OBJECT AND CLASSES

What is Class?
A class is an entity that determines how an object will behave and what the object will contain. In other words, it is a blueprint or a set of instruction to build a specific type of object.

What is an object?

An object is nothing but a self-contained component which consists of methods and properties to make a particular type of data useful.

Object determines the behavior of the class. When you send a message to an object, you are asking the object to invoke or execute one of its methods.

An object has three characteristics:

State: represents data (value) of an object.

Behavior: represents the behavior (functionality) of an object such as deposit, withdraw etc.

Identity: Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But, it is used internally by the JVM to identify each object uniquely.

For Example: Pen is an object. Its name is Reynolds, color is white etc. known as its state. It is used to write, so writing is its behavior.

Object is an instance of a class. Class is a template or blueprint from which objects are created. So object is the instance (result) of a class.

Object Definitions:

- Object is *a real world entity*.
- Object is *a run time entity*.
- Object is *an entity which has state and behavior*.
- Object is *an instance of a class*.

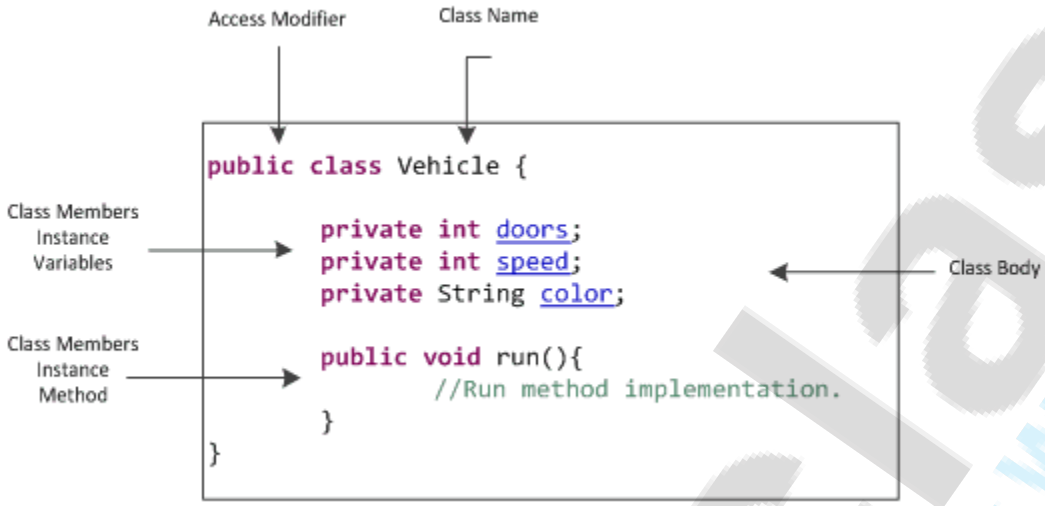
Declaration of Class

A class is declared by use of the class keyword.

The class body is enclosed between curly braces {and}.

The data or variables, defined within a class are called instance variables or local variable.

The code is contained within methods. Collectively, the methods and variables defined within a class are called members of the class.

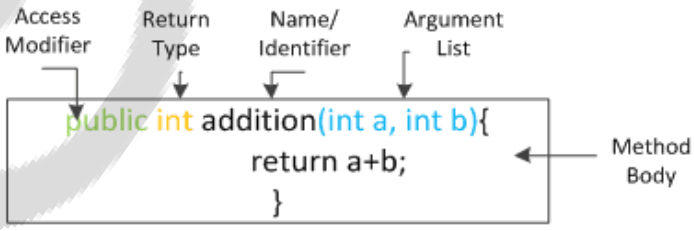


Declaration of Instance Variables:

Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another.

Declaration of Methods:

A method is a program module that contains a series of statements that carry out a task.
To execute a method, you invoke or call it from another method; the calling method makes a method call, which invokes the called method.
Any class can contain an unlimited number of methods, and each method can be called an unlimited number of times. The syntax to declare method is given below.



More generally, method declarations have

six components, in order:

1. Modifiers—such as public, private, and protected.
2. The return type—the data type of the value returned by the method, or void if the method does not return a value.
3. The method name—the rules for field names apply to method names as well, but the convention is a little different.
4. The parameter list in parenthesis—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, (). If there are no parameters, you must use empty parentheses.
5. An exception list
6. The method body, enclosed between braces—the method's code, including the declaration of local Variables

Naming a Method

Although a method name can be any legal identifier, code conventions restrict method names. By convention, method names should be a verb in lowercase or a multi-word name that begins with a verb in lowercase, followed by adjectives, nouns, etc.
In multi-word names, the first letter of each of the second and following words should be capitalized. Here are some examples:

run
runFast
getBackground

There are two basic types of methods:

Built-in: Build-in methods are part of the compiler package, such as System.out.println( ) and System.exit(0).
User-defined: User-defined methods are created by you, the programmer. These methods take-on names that you assign to them and perform tasks that you create.

Method Calling

The process of method calling is simple. When a program invokes a method, the program control gets transferred to the called method. This called method then returns control to the caller in two conditions, when –

- the return statement is executed.
- it reaches the method ending closing brace.

Following is the example to demonstrate how to define a method and how to call it –

Example

```
public class ExampleMinNumber {

  public static void main(String[] args) {
    int a = 11;
    int b = 6;
    int c = minFunction(a, b);
    System.out.println("Minimum Value = " + c);
  }

  /** returns the minimum of two numbers */
  public static int minFunction(int n1, int n2) {
    int min;
    if (n1 > n2)
      min = n2;
    else
      min = n1;

    return min;
  }
}
```

This will produce the following result –

Output

```
Minimum value =
```

CONSTRUCTOR IN JAVA

Java constructor is *invoked at the time of object creation*. It constructs the values i.e. provides data for the object that is why it is known as constructor.

Constructor in java is a special type of method that is used to initialize the object.

- A constructor doesn't have a return type.
- The name of the constructor must be the same as the name of the class.
- Unlike methods, constructors are not considered members of a class.
- A constructor is called automatically when a new instance of an object is created.

There are basically two rules defined for the constructor.

1. Constructor name must be same as its class name
2. Constructor must have no explicit return type

There are two types of constructors:

1. **Default constructor (no-arg constructor):** A constructor that have no parameter is known as default constructor.

```
Class Bike1{
Bike1()
{System.out.println("Bike is created");}

public static void main(String args[])
    { Bike1 b=new Bike1();    }
}
```

2. **Parameterized constructor:** A constructor that have parameters is known as parameterized constructor.

```
class Student4{
    int id;
    String name;

    Student4(int i,String n){
    id = i;
    name = n;
    }
    void display(){System.out.println(id+"
"+name);}
```

```
    public static void main(String args[])
{
    Student4 s1 = new Student4(111,"Kar
an");
    Student4 s2 = new Student4(222,"Ary
an");
    s1.display();
    s2.display();
  }
}
```

Java also provide **constructor overloading**

Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists. The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

```
class Student5{
    int id;
    String name;
    int age;
    Student5(int i,String n){
    id = i;
    name = n;
    }
    Student5(int i,String n,int a){
    id = i;
    name = n;
    age=a;
    }
    void display(){System.out.println(id+" "+name+
    " "+age);}

    public static void main(String args[]){
    Student5 s1 = new Student5(111,"Karan");
    Student5 s2 = new Student5(222,"Aryan",25)
;
    s1.display();
    s2.display();
   }
}
```

Copy Constructor in java

There is no copy constructor in java. But, we can copy the values of one object to another like copy constructor in C++.There are many ways to copy the values of one object into another in java. They are:

- By constructor
- By assigning the values of one object into another
- By clone() method of Object class

In this example, we are going to copy the values of **one object into another using java constructor**.

```
class Student6{
    int id;
    String name;
    Student6(int i,String n){
    id = i;
    name = n;
    }

    Student6(Student6 s){
    id = s.id;
    name =s.name;
    }
    void display(){System.out.println(id+" "+nam
    e);}

    public static void main(String args[]){
    Student6 s1 = new Student6(111,"Karan");
    Student6 s2 = new Student6(s1);
    s1.display();
    s2.display();
   }
}
```

We can copy the values of one object into another **by assigning the objects values to another object**. In this case, there is no need to create the constructor.

```
class Student7{
    int id;
    String name;
    Student7(int i,String n){
    id = i;
    name = n;
    }
    Student7(){}
  void display(){System.out.println(id+" "+name
);}
```

```
  public static void main(String args[]){
    Student7 s1 = new Student7(111,"Karan");
    Student7 s2 = new Student7();
    s2.id=s1.id;
    s2.name=s1.name;
    s1.display();
    s2.display();
   }
}
```

By clone method

```
class Student18 implements Cloneable{
int rollno;
String name;

Student18(int rollno,String name){
this.rollno=rollno;
this.name=name;
}

public Object clone()throws CloneNotSupported
Exception{
return super.clone();
}
```

```
public static void main(String args[]){
try{
Student18 s1=new Student18(101,"amit");

Student18 s2=(Student18)s1.clone();

System.out.println(s1.rollno+" "+s1.name);
System.out.println(s2.rollno+" "+s2.name);

}catch(CloneNotSupportedException c){}
      }
    }
```

Constructor chaining is a phenomena of calling one constructor from another constructor of same class. Since constructor can only be called from another constructor in Java, constructor chaining is used for this purpose.

```
classconstr_Channing
{
                constr_Channing()
                {
                this(10);

                System.out.println("IN default constructor");
                }

constr_Channing(int x)
    {     this(77,55);
System.out.println("x="+x);
System.out.println("IN 1 parameter constructor");
    }
constr_Channing(int a, int b)
                {
                System.out.println(" a= "+ a);
                System.out.println(" b= "+ b);
```

```
                System.out.println("IN 2 parameter constructor");
                 }
                public static void main(String arg[])
                {
                constr_Channing object = new constr_Channing();
                        constr_Channing object2 = new constr_Channing(3,5);
                        constr_Channing object3 = new constr_Channing();
                }
}

/*

C:\JAVA Programs>java constr_Channing
 a= 77
 b= 55
IN 2 parameter constructor
x=10
IN 1 parameter constructor
IN default constructor
 a= 3
 b= 5
IN 2 parameter constructor
 a= 77
 b= 55
IN 2 parameter constructor
x=10
IN 1 parameter constructor
IN default constructor
*/
```

Interface	Abstract Class
1. If we don't know anything about implementation just we have requirement specification then we should go for interface.	1. If we are talking about implementation but not completely (partial implementation) then we should go for Abstract class.
2. Inside Interface every method is always public and abstract whether we are declaring or not. Hence interface is also considered as 100% pure Abstract class.	2. Every method present in Abstract class need not be public and Abstract. In addition to abstract methods we can take concrete methods also.
3. We can't declare interface method with the following modifiers. Public ---> private, protected, Abstract ---> final, static, synchronized, native, strictfp	3. There are no restrictions on Abstract class method modfiers.
4. Every variable present inside interface is always public, static and final whether we are declaring or not.	4. The variables present inside Abstract class need not be public static and final.
5. We can't declare interface variables with the following modifiers. private, protected, transient, volatile.	5. There are no restrictions on Abstract class variable modifiers.
6. For Interface variables compulsory we should perform initialization at the time of declaration otherwise we will get compile time error.	6. For Abstract class variables it is not required to perform initialization at the time of declaration.
7. Inside interface we can't declare instance and static blocks. Otherwise we will get compile time error.	7. Inside Abstract class we can declare instance and static blocks.
8. Inside interface we can't declare constructors.	8. Inside Abstract class we can declare constructor, which will be executed at the time of child object creation.