Ch-1(Basic Classes and objects(part-2))

OOP concepts

Creating and using class with members

constructor

Finalize() method

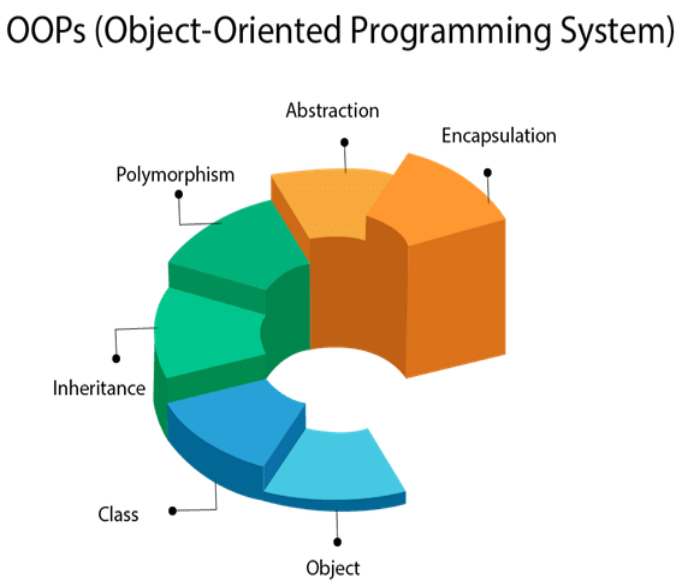
Static and non-static members

Overloading

Var args ,IIB in java

OOP concepts

- Object means a real-world entity such as a pen, chair, table, computer, watch, etc.
- Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:
- Code reusability:
- Inheritance
- polymorphism
- Security:
- abstraction
- encapsulation



Object

- Any entity that has state and behaviour is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.

- An Object can be defined as an instance of a class.

- An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.

class

- Collection of objects is called class. It is a logical entity.
- A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

Inheritance

- When one object acquires all the properties and behaviours of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

- If one task is performed in different ways, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.
- In Java, we use method overloading and method overriding to achieve polymorphism.
- Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.

Abstraction

- Hiding internal details and showing functionality is known as abstraction. For example phone call, we don't know the internal processing.
- In Java, we use abstract class and interface to achieve abstraction.

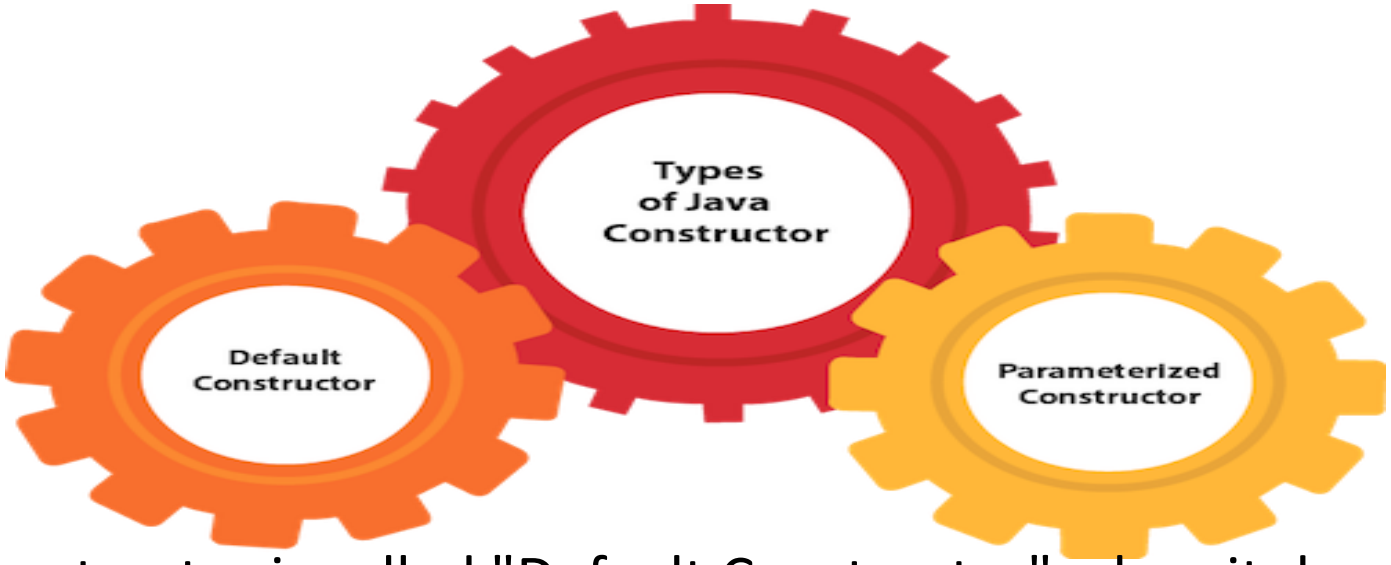
Encapsulation

- Binding (or wrapping) code and data together into a single unit are known as encapsulation. For example, a capsule, it is wrapped with different medicines.
- A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Constructors

- In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.
- It is a special type of method which is used to initialize the object.
- Every time an object is created using the new() keyword, at least one constructor is called.
- It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.
- 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



A constructor is called "Default Constructor" when it doesn't have any parameter.

Syntax: <class_name>(){}

A constructor which has a specific number of parameters is called a "parameterized constructor".

Why use the parameterized constructor?

The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Constructor Overloading in Java

- In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

- Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task.
- They are differentiated by the compiler by the number of parameters in the list and their types.

```
public class Colleges
{
        String collegeId;
            //Colleges(){}
           Colleges(String clg123)
           {
              collegeid=cld123;
           }
 public static void main(String[] args)
           {
           Colleges clg = new Colleges();
                    //this can't create colleges constructor now.
           //System.out.println(clg.collegeId);
           }
 }
```

Java Object finalize() Method

- Finalize() is the method of Object class(The Object class is the parent class of all the classes in java by default). This method is called just before an object is garbage collected. finalize() method overrides to dispose system resources, perform clean-up activities and minimize memory leaks.

- Syntax:

  protected void finalize() throws Throwable

- Throw

  Throwable - the Exception is raised by this method

```
public class JavafinalizeExample1
{
    public static void main(String[] args)
   {
       JavafinalizeExample1 obj = new JavafinalizeExample1();
       System.out.println(obj.hashCode());
       obj = null;
         System.gc(); // calling garbage collector
       System.out.println("end of garbage collection");

   }
   protected void finalize()
   {
       System.out.println("finalize method called");
   }
}
```

Output:

```
705927765
end of garbage collection
finalize method called
```

Static and Non-static Members

Static (Associated, Memory Allocation)

- Static members (variables and methods) are associated with the class itself rather than with individual instances.
- Static members are allocated memory only once, at the time of class loading. They are shared among all instances of the class.

Non-Static

- Non-static members are specific to each instance of a class, as they are tied to objects created from the class.
- Non-static members have memory allocated separately for each instance of the class. Each object has its own copy of non-static members.

,

Accessing ,Initialization

- Static members can be accessed directly using the class name followed by the member name (e.g., ClassName.memberName). They are accessible from anywhere within the program.
- Static members are initialized when the class is loaded into memory, typically during program startup. Initialization happens only once.
- Non-static members are accessed using an object reference followed by the member name (e.g., objectReference.memberName). They are specific to a particular instance of the class.
- Non-static members are initialized when each instance of the class is created, usually using the new keyword. Initialization occurs separately for each object.

Scope, Access to Members

- Static members have a global scope and can be accessed from anywhere within the program, even without creating an instance of the class.
- Static members can only access other static members within the same class. They cannot directly access non-static members.
- Non-static members have a local scope and can be accessed only through an instance of the class. They are not accessible without creating an object.
- Non-static members can access both static and non-static members within the same class. They have direct access to all members.

Method Overloading in Java

- If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.
- If we have to perform only one operation, having same name of the methods increases the readability of the program.

- Advantage of method overloading
- Method overloading *increases the readability of the program*.

- Different ways to overload the method
- There are two ways to overload the method in java
- By changing number of arguments
- By changing the data type

Note: In Java, Method Overloading is not possible by changing the return type of the method only.

Method Overloading: changing no. of arguments

```
class TestOverloading1
{
 static int add(int a,int b)      //method is static so,don't create instance ,2-arg.
     {     return a+b;     }
 int add(int a,int b,int c)    //3-argument
        {    return a+b+c;      }
public static void main(String[] args)
{
    TestOverloading1 a=new TestOverloading1();
     System.out.println (TestOverloading1.add(11,11));
     System.out.println (a.add(11,11,11));
}   }
```

Method Overloading: changing data type of arguments

```
class Adder
{
static int add(int a, int b)
{return a+b;}
static double add(double a, double b)
{return a+b;}
}
class TestOverloading2{
public static void main(String[] args)
{
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
} }
```

In java, method overloading is not possible by changing the return type of the method only because of ambiguity. Let's see how ambiguity may occur:

```
class Adder
{
    static int add(int a,int b)     {return a+b;}
    static double add(int a,int b){return a+b;}
}
class TestOverloading3
{
public static void main(String[] args)
{
        System.out.println(Adder.add(11,11));     //ambiguity
} }
```

Output:
Compile Time Error: method add(int,int) is already defined in class Adder

Can we overload java main() method?
Yes, by method overloading. You can have any number of main methods in a class by method overloading.
But JVM calls main() method which receives string array as arguments only. Let's see the simple example:

```
class TestOverloading4
{
public static void main(String[] args)
{
        System.out.println("main with String[]");
}
public static void main(String args)
{
        System.out.println("main with String");
}
public static void main()
{
        System.out.println("main without args");
} }
```

Variable Argument (Varargs):

- The varrags allows the method to accept zero or muliple arguments.
- Before varargs either we use overloaded method or take an array as the method parameter but it was not considered good because it leads to the maintenance problem. If we don't know how many argument we will have to pass in the method, varargs is the better approach.

Advantage of Varargs:

- We don't have to provide overloaded methods so less code.

Syntax of varargs:

- The varargs uses ellipsis i.e. three dots after the data type.

```
return_type method_name(data_type... variableName){}
```

```
class VarargsExample2
{
  static void display(String... values)
{
 System.out.println("display method invoked ");
        for(String s:values)
       {
         System.out.print(s);
        }
 }
  public static void main(String args[])
{
         display();          //zero argument
       display("hello");  //one argument
       display("my","name","is","varargs");       //four arguments
 }  }
```

Output:
display method invoked
display method invoked
hellodisplay method invoked
my nameisvarargs

Rules for varargs:

1. There can be only one variable argument in the method.
2. Variable argument (varargs) must be the last argument.

- **void** method(String... a, **int**... b){} //Compile time error
- **void** method(**int**... a, String b){} //Compile time error

```
class VarargsExample3
{

 static void display(int num, String... values)
{
  System.out.println("number is "+num);
  for(String s:values)
{
   System.out.println(s);
  }
 }

 public static void main(String args[])
{

 display(500,"hello");//one argument
 display(1000,"my","name","is","varargs");  //four arguments
 }
}
```

Instance initializer block

- Instance Initializer block is used to initialize the instance data member. It run each time when object of the class is created.
- The initialization of the instance variable can be done directly but there can be performed extra operations while initializing the instance variable in the instance initializer block.

- Que) What is the use of instance initializer block while we can directly assign a value in instance data member? For example:

```
class Bike
{   int speed=100; }
```

- Why use instance initializer block?

  Suppose I have to perform some operations while assigning value to instance data member e.g. a for loop to fill a complex array or error handling etc.

What is invoked first, instance initializer block or constructor?

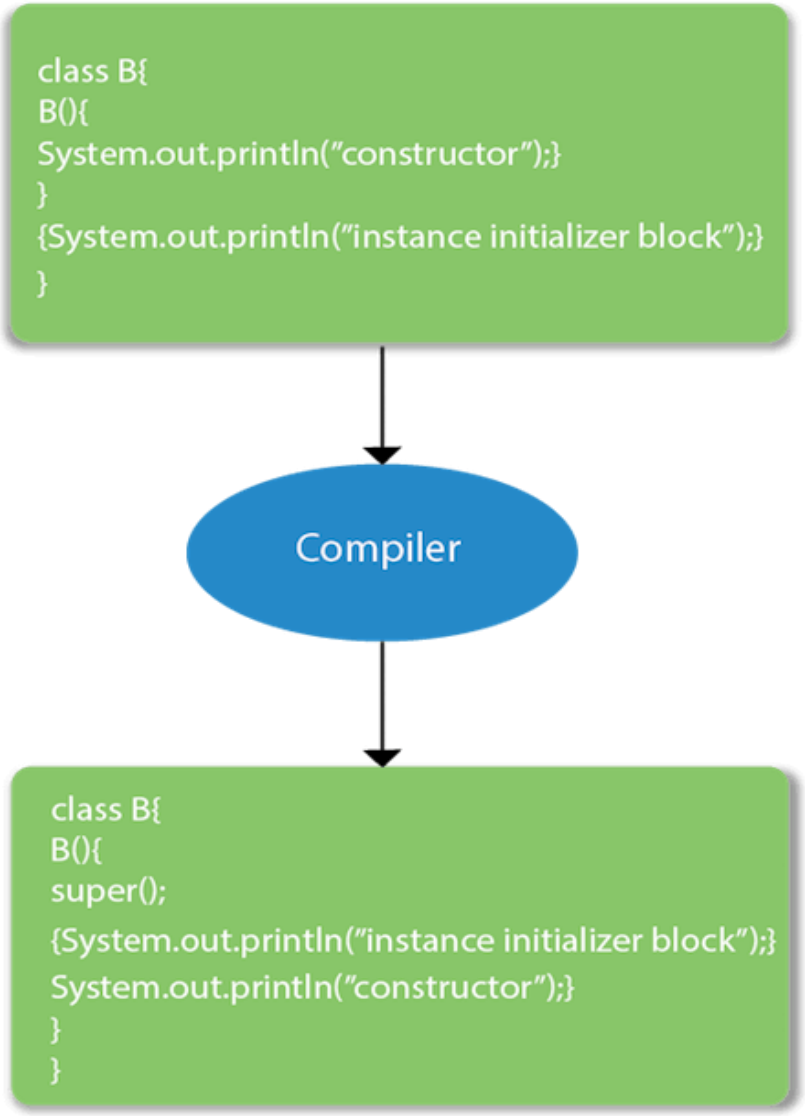
```
class Bike
{
  int speed;
  Bike()
{       System.out.println("constructor is invoked");    }
  {    System.out.println("instance initializer block invoked"); }
                                                    //instance - initializer block
       public static void main(String args[])
{
    Bike b1=new Bike();
    Bike b2=new Bike();
    }
}
```

Output:

```
instance initializer block invoked
 constructor is invoked
instance initializer block invoked
constructor is invoked
```

In the above example, it seems that instance initializer block is firstly invoked but NO. Instance intializer block is invoked at the time of object creation.
The java compiler copies the instance initializer block in the constructor after the first statement super().
So firstly, constructor is invoked.

Note: The java compiler copies the code of instance initializer block in every constructor.



Rules for instance initializer block :

- There are mainly three rules for the instance initializer block. They are as follows:

1. The instance initializer block is created when instance of the class is created.
2. The instance initializer block is invoked after the parent class constructor is invoked (i.e. after super() constructor call).
3. The instance initializer block comes in the order in which they appear.

- There are three places in java where you can perform operations:
- method
- constructor
- block

```
class A    {
A()
{    System.out.println("parent class constructor invoked");
} }
  class B3 extends A
{    B3()
{    super();
     System.out.println("child class constructor invoked");
}
B3(int a)
{     super();
     System.out.println("child class constructor invoked "+a);
}
  {   System.out.println("instance initializer block is invoked");}
  public static void main(String args[]){
B3 b1=new B3();
B3 b2=new B3(10);
} }
```

Output:

parent class constructor invoked I
nstance initializer block is invoked
child class constructor invoked
parent class constructor invoked
 instance initializer block is invoked
child class constructor invoked 10