Classes and Object in Java

What is the Class?

A class is a collections/group of objects which have common properties. It is a template or blueprint from which objects are created. It means a class is the **specification or template of an object**. Or we can said that user defined datatype.

Syntax:

```
Access-modifier class ClassName {
        // Class body starts here.
        // Members of a class.
    1. Field declarations;
    2. Constructor declarations;
    3. Method declarations;
    4. Instance block declarations;
    5. Static block declarations;
}
```

The class body (the area between the curly braces) contains all the code that provides for the life cycle of the objects created from the class: constructors for initializing new objects, declarations for the fields that provide the state of the class and its objects, and methods to implement the behaviour of the class and its objects and etc.

Example:

```
        class Box
        {
                int length, width, height;
        }
```

Note: class does not occupy memory because class is not real time entity. Memory Occupied for object because object is a runtime entity.

What is an Object?

- The Object is the real-time entity having some state and behaviour.
- In Java, Object is an instance of the class having the instance variables as the state of the object and the methods as the behaviour of the object.
- Creating an object means allocating memory to store the data of variables temporarily. i.e. we create an object of a class to store data temporarily.
- The object of a class can be created by using the new keyword in Java Programming language.

- A class is a template or blueprint from which objects are created. So, an object is the instance (result) of a class.
- Creating an object is also called **instantiating an object**. By creating an object, we can access members of any class.

Syntax to declare an object

ClassName objectname;

When we only declared object of a class type, only a reference is created (memory is not allocated for the object). To allocate memory to an object, we must use new().

New Keyword in Java

- In Java, a new operator is a special keyword which is used to create an object of the class. It allocates the memory to store an object during runtime and returns a reference to it.
- This reference is the address of the object in the heap memory allocated by the new operator.
- This reference (memory address) is then stored in a variable called object reference variable that can be accessed from anywhere in the application

Instantiating a Class

The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The new operator also invokes the object constructor.

Example:

```
Box b1 = new Box();
```

Hash Code Number

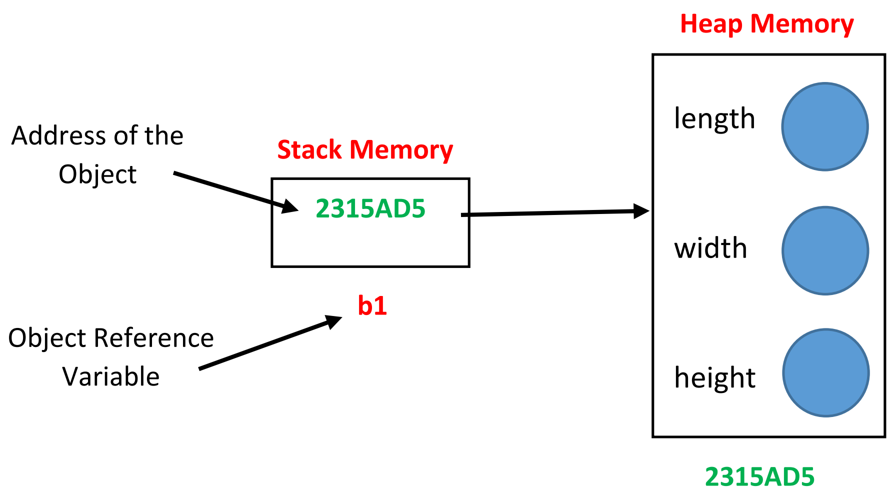
- When an object of class is created, the memory is allocated in the heap area to store instance variables.
- After the creation of an object, **JVM** produces a unique reference number (address) for that object.
- This unique reference number is called **hash code number**.
- This reference number is unique for all objects, except string objects. The address of the object is stored in the object reference variable in the stack memory.
- We can know the hash code number or reference number (address) of an object by using hashCode() method of **Object class.**

Example:

```
Box b1 = new Box();
System.out.println(b1.hashCode()); // it will display hash code stored in b1.
```

Memory Allocation for Storing an Object in Java

Let’s understand the memory allocation for storing an object in Java by given below figure.



Note:

- **b1 stores the address of the object on the stack memory.**
- **New keyword allocates memory to store the object on the heap memory**

Access Modifiers:

Access modifiers are keywords that can be used to control the visibility of fields, methods, and constructors in a class.

Java supports four types of access modifiers:

1. **Private**
2. **Default (no access modifier specified)**
3. **Protected**
4. **Public**

1. **Private Access Modifier:** We can access the **private modifier** only within the same class and not from outside the class.

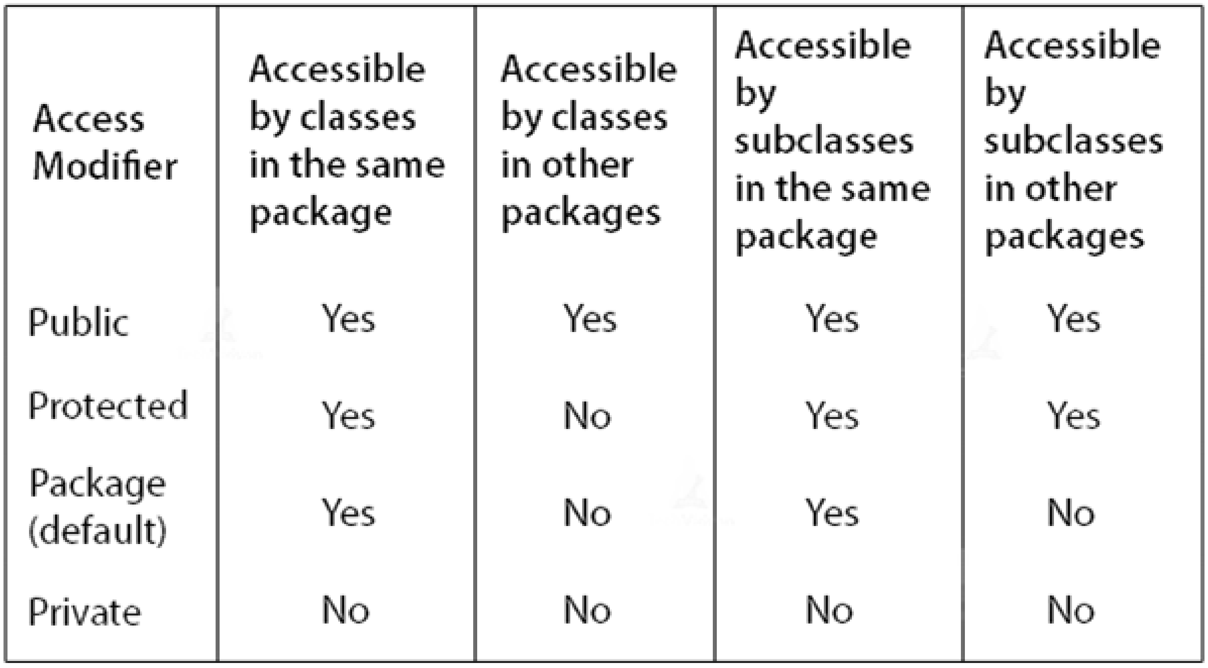
2. **Default Access Modifier (no access modifier specified):** It is not a keyword. Any Java members such as class or methods or data members when not specified with any access modifier they are by default considered as **default access modifiers.** These methods or data members are **only accessible within the same package** and they cannot be accessed from outside the package. It provides more visibility than a private access modifier. But this access modifier is more restricted than protected and public access modifiers.

3. **Protected Access Modifier:** We can access the protected modifier within the same package and also from outside the package with the help of the child class. If we do not make the child class, we cannot access it from outside the package. So inheritance is a must for accessing it from outside the package. We can also say that the protected access modifier is similar to default access modifier with one exception that it has visibility in subclasses.

4. **Public Access Modifier:** If a class or its members are declared as public, then we can access the public modifier from anywhere. We can access public modifiers from within the class as well as from outside the class and also within the package and outside the package.
It is comparable to a public place in the real world, such as a company cafeteria that all employees can use irrespective of their department.
Let us see which all members of Java can be assigned with the access modifiers:

Members of JAVA	Private	Default	Protected	Public
Class	No	Yes	No	Yes
Variable	Yes	Yes	Yes	Yes
Method	Yes	Yes	Yes	Yes
Constructor	Yes	Yes	Yes	Yes
interface	No	Yes	No	Yes
Initializer Block	NOT ALLOWED			

Scope of these access modifiers



Access Modifier	Accessible by classes in the same package	Accessible by classes in other packages	Accessible by subclasses in the same package	Accessible by subclasses in other packages
Public	Yes	Yes	Yes	Yes
Protected	Yes	No	Yes	Yes
Package (default)	Yes	No	Yes	No
Private	No	No	No	No

Object Initialization in Java

The process of assigning a value of the variable is called initialization of state of an object. In other words, initialization is the process of storing data into an object.

Program:

```
public class Box {
          int length = 13;
          int width = 14;
          int height = 15;

          public static void main(String[] args) {
                    Box b1 = new Box();

                    System.out.println("Box Dimension:"+b1.length+"x"+b1.width+"x"+b1.height);
                    System.out.println("Box Volume:"+(b1.length*b1.width*b1.height));
          }
}
```

How to Initialize State of Object in Java?

There are three ways by which we can initialize state of an object. In other words, we can initialize the value of variables in Java by using three ways as

1. **By using Reference Variable**
2. **By using a method**
3. **By using a constructor.**

By using Reference Variable

We can initialize the value of objects through the reference variable as

```
public class Box {
          int length, width, height;
          public static void main(String[] args) {
                    Box b1 = new Box();

                    b1.length=13;
                    b1.width=14;
                    b1.height=15;

                    System.out.println("Box Dimension:"+b1.length+"x"+b1.width+"x"+b1.height);
                    System.out.println("Box Volume:"+(b1.length*b1.width*b1.height));
          }
}
```

Program: If we declare data member as public access modifier.

```
public class Box {

          public int length, width, height;
}
```

```
public class PublicAccessModifier {

	public static void main(String[] args) {
		Box b1 = new Box();
		b1.length=13;
		b1.width=14;
		b1.height=15;
		System.out.println("Box Dimension:"+b1.length+"x"+b1.width+"x"+b1.height);
		System.out.println("Box Volume:"+(b1.length*b1.width*b1.height));
	}
}
```

Program: if we declare data member as private access modifier.

```
public class Box {
	private int length, width, height;
}
public class PrivateAccessModifier {
	public static void main(String[] args) {
		Box b1 = new Box();
		b1.length=13;
		b1.width=14;
		b1.height=15;
		/ *The above member can not be access because Box class field is private which is
		    not accessible outside class by its object. */
	}
}
```

The above member cannot be access because Box class field is private which is not accessible outside class by its object. But if we want to access or set values to private member of class then we can implement getter and setter methods. Because private member can access by only its methods of class.

By using Method

Defining Method

Syntax:

```
AccessModifier returntype methodName([Parameter List])
{
	---------
	---------
}
```

Program: Using Getter and Setter Method

```
public class Box {
	int length, width, height;
	public void setDimension(int l, int w, int h) {
```

```
            length=l;
            width=w;
            height=h;
    }
    public void setLength(int l) {
            length=l;
    }
    public void setWidth(int w) {
            width=w;
    }
    public void setHeight(int h) {
            height=h;
    }
    public int getLength() {
            return length;
    }
    public int getWidth() {
            return width;
    }
    public int getHeight() {
            return height;
    }
    public void showDimension() {
            System.out.println("Box Dimension:"+length+"x"+width+"x"+height);
    }
    public int getVolume() {
            return length*width*height;
    }
    public static void main(String[] args) {
            Box b1 = new Box();
            b1.setLength(13);
            b1.setWidth(14);
            b1.setHeight(15);
            System.out.println("Box b1");
            b1.showDimension();
            System.out.println("Box Volume:"+ b1.getVolume());

            Box b2 = new Box();
            b2.setDimension(7, 10, 12);
            System.out.println("Box b2");
            b2.showDimension();
            System.out.println("Box Volume:"+ b2.getVolume());
    }
}
```
In above program we can see how to assign or initialise value to instance variable. For this, we can create first object of Box class then by using setter method we can set value to variable. Suppose if

we want to initialize value to instance variable at the time of creation of object then we can go with the concept of **constructor**.

By using Constructors

A constructor is a block of codes similar to the method. It is called when an instance of the object is created, and memory is allocated for the object. Every class has a constructor. If you do not explicitly write a constructor for a class, then Java compiler provides a default constructor (without any parameter) for that class.

Rules for creating Java constructor

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

Types of Constructor: Constructors can be divided into 4 types:

1. Default Constructor
2. No-Argument Constructor
3. Parameterized Constructor
4. Copy Constructor

Default Constructor

If we do not create any constructor, the Java compiler automatically create a no-argument constructor during the execution of the program. This constructor is called default constructor.

Example 1: Default constructor

```
public class Box {
	private int length, width, height;
	public void ShowDimension() {
		System.out.println("Box Dimension:"+length+"x"+width+"x"+height);
	}
	public static void main(String[] args) {
		Box b1 = new Box();
		b1.ShowDimension();
	}
}
```

Java No-Argument Constructors

If a constructor does not accept any parameters, it is known as a no-argument constructor.

Example 2: private No-Argument constructor

```
 class Main {
	int i;
	// constructor with no parameter
	private Main() {

		i = 5;
		System.out.println("Constructor is call");
```

```
          }
          public static void main(String[] args) {
                   // calling the constructor without any parameter
                   Main obj = new Main();
                   System.out.println("Value of i: " + obj.i);
  }
```

Note: Once a constructor is declared private, it cannot be accessed from outside the class. So, creating objects from outside the class is prohibited using the private constructor. Here, we are creating the object inside the same class. Hence, the program is able to access the constructor. But if we want to create objects outside the class, then we need to declare the constructor as public.

Example 3: public No-Argument constructor

```
 class Company {
        String name;
        public Company() {
             name = "ShreeSoft";
        }
 }
 class Main {
        public static void main(String[] args) {
             Company obj = new Company();// object is created in another class
             System.out.println("Company name = " + obj.name);
        }
 }
```

Parameterized Constructor:

A Java constructor can also accept one or more parameters. Such constructors are known as parameterized constructors (constructor with parameters).

Example 4: Parameterized constructor

```
public class Box {
         private int length, width, height;
         public Box(int l, int w, int h) {
                  length=l;
                  width=w;
                  height=h;
         }
         public void ShowDimension() {
                  System.out.println("Box Dimension:"+length+"x"+width+"x"+height);
         }
         public static void main(String[] args) {
                  Box b1 = new Box(7,13,16);
                  b1.ShowDimension();
         }
}
```

Copy Constructor:

A Copy Constructor in Java is a special type of constructor that is used to create a new object using the existing object of a class that we have created previously. It creates a new object by initializing the object with the instance of the same class.

Example 4: Copy constructor

```
public class Box {
	private int length, width, height;
	public Box(int l, int w, int h) {
		length=l;
		width=w;
		height=h;
	}
	public Box(Box b) {
		length=b.length;
		width=b.width;
		height=b.height;
	}
	public void ShowDimension() {
		System.out.println("Box Dimension:"+length+"x"+width+"x"+height);
	}
	public static void main(String[] args) {
		Box b1 = new Box(7,13,16);
		Box b2 = new Box(b1);
		b1.ShowDimension();
		b2.ShowDimension();
	}
}
```

Constructors Overloading in Java

We can also create two or more constructors with different parameters. This is called constructors overloading.

Example: Constructor Overloading

```
public class Box {
	private int length, width, height;
	public Box() {
		length=width=height=7;
	}
	public Box(int k) {
		length=width=height=k;
	}
	public Box(int l, int w, int h) {
		length=l;
		width=w;
		height=h;
	}
```

```
        public Box(Box b) {
                length=b.length;
                width=b.width;


                height=b.height;
        }
        public void ShowDimension() {
                System.out.println("Box Dimension:"+length+"x"+width+"x"+height);
        }
        public int getVolume() {
                return length*width*height;
        }
        public static void main(String[] args) {
                Box b1 = new Box(); //Default Constructor
                Box b2 = new Box(5); // Parameterised Constructor
                Box b3 = new Box(7,13,16); // Parameterised Constructor
                Box b4 = new Box(b2); //Copy Constructor

                System.out.println("Box b1");
                b1.ShowDimension();
                System.out.println("Box Volume:"+b1.getVolume());

                System.out.println("\nBox b2");
                b2.ShowDimension();
                System.out.println("Box Volume:"+b2.getVolume());

                System.out.println("\nBox b3");
                b3.ShowDimension();
                System.out.println("Box Volume:"+b3.getVolume());

                System.out.println("\nBox b4");
                b4.ShowDimension();
                System.out.println("Box Volume:"+b4.getVolume());
        }
}
```

Anonymous object in Java

- An object which has no reference variable is called anonymous object in Java.
- Anonymous means nameless.
- If you want to create only one object in a class then the anonymous object is a good approach.

Syntax for Declaration of Anonymous object

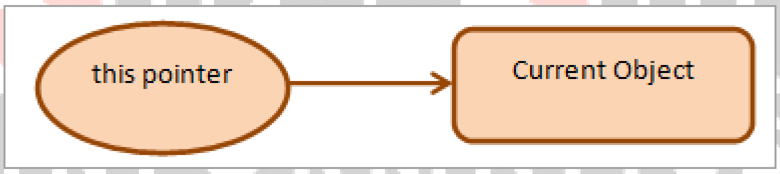
```
                new ClassName();
```

Example:

```
public class Box {
	int length, width, height;
	public Box(int l, int w, int h) {
		length=l;
		width=w;
		height=h;
	}
	public void showVolume() {
		System.out.println("Box volume:"+length*width*height);
	}
	public static void main(String[] args) {
		new Box(7, 10, 12).showVolume();
		new Box(13,14,15).showVolume();
	}
}
```

this keyword:

The reference ‘this’ is generally termed as ‘this pointer’ as it points to the current object. The ‘this pointer’ is useful when there is some name for the class attributes and parameters. When such a situation arises, the ‘this pointer’ eliminates the confusion as we can access parameters using ‘this’ pointer.



When to Use ‘this’ In Java?

1. The reference ‘this’ is used to access the class instance variable.
2. You can even pass ‘this’ as an argument in the method call.
3. ‘this’ can also be used to implicitly invoke the current class method.
4. If you want to return the current object from the method, then use ‘this’.
5. If you want to invoke the current class constructor, ‘this’ can be used.
6. The constructor can also have ‘this’ as an argument.

1) Access Instance Variable Using ‘this’

Instance variables of class and method parameters may have the same name. ‘this’ pointer can be used to remove the ambiguity that arises out of this.

Example:

```
public class Complex {
	private int real, img;
	public void setComplex(int real, int img) {
		this.real=real;
		this.img=img;
	}
	public void showComplex() {
		System.out.println(real+"+"+img+"i");
```

```
        }
        public void addComplex(Complex com1, Complex com2) {
                real=com1.real+com2.real;
                img=com1.img+com2.img;
        }
        public static void main(String[] args) {
                Complex c1 = new Complex();
                c1.setComplex(7, 13);

                Complex c2 = new Complex();
                c2.setComplex(16, 18);

                System.out.println("First Complex Number:");
                c1.showComplex();

                System.out.println("Second Complex Number:");
                c2.showComplex();

                Complex c3 = new Complex();
                c3.addComplex(c1, c2);

                System.out.println("Addition:");
                c3.showComplex();
        }
}
```

In the above program, we can see that the instance variables and method parameters share the same names. We use 'this' pointer with instance variables to differentiate between the instance variables and method parameters.

2) 'this' Passed as the Method Parameter

We can also pass this pointer as a method parameter. Passing this pointer as a method parameter is usually required when you are dealing with events. Suppose you want to trigger some event on the current object/handle, then you need to trigger it using this pointer.

Example:

```
public class Test {
        private int a,b;
        public Test() {
                a=7;
                b=13;
        }
        public int getMult(Test t) {
                return t.a*t.b;
        }
        void showMult() {
                int r = getMult(this);
                System.out.println("Multiplication:"+r);
        }
        public static void main(String[] args) {
                Test t1 = new Test();
                t1.showMult();
        }
}
```

In this program, we create an object of the class Test in the main function and then call showMult() method with this object. Inside the showMult() method, 'this' pointer is passed to the getMult() method that displays the multiplication of instance variable.

3) Invoke the Current Class Method With 'this'

Just as we can pass 'this' pointer to the method, you can also use this pointer to invoke a method. If at all you forget to include this pointer while invoking the method of the current class, then the compiler adds it for you.

Example:

```
public class Test {
        private int a,b;
        public Test() {
                a=7;
                b=13;
        }
        public int getMult() {
                return a*b;
        }
        void showMult() {
                System.out.println("Multiplication:"+this.getMult());
        }
        public static void main(String[] args) {
                Test t1 = new Test();
                t1.showMult();
        }
}
```

5) Using 'this' keyword to return the current class instance:

If the return type of the method is the object of the current class, then you can conveniently return 'this' pointer. In other words, you can return the current object from a method using 'this' pointer.

Example:

```
public class Test {
        private int a,b, mult;
        public Test() {
                a=7;
                b=13;
        }
        public Test getMult() {
                mult = a*b;
                return this;
        }
        void showMult() {
                System.out.println("Multiplication:"+mult);
        }
        public static void main(String[] args) {
                Test t1 = new Test();
                t1.getMult().showMult();
        }
}
```

6) Using ‘this’ To Invoke the Current Class Constructor

We can also use ‘this’ pointer to invoke the constructor of the current class. The basic idea is to reuse the constructor. Again if you have more than one constructor in your class, then you can call these constructors from one another resulting in constructor chaining.

Example:

```
public class Test {
        private int a,b;
        public Test() {
                this(7,13);
        }
        public Test(int x, int y) {
                a=x;
                b=y;
        }
        public int getMult() {
                return a*b;
        }
        void showMult() {
                System.out.println("Multiplication:"+getMult());
        }
        public static void main(String[] args) {
                Test t1 = new Test();
                t1.showMult();
        }
}
```

7) Using ‘this’ As The Argument To Constructor

You can also pass ‘this’ pointer as an argument to a constructor. This is more helpful when you have multiple classes as shown in the following implementation.

Example:

```
class Demo {
        Test t1;
        public Demo(Test t) {
                this.t1=t;
                t1.showMult();
        }
}
public class Test {
        private int a,b;

        public Test(int x, int y) {
                a=x;
                b=y;
                Demo d1 = new Demo(this);
        }
```

```
	public int getMult() {
		return a*b;
	}
	void showMult() {
		System.out.println("Multiplication:"+getMult());
	}
	public static void main(String[] args) {
		Test t1 = new Test(7,13);
	}
}
```