1;

type instance-variable2;

// ...

type instance-variableN;

type methodname1(parameter-list) {

// body of method

}

type methodname2(parameter-list) {

// body of method

}

// ...

type methodnameN(parameter-list) {

// body of method

}

}

## The data or attributes, defined within a class are called instance variables as each instance of the class

## (that is, each object of the class) contains its own copy of these variables. The code is contained within

## behavior (methods). A class contains methods and attributes. These methods and attributes are

## collectively known as members of a class. Methods (which are accessible outside of the class) that are

## defined for the same classes can act upon and access instance variables within that class itself. This

## happens for most classes. How the data of a class can be used is thus decided by the method.

## The main () method has the same general form as all the other methods. Most of the methods, however,

## do not have access specifier as static or public.

## You can specify multiple classes within a program. You should keep in mind that every class does not

## consist of a main() method. You can only specify one, in that class which will be executed by the Java

## Runtime.

**Defining a Simple Class**

## Now you may consider that a class is a sort of a template for an object. In this way, a class is equivalent

## to a data type such as int. The main difference is that Java already knows what an integer is. However,

## when you create a class, you must tell Java about the class's distinctiveness. You define a class by using

## the ‘class’ keyword along with the class name, like this:

class MyClass

{

}

## Believe it or not, the preceding lines are a complete Java class. If you save the lines in a file called

## MyClass.java, you could even compile the class into a .CLASS file, although the file won't actually do

## anything if you tried to run it. As you can see, the class definition begins with the keyword ‘class’ followed

## by the name of the class. The body of the class is marked off by curly braces just like any other program

## block. In this case, the class's body is empty.

## Because its body is empty, this example class doesn't do anything. You can, however, compile the class

## and even create an object from it. To create an object from a class, you need to type the class's name

## followed by the name of the object. For example, the line below creates an object from the MyClass class:

MyClass myObject = new MyClass();

## Declaring a Class

You have seen how classes are declared:

class *MyClass* {

//field, constructor, and method declarations

}

In this class declaration, the class body i.e. the area between the braces, contains all the code that provides for the life cycle of the objects created form the class. This might include constructors for initializing new objects, declarations for the fields that provide the state of the class and its objects, and methods to implement the behavior of the class and its objects.

This class declaration is a minimal one, containing only the required components of a class declaration. More developed instances of class declaration are also available. For instance, you can give more information about the class –i.e. the name of its superclass, whether any interfaces are implemented by it and similar information. You need to provide this information at the start of the class declaration. For example:

class MyClass extends MySuperClass implements YourInterface {

//field, constructor, and method declarations

}

Here MyClass is a subclass of MySuperClass and it implements the interface - YourInterface.

Modifiers like public or private can be also be added at the beginning of the declaration. These two modifiers decide which other classes can access MyClass. However, using these modifiers can make the opening line of a declaration far too complicated.

So what is the order in which class declaration includes its components?

1. Wherever modifiers like public and private (and similar ones) they come first.
2. Second comes the class name. The convention is to capitalize the initial letter.
3. After that comes the name of the class’s parent, i.e. the superclass. The keyword *extends* precedes this superclass. Remember that a class can extend only one parent, i.e. subclass.
4. If there is a list of comma-separated interfaces, the keyword implement precedes them. A class can implement more than one interface.
5. Finally, the class body. This contains all the information within that class and comes surrounded by braces.

## Declaring Fields for a Class

In order to be useful, a class needs both data fields and methods. You declare fields for your class in much the same way you declare any variable in a program, by typing the data type of the field followed by the name of the field, like this:

int myField;

The above line declares a data field of type integer. However, looking at the above line doesn't tell you much about how data fields are used with classes. In fact, you can't deduce from the above line whether myField is actually part of an object or just a normal variable. To clear up this ambiguity, you can plug the above line into the MyClass class definition, as shown below.

class MyClass

{

int myField;

}

Now you can see that myField is a data field of the MyClass class. Moreover, this data field is by default accessible only by methods in the same package. You can change the rules of this access by using the public, protected, and private keywords. A public data field can be accessed by any part of a program, inside or outside of the class in which it's defined. A protected data field can only be accessed from within the class or from within a derived class (a subclass). A private data field cannot even be accessed by a derived class.

## Defining a Constructor

You have now added a data field to MyClass. However, the class has no methods and so can do nothing with its data field. The next step is to create methods. One special type of method, called as constructor, enables an object to initialize itself when it's created. A constructor is a public method (constructors can also be private) with the same name as the class. Following code shows the MyClass class with its two constructors in place.

class MyClass

{

int myField;

public MyClass() //Default constructor

{

myField=20;

}

public MyClass(int myfield) // Constructor with one argument

{

myField = myfield;

}

}

As you can see, the class's constructor starts with the ‘public’ keyword. This is important because you want to be able to create an object from the class anywhere in your program, and when you create an object, you're actually calling its constructor. After the public keyword comes the name of the constructor followed by the constructor's arguments in parentheses. When you create an object of the class, you must also provide the required arguments.

## Creating an Object by Calling a Constructor

MyClass contains two constructors, one without argument and one with argument. If you want to create object by calling default constructor, no need to supply any argument. But if you want to create an object from MyClass by calling Constructor with argument, you must supply an integer value that the class uses to initialize the myField data field. You can create an object of the class like this:

MyClass myObject1 = new MyClass(); //calling default constructor

Above line not only creates an object of the MyClass class, but also initializes the myField data field to 20.

MyClass myObject2 = new MyClass(10); //calling constructor with

//argument

Above line not only creates an object of the MyClass class, but also initializes the myField data field to 10.

## The first word in the object declaration lines tell Java that myObject1 and myObject2 are going to be

## objects of the MyClass class. The next word is the object's name. After the equals sign comes the

## keyword ‘new’ and the call to the class's constructor.

### Introducing Methods

Java method is a series of statements that perform some repeated task. Instead of writing 10 lines of code we can put those ten lines in a method and just call it in one line. It is like a shortcut.

For example if we have to repeatedly output a address such as:

System.out.println(“SQL Star International Ltd”);  
System.out.println(“Sanali Info Park, Block B, 4th Floor”);  
System.out.println(“Road No 2, Banjara Hills”);   
System.out.println(“Hyderabad - 500 033”);  
System.out.println(“Andhra Pradesh, India”);  
System.out.println(“T: 040 - 23101600 - 603”);  
System.out.println(“F: 040 – 23101663”);

We can put it all in a method like this:

public static void printAddress(){

System.out.println(“SQL Star International Ltd”);  
System.out.println(“Sanali Info Park, Block B, 4th Floor”);  
System.out.println(“Road No 2, Banjara Hills”);   
System.out.println(“Hyderabad - 500 033”);  
System.out.println(“Andhra Pradesh, India”);  
System.out.println(“T: 040 - 23101600 - 603”);  
System.out.println(“F: 040 – 23101663”);

}

To call printAddress() method we simply write:

Object\_of\_class.printAddress();

The methods that can be called from the outside of the class should be defined as public; the methods that can be callable only from the class and its derived classes should be defined as protected, and the methods that is callable only from within the class should be declared as private.

Suppose myField is defined as private, and you now want to set the value of myField from outside the MyClass class. As data field is defined as private, which means that it can be accessed only within the same class, you cannot access it directly by name. To solve this problem, you might also want to create a method that returns the value of the field, as shown in MyClass.java program.

Example: MyClass.java

public class MyClass

{

private int myField;

public MyClass()

{

myField = 20;

}

public MyClass(int myfield)

{

myField = myfield;

}

public int getField() //method

{

return myField;

}

public static void main(String args[]) //main method

{

MyClass object1 = new MyClass();

MyClass object2 = new MyClass(100);

int ob1\_value = object1.getField(); //will return 20

int ob2\_value = object2.getField(); //will return 100

System.out.println(ob1\_value);

System.out.println(ob2\_value);

}

}

## Output:

## 20

## 100

### Objects

What are objects? You can term them as the physical instantiation of the classes i.e. they represent an idea in the form of an instance of it. Objects have their own independent identities and life cycles of within a program. Objects are created completely in accordance with the class that describes them. Imagine the blueprint of a house. You can create as many houses as you want from the one blueprint. Similarly, in JAVA one class can instantiate many objects. In JAVA, you can create many objects of different classes. Moreover, you can use them and finally destroy them within the course of the program.

### Declaring, Instantiating and Initializing Objects

## A class provides the blueprint for objects; you create an object from a class. The following two statements

## taken from the Myclass.java program creates an object and assigns it to a variable.

**int ob1\_value**  = object1.getField();

**int ob2\_value** = object2.getField();

Both the statements have three parts:

1. **Declaration:** The code that is set in bold is a variable declaration, associating a variable name with an object type.
2. **Instantiation:** This is done by the new keyword, a JAVA operator that creates the object.
3. **Initialization:** A call to the constructor follows this new operator. This in turn, initializes the new object.

### Using the “this” keyword

“this” keyword is a reference to the current object within an instance method or constructor. Current object is the object whose method or constructor is being called. You can use “this” keyword to refer to any member of the current object from within an instance method or constructor. This solves the variable shadowing issue.

For example, the Myclass class was written like this:

public class MyClass

{

private int myField;

public MyClass(int myfield) //constructor with argument

{

myField = myfield;

}

}

Now, see what happens if you write this program, using “this” with a field:

public class MyClass

{

private int myField;

public MyClass(int myfield) //constructor with argument

{

this.myField = myfield;

}

}

What is the necessity of using “this” here? Notice that each argument to the constructor shadows one of the object’s fields. Yet, inside the constructor, “myField” is a local copy of the first argument of the constructor. Using “this” you can refer to the MyClass field myField. This happens only if the constructor uses this.myField.

You can use the “this” keyword for further uses as well. For example, you can use it to call another constructor in the same class. This process is known as explicit constructor invocation. Consider the following example where Myclass uses “this” with a constructor and notice the difference from the first instance.

public class MyClass {

private int x, y, a, b;

public MyClass() {

**this(0, 0, 0, 0);**

}

public (int a, int b) {

**this(0, 0, a, b);**

}

public Myclass(int x, int y, int a, int b) {

this.x = x;

this.y = y;

this.a = a;

this.b = b;

}

...

}

As you can see, a set of constructors is available within this class. Either some, or all of the member variables are initialized by each of the constructors. A default value for any member variable whose initial value is not provided by an argument is provided by the constructors. Here, for instance, the four-argument constructor with four 0 values is called by the no-argument constructor. Similarly, the four argument constructor with two 0 values is called by the two-argument constructor. You might ask, how is it determined that which constructor is to be called? The compiler decides that, based on the number and type of arguments.

### The Garbage Collector

In some object-oriented languages you need to keep track of all the objects that you have created as well as destroy them when they are not needed any longer. This means that managing memory becomes highly tedious. This might give a lot of errors as well. The JAVA platform handles this situation much more efficiently. Depending on the system’s handling capacity, you can create as many as objects as you need to without worrying about destroying them. The JAVA runtime environment determines that for you. It deletes objects when it determines that they are not in use any more. This process is known as garbage collection.

How is it determined that an object is eligible for garbage collection? As you might have noticed, when they are no longer in use, i.e. when there are no more references to that object are available. This can happen in two ways. Either, references that are held in a variable are dropped when that variable goes out of scope; or an object reference can be explicitly dropped by setting the variable to the special value null. A program might have multiple references to the same object. Remember that all the reference to an object must be dropped to make the object eligible for garbage collection.

The garbage collector in the JAVA runtime environment does this automatically, as it periodically frees the memory used by un-referenced objects.

### Advantages of garbage collection

The main advantage of garbage collection is that it is a very handy process. The JAVA Virtual Machine helps on two counts:

1. This saves a lot of time for the human programmer by relieving him off the tedious and tricky job of freeing the allocated memory manually. This saves a lot of time for the human programmer.
2. The second advantage of garbage collection is that this process helps in maintaining the program integrity. Since garbage collection is done automatically, it does not allow the programmer to crash the JVM by incorrectly freeing the memory. Garbage collection enhances JAVA’s security strategy.

### Disadvantages of garbage collection

Garbage collection has its own disadvantages.

1. The program performance can be affected through garbage collection. Why? It is because the garbage-collected heap adds an overhead. Moreover, this process might take a bit more of the CPU time than usual, as the JVM needs to keep track of the objects as they are being referenced by the executing program as well as finalize and free object at the same time. If you had the option of freeing the unnecessary memory explicitly, it would have taken less time. In a nutshell, the programmer has less control over scheduling the CPU time that is reserved for freeing unnecessary objects.
2. If a programmer really wants to explicitly request a garbage collection at some point, System.gc() or Runtime.gc() can be invoked, which is a "hint" to the runtime engine that the present moment might be a good time to run the GC. Some implementations take this into consideration and some do not. Strictly speaking no one can force GC.

## The finalize() Method

Think of a situation when an object might actually perform some actions when it is just about to be destroyed. For instance, think of the situation, when an object is holding some non-JAVA resources like a file handle or a window character font. You want to free these resources before the object is destroyed. JAVA provides the process called finalization for situations like this. Through finalization you can define the specific actions that will take place when an object is just about ready to go for garbage collection.

To implement this, you simply define the **finalize()** method. Whenever an object of that class needs to be recycled, JAVA run time calls the **finalize()** method. As you know, the garbage collector runs automatic and periodic checks to find unreferenced objects. Note that the garbage collector runs periodic automatic checks for those objects that are no longer referenced by any running state. It can also apply the indirect method of checking through other referenced objects for a long period.

Before an object is ready for destruction by the Garbage Collector, any specified actions within **finalize()** will be performed. To do this, the JAVA run time environment calls the **finalize()** method, just when an asset is about to be freed.

The general form of a **finalize()** might look like this:

protected void finalize()

{

// finalization code goes here

}

Notice that here the keyword **protected** is a specifier. This prevents access to the **finalize()** method by defining a code outside its class. This determines a central feature of **finalize()** –it is called only before garbage collection, not when an object goes out-of-scope.

The GCTest.java illustrates when a string Object becomes available for Garbage Collection.

public class GCTest {

public static void main(String args[]) {

String a,b;

String c = new String("test");

a = c;

c = null; // The String "test" is not yet

//available for GC as “a” still points to "test"

b = new String("xyz");

b = c; // String "xyz" is now available for GC.

a = null;

//String "test" is now available for GC.

}}

/\*\* Example shows garbage collector in action Note that the finalize()method of object GC1 runs without being specifically called and that the id's of garbage collected objects are not always sequential.

\*/

class TestGC {

public static void main(String[] args) {

Runtime rt = Runtime.getRuntime();

System.out.println("Available Free Memory: " + rt.freeMemory());

for(int i=0; i<10000; i++ ) {

GC1 x = new GC1(i);

}

System.out.println("Free Memory before call to gc(): " +

rt.freeMemory());

System.runFinalization();

System.gc();

System.out.println(" Free Memory after call to gc(): " +

rt.freeMemory());

}

}

class GC1 {

String str;

int id;

GC1(int i) {

this.str = new String("abcdefghijklmnopqrstuvwxyz");

this.id = i;

}

protected void finalize() {

System.out.println("GC1 object " + id + " has been finalized.");

}

}

**Summary:**

After going through this chapter, you will be able to comprehend and define classes, objects, methods, constructors and garbage collection. You will learn how to create a new class. This apart, you will also learn how to create new objects, variables, constructors and methods in a class. By the end of the chapter you will develop a pretty good idea how to use garbage collection and finalize() method in a beneficial way.

In a nutshell, after studying this chapter you will have a clear concept about the following:

* ‘class’ is the logical construct upon which the entire Java language is built.
* ‘class’ is a template for an object, and an object is an instance of a class.
* ‘class’ is declared by use of the class keyword.
* The data, or variables, defined within a class are called instance variables.
* Objects are the physical instantiations of classes. Many objects can be instantiated from one class.
* A constructor is a method with the same name as its class that creates an object.
* Java constructors are invoked using the new keyword.
* ‘this’ keyword is a special type of reference variable, which is used to refer the current instance of an object.
* ‘method’ is a series of statements that perform some repeated task.
* Java runtime environment deletes objects when it determines that they are no longer being used, this process is called garbage collection.
* Every class inherits the finalize() method from java.lang.Object.
* The finalize() method is called by the garbage collector when it determines no more references to the object exist.
* The finalize() is never run more than once on any object.