OOPs In Java

1.Introduction to OOPs

OOPs Defination

Real World Class modelling and why it is needed

Example of OOPs in the industry

2.Classes, Objects and Access Modifiers

Access Specifiers

Class

Object

Interview Questions

3.Constructor and Destructor

Constructor

Constructor Overloading

Destructor

Interview Questions

4.Special Keywords

Static keyword

Final keyword

Super keyword

this keyword

Interview Questions

5.Pillors of OOPs

Encapsulation

Inheritance

Polymorphism

Abstraction

Interface

Interview Questions

OOPs Definition

Definition

Object-Oriented Programming is a programming style that relates the programming to real-world entities/models. Object-Oriented programming is associated with the concept of class and objects. Oops tries to map the code/instructions with the real world, making the code short, simple, and easier to understand. Popular object-oriented programming languages are java, python, c++, etc. The main objective of OOPs is to provide certain features like data security, reusability and ensure a higher level of accuracy.

Nowadays, protecting data stored in a computer is a difficult task, particularly in a society that has become increasingly dependent on computer systems. This paper focuses on a new model proposed for data security and explores the extension of this model to object-oriented programming systems. With the help of oops concepts like inheritance, polymorphism, encapsulation, abstraction, it becomes easier and safe to model the real-world entities into the code/programs.

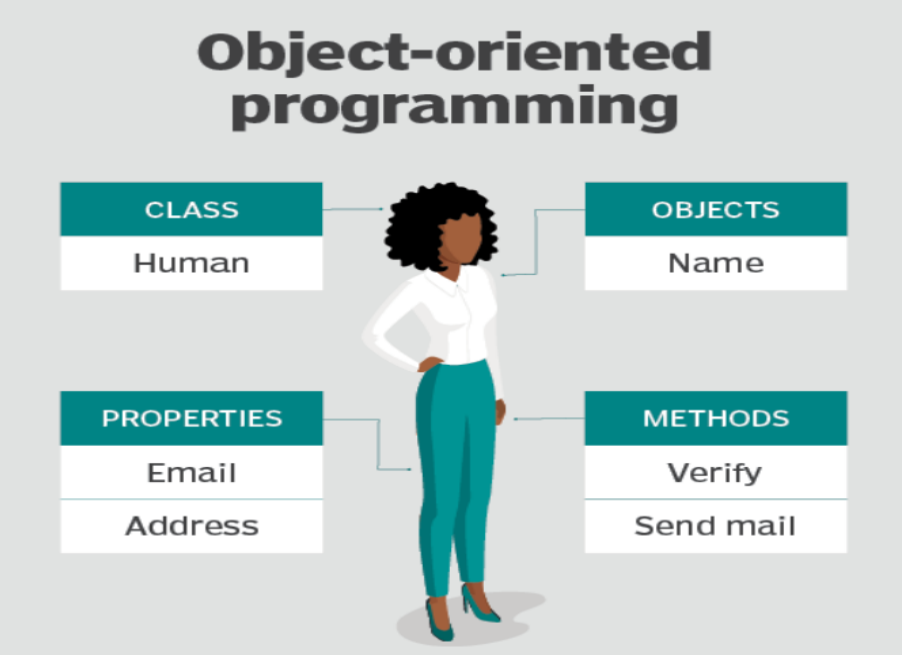
Advantages of OOPs:-

- fast, easier to execute, maintain, modify and debug
- provides a clear structure for the programs
- helps to keep the Java code DRY "Don't Repeat Yourself."
- OOP makes it possible to create complete reusable applications with less code and shorter development time

Structure of object-oriented programming

The structure of object-oriented programming like java includes the following things:

- **Classes** are user-defined data types that act as a blueprint for creating individual objects, methods, and properties.
- **Objects** are instances of a class. Objects can correspond to real-world entities like a human.
- **Methods** are functions that describe the behaviors of an object. Usually, programmers use methods for code reusability or keeping functionality encapsulated for security purposes. eg, verify, Sendmail, talk, etc
- **Attributes** represent the current state of an object. Objects will have data stored in the attributes field. e.g., color, score, email, etc



What Does a Class Mean?

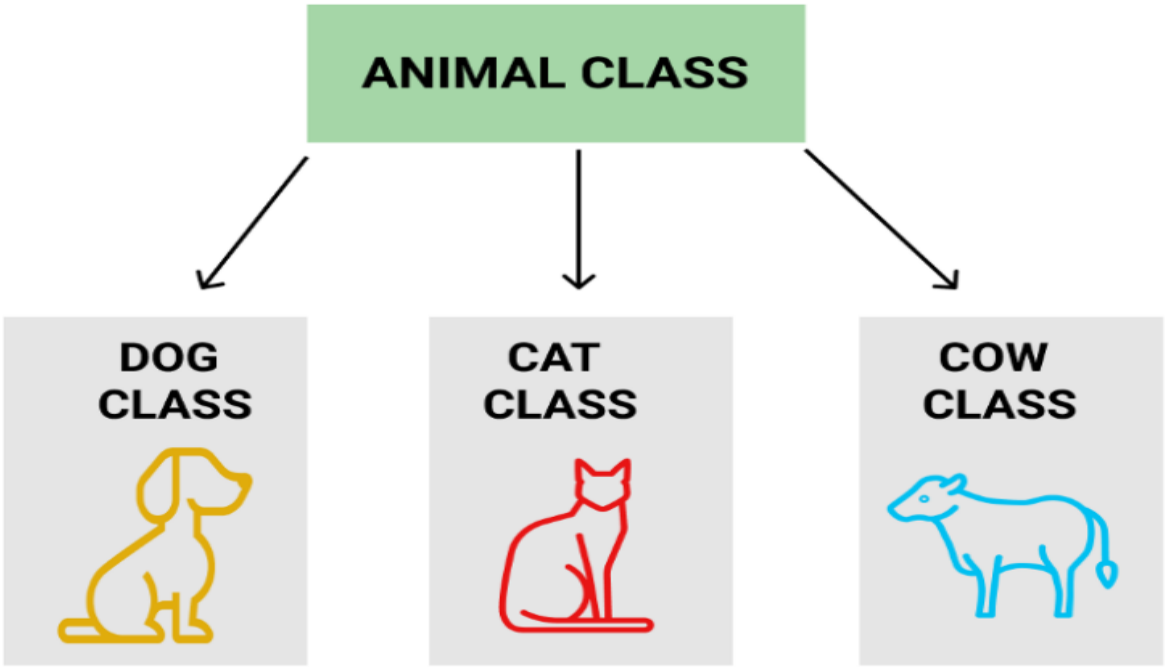
- In the context of Java, class is a template used to create objects and define objects' data types/properties and the methods.

- Classes are like categories, and objects are like items within each category.
- All the class objects should have the basic properties of the class.
- Main Core properties include the actual properties/attributes and methods used by the object.

Class explanation with real-world entities

For example, a specific Dog is an object of the "Dogs" class in this real world. All Dogs in the world share some characteristics from the same template. Being an animal, they have a tail and are the faithful of all animals.

In Java, the "Dogs" class is the blueprint from which all individual Dogs can be generated that includes all Dog's characteristics, such as race, fur color, tail length, eyes shape, etc. So, for example, you cannot create a car from the cat class because a car must have specific characteristics such as:- having an engine, windows, and lights — and none of these objects' properties can be found in the dogs class.



How to create Class in Java

A class declaration contains the following parts:

- Modifiers (optional otherwise default modifier is considered by default)

- Class keyword followed by the Class name
- superclass with appropriate keyword extends (the name of a class' parent, if available)
- appropriate Keywords depending on whether the class implements one or more interfaces or extends from a superclass (if any)
- The class body should be enclosed within curly brackets {}

Syntax:-

```
Access-specifier/modifier  class  <classname> {
        //properties
        //methods
 }
```

What Does Java Object Mean?

- Java object is an instance of a Java class. Each object has a unique identity, a behavior, and a state of it.
- We store the state of an object in fields(variables), while methods show the object's behavior. JVM creates the Objects at runtime from templates, also known as blueprints/classes.
- In Java, we create an object using the keyword "new."
- Memory allocation takes place when the object is created
- The new keyword is used to allocate memory at runtime. All objects get memory in the Heap memory area.

Features used to characterize an object:

- **State:** represents the properties of the object.
- **Behavior:** represents the functionality of an object such as walking, talking, running, etc.
- **Identity:** An object identity is implemented by a unique ID. The value of the ID is hidden to the external user. It is only used internally by the JVM to identify each object uniquely.

Object explanation with real-world entities

Java objects are pretty similar to what we come across in the real world like A Dog, a lighter, a cat, or vehicles are all objects.

For example, a dog's state includes its color, size, gender, and age, while its behavior is sleeping, barking, walking around like a security guard at 3 a.m.

State	Name Color Breed Hungry
Behavior	Barking Fetching Wagging Tail

How to Create Objects in Java

Using the **new** keyword is the best way to create an instance of the class. When we create an object by using the new keyword, it allocates memory (heap) for the **object** and it also returns the **reference** of that object.

Syntax:-

```
ClassName object = new ClassName();
```

Let's create a program to get familiar with classes and objects

```java
public class codingninjas
{


  void show ()
  {

    System.out.println ("Welcome to codingninjas");

  }

  public static void main (String[]args)
  {

    //creating an object using new keyword
    codingninjas obj = new codingninjas ();

    //invoking method using the object
    obj.show ();

  }
}
```

```
Welcome to codingninjas


...Program finished with exit code 0
```

Features of OOPs:-

Four major object-oriented programming features make them different from non-OOP languages:

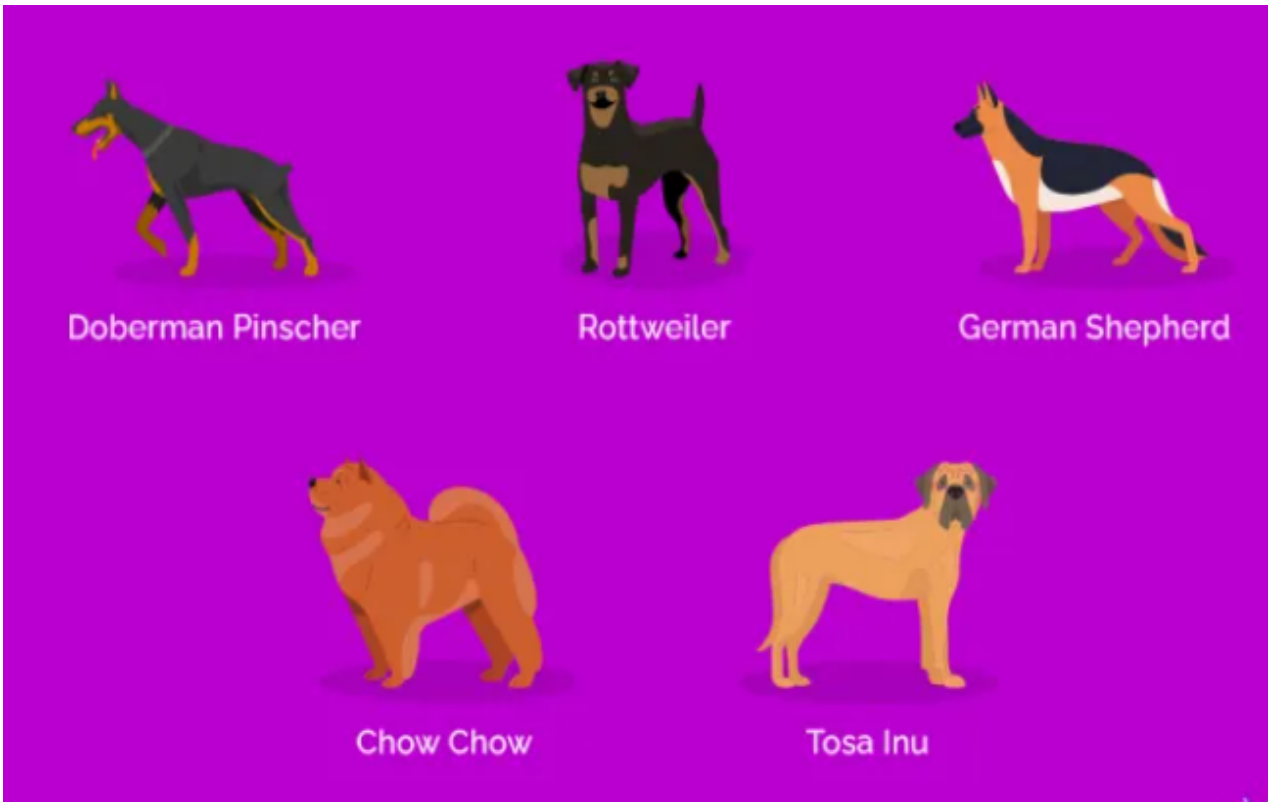
- **Abstraction** is the property by virtue of which only the essential details are displayed to the user.
- **Inheritance** allows you to create class hierarchies, where a base class gives its behavior and attributes to a derived class.
- **Polymorphism** ensures that it will execute the proper method based on the calling object's type.
- **Encapsulation** allows you to control access to your object's state while making it easier to maintain or change your implementation at a later date.

Real-world class modeling

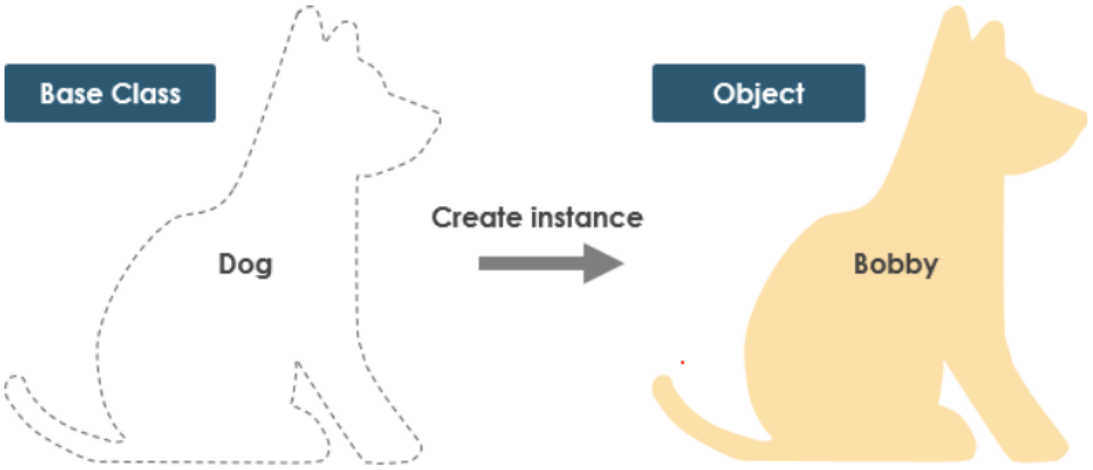
What is Real-world class modeling?

Real-world Object-oriented class modeling designs and prepares the model's code and structure similar to the real-world entity. During the programming phase of construction, the modeling is implemented by using a programming language that supports the object-oriented programming model.

Let us take the example of a dog. In our day to day life, we see dogs of various breeds having different colors, height, length, weight, eye color, etc.;-



So you can see here that different dog breeds are there depending upon the height, color, tail, etc., so we can create a real-world class simulating all the dog species depending upon the properties and methods of the class.



Properties	Methods
Color	Sit
Eye Color	Lay Down
Height	Shake
Length	Come
Weight	

Property Values	Methods
Color: Yellow	Sit
Eye Color: Brown	Lay Down
Height: 17 in	Shake
Length: 35 in	Come
Weight: 24 pounds	

Let's model these real-world entities into the program.

Dog.java

```
class Dog{

    //properties of dog

    String name;
    String color;
    double height;
    double weight;
    double length;
    String eyeColor;

    // methods/functions that the dog performs
    public void  bark(){
        System.out.println(this.name+" bow-wow");
    }

    public void  eat(){
        System.out.println(this.name +" is eating");
    }

}
```

Main.java

```
public class Main
{
    public static void main(String[] args) {
        //create the Doberman object
        Dog Doberman=new Dog();
        Doberman.name="Doberman";
        Doberman.color="brown";
        Doberman.height=3;
        Doberman.length=5;
        Doberman.eyeColor="black";
        Doberman.bark();
        Doberman.eat();
        //create the RottWeiler object
        Dog RottWeiler=new Dog();
        RottWeiler.name="RottWeiler";
        RottWeiler.color="Black";
        RottWeiler.height=3;
        RottWeiler.length=5;
        RottWeiler.eyeColor="brown";
        RottWeiler.bark();
        RottWeiler.eat();
        //create the Germanshepherd object;
        Dog Germanshepherd =new Dog();
        Germanshepherd.name="Germanshepherd";
        Germanshepherd.color="brown";
        Germanshepherd.height=2;
        Germanshepherd.length=4;
        Germanshepherd.eyeColor="lightBrown";
        Germanshepherd.bark();
        Germanshepherd.eat();

        //similarly in the same manner you can create the objects
        // and set the properties of the object
    }
}
```

The output of the above-written program

```
Doberman bow-wow
Doberman is eating
RottWeiler bow-wow
RottWeiler is eating
Germanshepherd bow-wow
Germanshepherd is eating


...Program finished with exit code 0
```

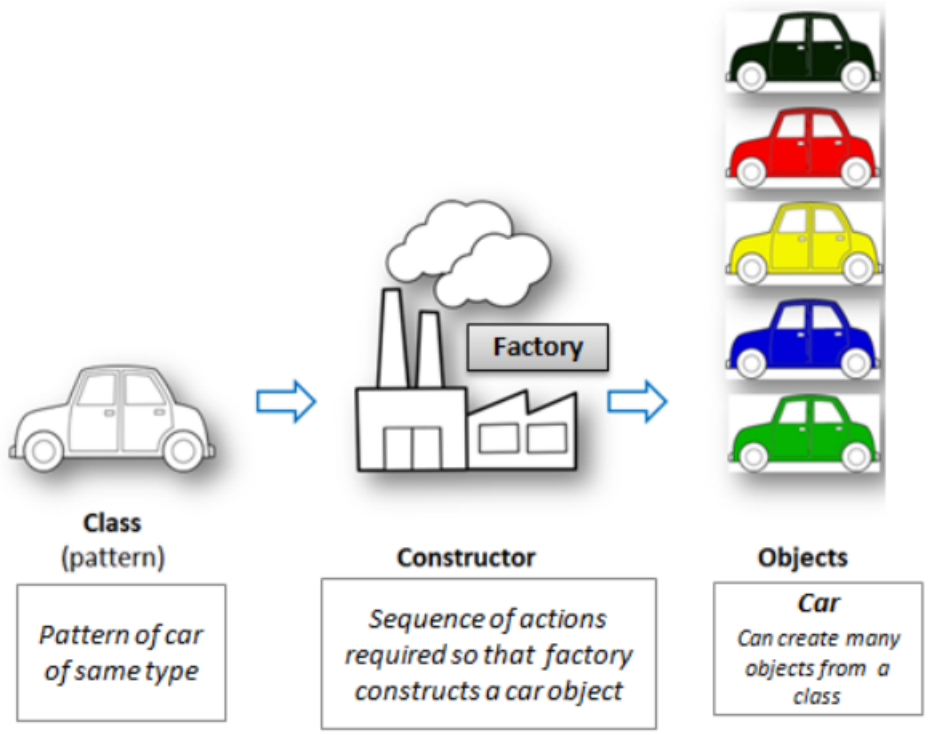
Why do we need Real-world object-oriented Class modeling?

- To make the development and **maintenance** of projects more effortless.
- To provide the feature of **data hiding** that is good for security concerns.
- We can **solve real-world problems** by using object-oriented programming.
- It ensures **code reusability**.
- It lets us write **generic code**: which will work with a range of data, so we don't have to write basic stuff over and over again.

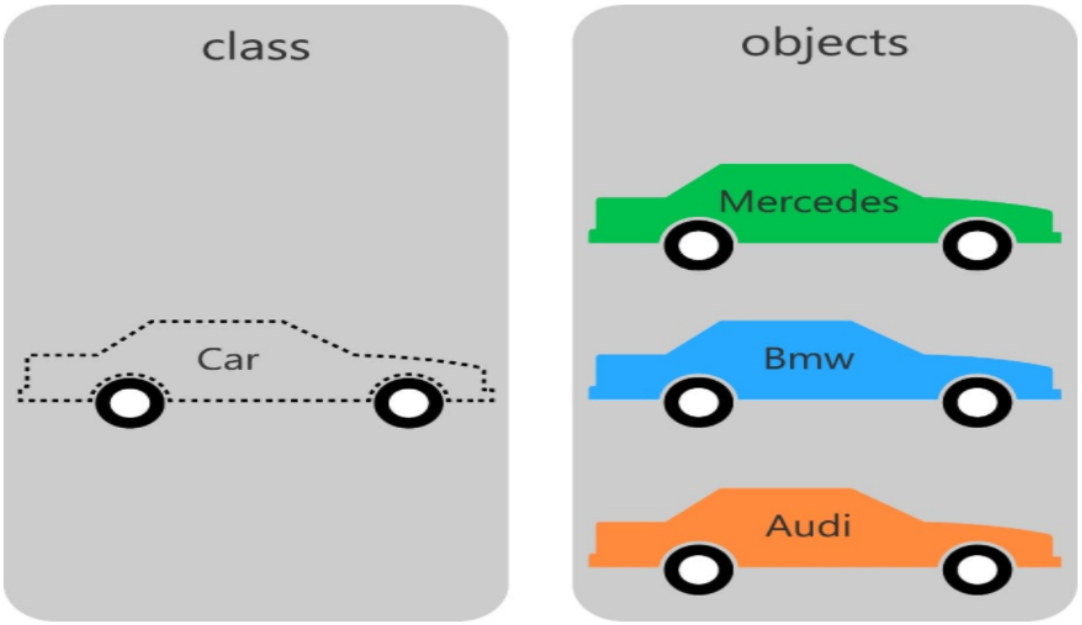
Example of OOPs in the Industry

Real-world Example of OOPs In the industry:-

Let's consider that you are the designer of the cars in the company. You have the sample car design and want to produce different cars with certain modifications in the sample so that something new can be launched in the market.

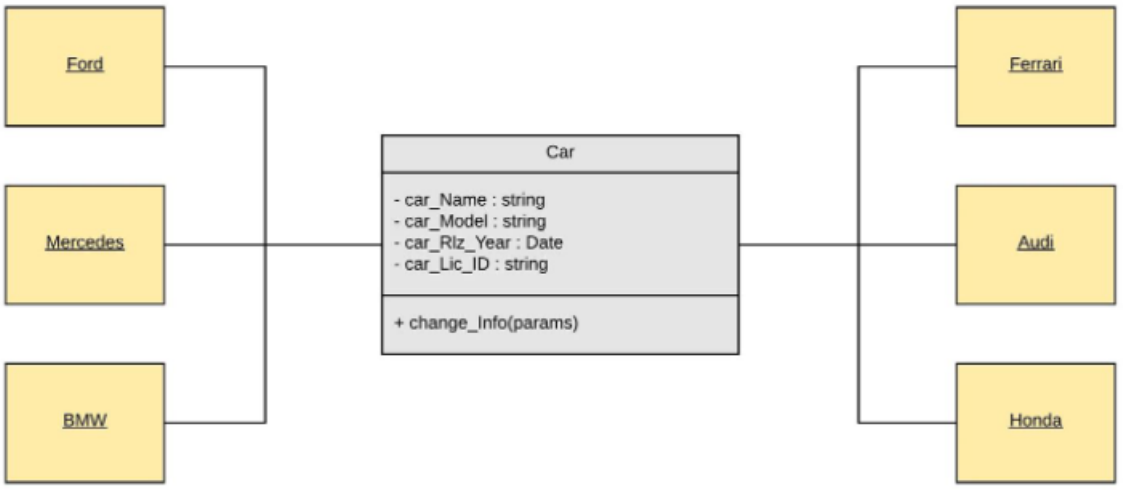


- Here we will make Car class, and it will work as a basic/sample template for other objects.
- We will make car class objects (Ferrari, BMW, and Audi).
- Each Car Object will have its Year of Manufacture(car_Riz_Year), model(car_model), name(car_name), registration Details (car_Lic_ID), etc.;
- you can further add properties if you want accordingly like TopSpeed, price, efficiency, etc.

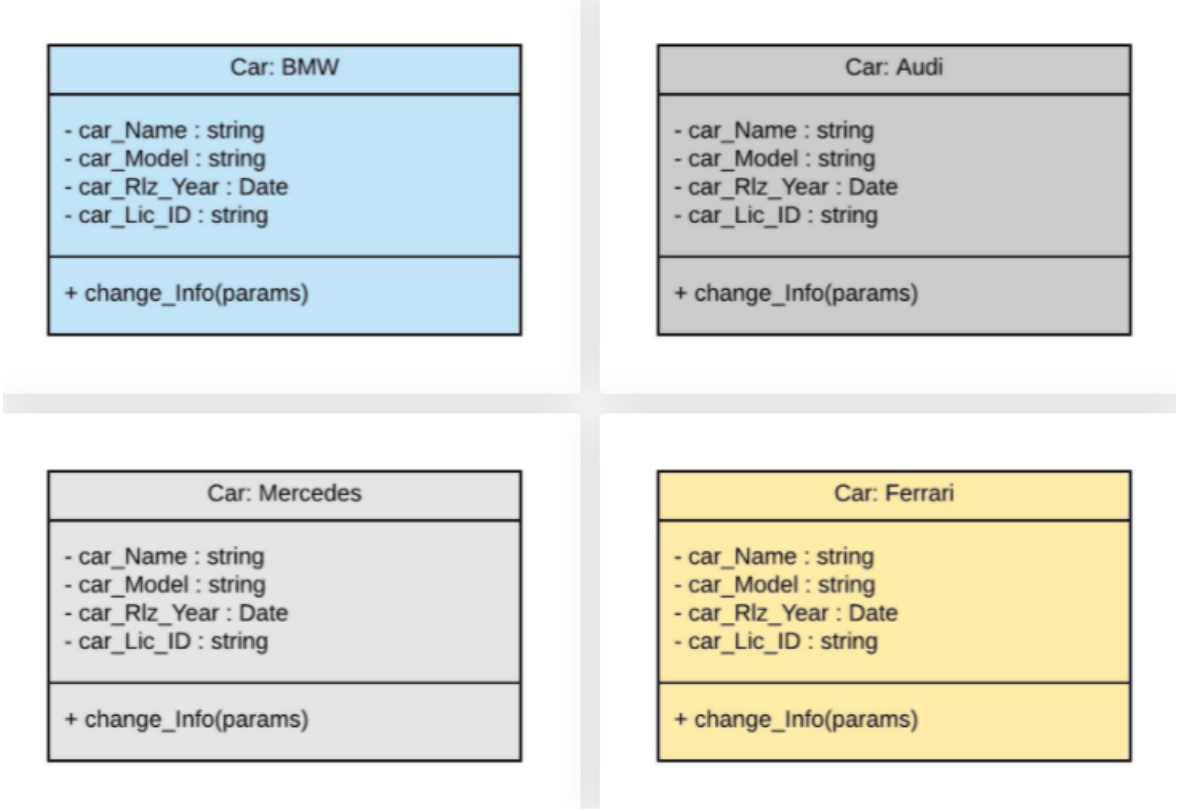


Here, car class would allow the programmer to store similar information unique to each car (different models, name, year of manufacture, top speeds, etc.) and associate the appropriate information.

Understanding example using flowchart:-



Instances of a specific class can also be represented as follows :



Access Specifiers

we will cover the following

- Private
- Public
- Protected
- Default

Overview

In Java, we can impose access restrictions on different data members and member functions. The restrictions are specified through access modifiers.

Access modifiers are tags we can associate with each member to define which parts of the program can access it directly.

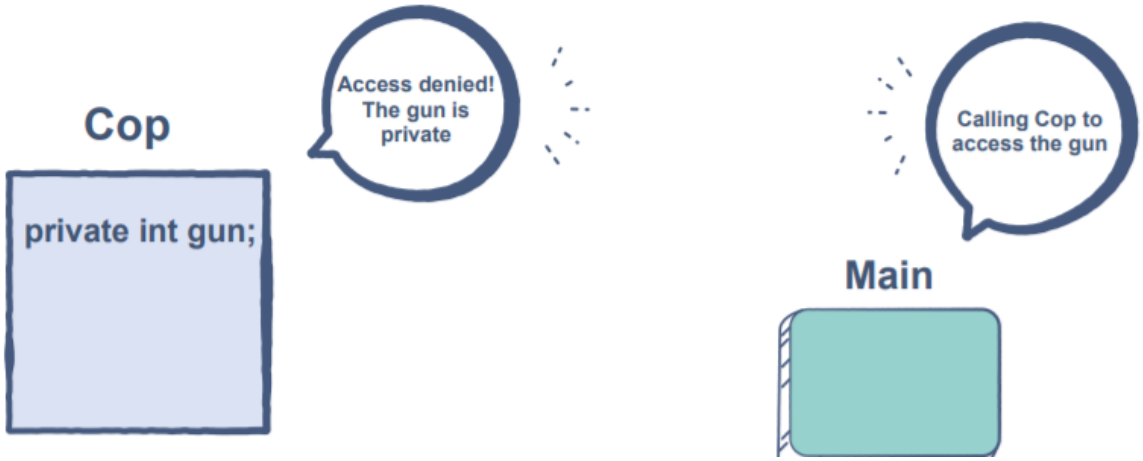
There are three types of access modifiers. Let's take a look at them one by one.

1. Private
2. Public
3. Protected
4. Default

Private:-

A private member cannot be accessed directly from outside the class. The aim is to keep it hidden from the users and other classes. It is a popular practice to **keep the data members private** since we do not want anyone manipulating our data directly. We can make members private using the keyword private.

The access level of a private modifier is only within the class. It cannot be accessed from outside the class.



When we try to access private members from outside the class, then there will be a compile-time error.

```
class Cop {

  private int gun; // We have explicitly defined that the variable is private
  // ...
}
```

Public:-

The public has the widest scope among all the modifiers. This tag indicates that the members can be directly accessed by anything anywhere, either in the same package, outside the package, inside another class, etc.

Here we can see that the getGun method is public, so we can call the public method from anywhere so the public method getGun will be called by the main method, and the getGun method has access to the private variable gun.

as the getGun() method and gun variable both are defined in the same class, so the getGun method will access the gun variable, and the getGun() method will be accessed by the main method.

This technique is used in data security, like those who design the program knows how to access those variables. Anyone who is an outsider will not be able to get access to private property. Just assume that you are designing the banking system, and any outsider tries to know the total cash in the bank so he will not be able to do so because he does not know which method he needs to call to fetch the data because the variable cash can't be accessed directly as it is private.

```
class Cop {
  private int gun; // Private variable

  public int getGun(){
    return gun;  // The private variable is directly accessible over here!
  }
}
```

Here we will create the object of Cop class in the main class and call the getGun() method with the help of object c.

```
Cop c = new Cop(); // Object created
c.getGun(); // Can access the gun
c.gun = 0; // This would cause an error since gun is private
```

Protected:-

The protected access modifier is unique. The level of access to the protected members lies somewhere between public and private. In Java, the protected access modifier behaves like default. But the primary use of the protected tag can be found when we use inheritance, we will cover inheritance in detail in the upcoming section of the course, as of now the protected data members can be accessed inside a Java package. However, outside the package, they can only be referred to through an inherited class.

Cop.java

```
package justice;

public class Cop {
  private int gun;
  public int getGun(){
    return gun;
  }
  protected void fire(){
    System.out.println("shoot!")
  }
}
```

Thief.java

```
package crime;
import justice.*;

class Thief{
  public static void main(String args[]){
   Cop obj = new Cop();
   Cop.fire(); //Compile Time Error
  }
}
```

Class Thief will fall into a compile-time error because it is trying to access the method fire(), and we can see that it is defined in a different package.

Default:-

If we don't use any modifier, then it is treated as the **default** access modifier. default is accessible only within the same package. It cannot be accessed from another package. It provides more accessibility than private. But, it is restrictive than protected and public

Tabular demonstration of all the access modifier/specifiers

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

CLASS

we will cover the following

- Declaration
- Implementation of Car Class
- Creating a Class and its object

1.1 Declaration

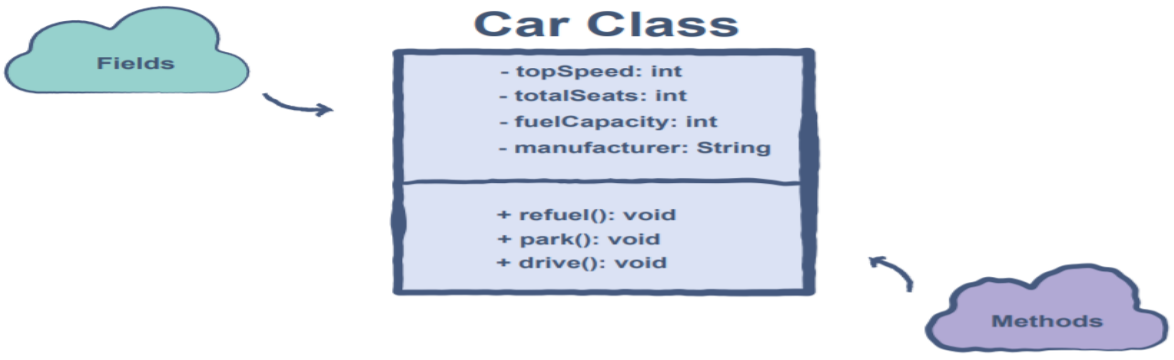
we define classes in the following way:

```
class ClassName {

    // Class name /* All member variables and methods*/

}
```

The class command tells the compiler that we are creating our custom class. All the members of the class will be defined within the class scope.

1.3 implementation of car class

Let's implement the class car below:



```
class Car { // Class name

    // Class Data members
    int topSpeed;
    int totalSeats;
    int fuelCapacity;
    String manufacturer;

    // Class Methods

    void refuel();
    void park();
    void drive();

 }
```

1.3 Creating a Class Object

The name of the class, car, will be used to create an instance of the class car in our main program. We can create an object of a class by using the keyword new :

```
// Class name ...
 class ClassName {

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

         car obj = new car(); // car class  object

       }
}
```

Object

we will cover the following

- Object
- Real-world explanation
- How to create objects?
- constructor
- Implementation

Object:-

- In Java object is an instance of a Java class. Each object has a unique identity, a behavior, and a state of it.

- We store the state of an object in fields(variables), while methods show the object's behavior. JVM creates the Objects at runtime from templates, also known as blueprints/classes.

- In Java, we create an object using the keyword "new."

- Memory allocation takes place when the object is created

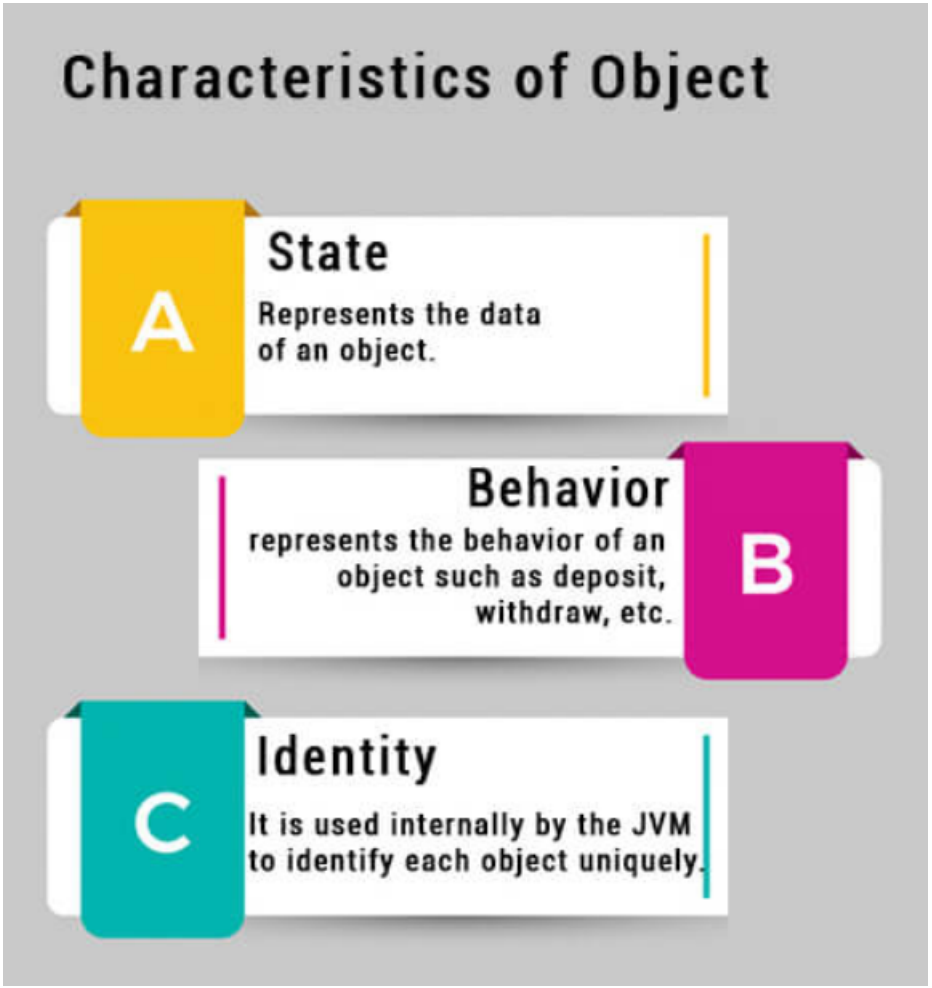
- The new keyword is used to allocate memory at runtime. All objects get memory in the Heap memory area.

Features used to characterize an object:

- **State:** represents the properties of the object.

- **Behavior:** represents the functionality of an object such as walking, talking, running, etc.

- **Identity:** An object identity is implemented by a unique ID. The value of the ID is hidden to the external user. It is only used internally by the JVM to identify each object uniquely.



Object explanation with real-world entities

Java objects are pretty similar to what we come across in the real world like A Dog, a lighter, a cat, or vehicles are all objects.

For example, a dog's state includes its color, size, gender, and age, while its behavior is sleeping, barking, walking around like a security guard at 3 a.m.

How to Create Objects in Java

Using the **new** keyword is the best way to create an instance of the class. When we create an object by using the new keyword, it allocates memory (heap) for the **object** and it also returns the **reference** of that object.

Syntax:-

```
ClassName object = new ClassName();
```

Here you can see ClassName() is looking like a method used here, basically, it is a constructor, after keyword new constructor is called so here we will study what is the constructor;

Constructor:-

constructor is a special method because it does not have a return type. We do not even need to write void as the return type. It is a good practice to declare/define it as the first member method. and its name should be the same as the name of the class.

It is called a constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because the java compiler creates a default constructor if your class doesn't have any.

It is a special type of method which is used to initialize the object.

Implementation:-

Let's create a program to get familiar with classes and objects

```
1  public class codingninjas
2  {
3
4
5    void show ()
6    {
7
8      System.out.println ("Welcome to codingninjas");
9
10   }
11
12   public static void main (String[]args)
13   {
14
15     //creating an object using new keyword
16     codingninjas obj = new codingninjas ();
17
18     //invoking method using the object
19     obj.show ();
20
21   }
22 }
```

```
Welcome to codingninjas


...Program finished with exit code 0
```

Interview Questions

1. Is java a fully object-oriented programming language?

Java is not a fully object-oriented programming language because it supports primitive data types like - int, byte, short, long, etc., which are not object-oriented and, of course, are the opposite of oops.

2. What are the advantages of packages in java?

There are various advantages of defining packages in Java.

- Packages avoid name clashes.
- The Package provides easier access control.
- We can also have the hidden classes that are not visible outside and used by the package.
- It is easier to locate the related classes.

3. What happens if you don't define a constructor in your class. Can we still create the object of that class?

Yes, we can create that class's object because the compiler automatically defines an empty, default constructor inside the class, which remains hidden to the programmer/user/outside world.

4. Why is OOPs so popular?

Oops is so popular because it helps in writing a complex piece of code easily, and it also allows users to handle and maintain them easily. With OOPs, the code's readability, understandability, and maintainability increase multifold.

5. What are the differences between the class and the object?

Class	Object
Class is a data type	An object is an instance of the class.
No memory is allocated when a class is declared.	Memory is allocated as soon as an object is created.
It generates objects	It gives life to the class
Class is a logical entity.	An object is a physical entity.
A class can only be declared once.	Objects can be created many times as per requirement.

6. What are the differences between the constructor and the method?

	Constructor	Method
Name	Constructor's name must be same as the name of the class.	Method's name can be anything.
Return Type	Constructor doesn't have a return type.	Method must have a return type.
Call	Constructor is invoked implicitly by the system.	Method is invoked by the programmer.
Main Job	Constructor can be used to initialize an object.	Method consists of Java code to be executed.
Overload	Constructor can be Overload.	Method also can be overload.

Constructor

We will cover the following

- What is a constructor?
- Default constructor
- Parameterized constructor
- Java Copy Constructor
- Diff b/w constructor and method

What is a Constructor?

It is a special method that is used to initialize the object when an object of a class is created in the program. As the name suggests, the constructor is used to construct the object of a class. It is called when an instance of the class is created.

- A constructor's name must be exactly the same as the name of its class.
- The constructor is a special method because it does not have a return type. We do not even need to write void as the return type.
- The purpose of a Java constructor is to initialize the newly created object before it is used.
- Every time an object is created using the new() keyword, at least one constructor is called.

There are two types of the constructors

- Default constructor
- Parameterized constructor

Default constructor

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

Main .java

```java
public class Main
{
    public static void main(String[] args) {
        // we are creating the object of the car class
        car c=new car();
    }
}

// car class
class car{

   car(){System.out.println("car is created");}

}
```

```
car is created


...Program finished with exit code 0
```

Here you can notice that we have created the object of the car class and the car constructor of the car class is called.

Point to be remembered: If there isn't any constructor in a class, the compiler automatically creates a default constructor within the class that will not be visible to us.

Let us assume we have a bike class and we have not created any constructor within the bike class then in that situation compiler will create the bike() constructor automatically that will not be visible in the code.

Main.java

```
public class Main
{
    public static void main(String[] args) {
        // we are creating the object of the Bike class
        Bike b=new Bike();
    }
}

// Bike class
class Bike{


}
```

```
...Program finished with exit code 0
```

Here you can notice that the program is still running as a default constructor will be created by the compiler automatically within the bike class which is not visible in the program.

Purpose of a default constructor:-

Basically, the purpose of the default constructor is to provide the default values to the objects like null, 0, etc. according to the type.

```
public class Main
{
    public static void main(String[] args) {
        //creating objects
        Student s1=new Student();
        Student s2=new Student();
        //displaying values of the object
        s1.display();
        s2.display();
    }
}

class Student{
    int id;
    String name;
    //method to display the value of id and name
    void display()
    {
        System.out.println(id+" "+name);
    }
```

```
0 null
0 null

...Program finished with exit code 0
```

Parameterized constructor:-

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

Purpose of a parameterized constructor:- The parameterized constructor is used to initialize the object with different-different values.

```
 9  public class Main
10  {
11      public static void main(String[] args) {
12          //creating objects
13          Student s1=new Student(12,"shacksham");
14          Student s2=new Student(14,"Deepak");
15          //displaying values of the object
16          s1.display();
17          s2.display();
18      }
19  }
20
21  class Student{
22      int id;
23      String name;
24
25      Student(int stdid,String stdname) {
26          id=stdid;
27          name=stdname;
28      }
29
30      void display(){
31          System.out.println(id+"  "+name);
32      }
33
34  }
```

```
12  shacksham
14  Deepak


...Program finished with exit code 0
```

Java Copy Constructor

There is no copy constructor in the Java language. But, we can copy the values from one object to another object like copy constructor in other programming languages.
There are several ways to copy the values of one object into another object in Java. They are:

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

By constructor

Here we are copying the values from one object to the other using the copy constructor CopyConstructroExample. You can see in the following code

.

Main.java

```
class CopyConstructorExample {
    private double x, y;
    // A normal parameterized constructor
    public CopyConstructorExample(double one, double two) {
        x = one;
        y = two;
    }
    // copy constructor
    CopyConstructorExample(CopyConstructorExample c) {
        System.out.println("Copy constructor called");
        x = c.x;
        y = c.y;
    }
    public String display() {
        return "(" + x + " + " + y + "i)";
    }
}

public class Main {

    public static void main(String[] args) {
        CopyConstructorExample c1 = new CopyConstructorExample(10, 15);

        // Following involves a copy constructor call
        CopyConstructorExample c2 = new CopyConstructorExample(c1);

        // Note that following doesn't involve a copy constructor call as
        // non-primitive variables are just references.
        CopyConstructorExample c3 = c2;

        System.out.println(c2.display()); // display() of c2 is called here
    }
}
```

input

```
Copy constructor called
(10.0 + 15.0i)


...Program finished with exit code 0
```

By assigning the values of one object to another

Here we are copying the values from one object to the other By assigning the values of one object to another. You can see in the following code

Main.java

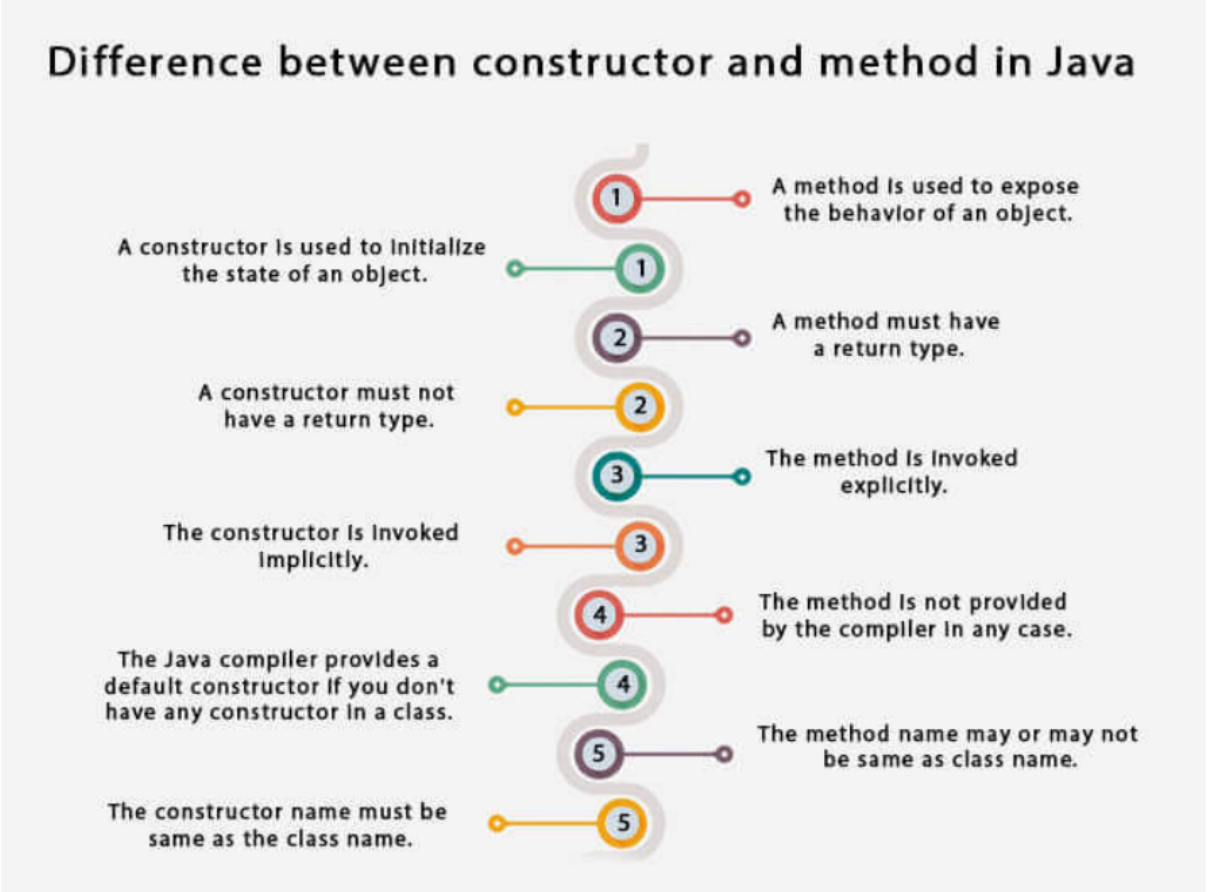
```
class CopyConstructorExample {
    double x, y;
    // A normal parameterized constructor
    public CopyConstructorExample(double one, double two) {
        x = one;
        y = two;
    }
    //default constructor
     public CopyConstructorExample (){

     }

    public String display() {
        return "(" + x + " + " + y + "i)";
    }
}

public class Main {

    public static void main(String[] args) {
        CopyConstructorExample c2 = new CopyConstructorExample(10, 15);


        CopyConstructorExample c3 =new CopyConstructorExample();
        //here we are copying values of one constructor into another.
        c3.x=c2.x;
        c3.y=c2.y;

        System.out.println(c3.display()); // display() of c3 is called here
    }
}
```

input

```
(10.0 + 15.0i)


...Program finished with exit code 0
```

Diff b/w Constructor and method in java

You need to understand that constructor is different than the method in various ways:-



Constructor Overloading

We will cover the following

- What is constructor overloading?
- Examples of constructor overloading
- Key points to remember

What is constructor overloading?

A constructor is just like a method in Java, but it does not have any return type. It can also be overloaded, just like other methods.

Constructor overloading is a tool/technique of having more than one constructor in the class with different no of the parameters. Each constructor performs a different task. The compiler differentiates them by the total number of parameters and their types. Overloading means more than one form. It refers to the use of the same thing for a different purpose.

Main.java

```
public class Main
{
    public static void main(String[] args) {
        //creating objects
        Student s1=new Student(12);
        Student s2=new Student(14,"Deepak");
        //displaying values of the object
        s1.display();
        s2.display();
    }
}

class Student{
    int id;
    String name;
    //constructor-1
    Student(int stdid,String stdname) {
        id=stdid;
        name=stdname;
    }
    //constructor-2
    Student(int stdid){
        id=stdid;
    }
    void display(){
        System.out.println(id+"  "+name);
    }
}
```

```
12  null
14  Deepak

...Program finished with exit code 0
```

You can notice that here that I have created two different constructors.

- Student(int stdid , String stdname)
- Student(int stdid)

I have created two objects s1 and s2 using constructor-1 and constructor-2 respectively. So I have overloaded the constructor. Constructor-2 will not be able to initialize the name of the object so null is printed on the screen.

Example of constructor overloading.

```
public class Employee {
 Employee(){
      System.out.println("Employee Details:");
 }
 Employee(String name){
      System.out.println("Employee name: " +name);
 }
 Employee(String nCompany, int id){
      System.out.println("Company name: " +nCompany);
      System.out.println("Employee id: " +id);
 }
}
public class Myclass {
public static void main(String[] args) {
  Employee emp=new Employee();
  Employee emp2=new Employee("Deep");
  Employee emp3=new Employee("HCL", 12234);
      }
}
```

Fig: Overloaded constructors based on parameter list.

Key Points to remember:-

1. Constructor overloading means having more than one class constructor with different signatures.
2. To compile each constructor must have a different no of parameters.
3. Parameter list consists of order and types of arguments.
4. We cannot have two constructors in a class with the same parameter lists.

Destructor

We will cover the following

What is a Destructor?

In Java, when we create an object of the class. It occupies space in the memory (heap). If we do not delete these objects, they will remain there in the memory and occupy some space that is not useful from programming aspects. In order to resolve this problem, we use the concept of **destructor**.

In this section, we will look over the alternate option to the **destructor in Java**. We will also learn how we can use the **finalize()** method as a destructor.
The **destructor** is just the opposite of the constructor. The constructor is used to initialize java objects, while the destructor is used to destroy the object in order to release the resource and memory occupied by the object.

You need to remember that **there is no concept of destructor in Java**. Instead of the destructor, Java provides an alternative as the garbage collector works the same as the destructor in any other programming language. The garbage collector is a thread/program that runs on the Java Virtual Machine. It automatically destroys/deletes the unused objects (objects no longer used in the program) and frees up the memory space. The programmer need not worry about memory management manually. It can be error-prone, vulnerable, and may lead to a memory leak.

Advantages of Destructor

- It releases the resources occupied by the object.
- No need to call explicitly, it is automatically invoked at the end of the execution of the program.
- it cannot be overloaded because does not accept any parameter.

How does java destructor work?

When we create the object, it occupies the space in the heap memory area. In the program, threads use these objects. If the thread no longer uses the objects, it becomes eligible for deletion/garbage collection. The memory occupied by that object gets available for new objects created in the program. when the garbage collector destroys the object from the heap, the JRE(Java Runtime Environment) calls the finalize() method to close the connections such as network connection and database connection.

How finalize() Method works as a Destructor

It is difficult for the programmers to forcefully execute the garbage collector to destroy the object from the heap. But Java provides an alternative method to do the same thing. The Java Object class (parent class of all the classes in Java) provides the **finalize()** method that works the same as the destructor in other programming languages. The syntax of the finalize() method is given below:

Syntax:

```
1. protected void finalize throws Throwable()
2. {
3. //resources to be close eg. DB connection or network connection
4. }
```

It is not exactly a destructor, but it provides extra security. It ensures the use of external resources in the program, like closing the file, etc., before closing the program. We can call it explicitly by using the method itself or invoking the method predefined in the java **System.runFinalizersOnExit(true)**.

- It is the protected method of the Object class defined in Java.lang package.
- It can be called only once in the program.
- We have to call finalize() method explicitly if we want to override the method in the program.
- The gc() is a method of JVM executed by the Garbage Collector in Java. It gets invoked when the heap memory is full and requires more memory for new objects that are being created in the memory.
- the JVM ignores all the exceptions that occur by the finalize() method except the unchecked exceptions,

Example of Destructor

DestructorExample.java

```java
public class DestructorExample
{

    public static void main (String[]args) {

    DestructorExample de = new DestructorExample ();

    de.finalize ();

    de = null;

    System.gc ();

    System.out.println ("Inside the main() method");

    }

    protected void finalize () {

    System.out.println ("Object is destroyed by the Garbage Collector");

    }
}

```

input

```
Object is destroyed by the Garbage Collector
Inside the main() method
Object is destroyed by the Garbage Collector


...Program finished with exit code 0
```

Interview Questions

Q1. What is a Constructor in Java?

Constructor is just like a method in Java that is used to initialize the state of an object and will be invoked during the time of object creation.

Q2. What are the Rules for defining a constructor?

1. The constructor name should be the same as the class name
2. It cannot contain any return type
3. It can have all Access Modifiers are allowed (private, public, protected, default)
4. It Cannot have any Non Access Modifiers (final, static, abstract, synchronized)
5. No return statement is allowed
6. It can take any number of parameters

Q3. What is a No-arg constructor?

A constructor **without arguments** is called a no-arg constructor. In Java Default constructor is a no-arg constructor.

```
class Demo
{
   public Demo()
   {
      //no-arg constructor
   }
}
```

Q4. Can we have both Default Constructor and Parameterized Constructor in the same class?

Yes, we have both Default Constructor and Parameterized Constructor in the same class.

Q5. What happens if you don't define a constructor in your class. Can we still create the object of that class?

Yes, we can create the object of that class because the compiler defines an empty, default constructor inside the class automatically which remains hidden to the programmer/user/outside world.

Q6. Will the compiler create the Default Constructor when we already have a Constructor defined in the class?

No, the compiler will not create the Default Constructor when we already have a Constructor defined.

Q7. What is the use of Private Constructors in Java?

When we use the **private** keyword for a constructor then an object for the class can only be created **internally** within the class, **no outside class** can create an object for this class. Using this we can **restrict** the caller from creating objects.

```
class ExampleOfPrivateConstructor
{
   /**
    * Private Constructor for preventing object creation
   from the  outside class
   **/
   private ExampleOfPrivateConstructor (){ }

   public void display()
   {
      System.out.println("disp() method called");
   }
}
public class Sample
{
   public static void main(String args[])
   {
      //Creating the object for the Private Constructor class
      ExampleOfPrivateConstructor pc = new ExampleOfPrivateConstructor ();

      pc.display();
   }
}
```

When we will try to run the above code we will be getting the below exception.
Exception in thread "main" java.lang.Error: Unresolved compilation problem:

The constructor ExampleOfPrivateConstructor () is not visible

at Sample.main(Sample.java:21)

Q8. What are the differences between the constructor and the method?

	Constructor	Method
Name	Constructor's name must be same as the name of the class.	Method's name can be anything.
Return Type	Constructor doesn't have a return type.	Method must have a return type.
Call	Constructor is invoked implicitly by the system.	Method is invoked by the programmer.
Main Job	Constructor can be used to initialize an object.	Method consists of Java code to be executed.
Overload	Constructor can be Overload.	Method also can be overload.

Q9. Does the Constructor create the object?

The new operator in Java creates objects. Constructor is the later step in object creation. The constructor's job is to initialize the members after the object has reserved memory for itself.

Static

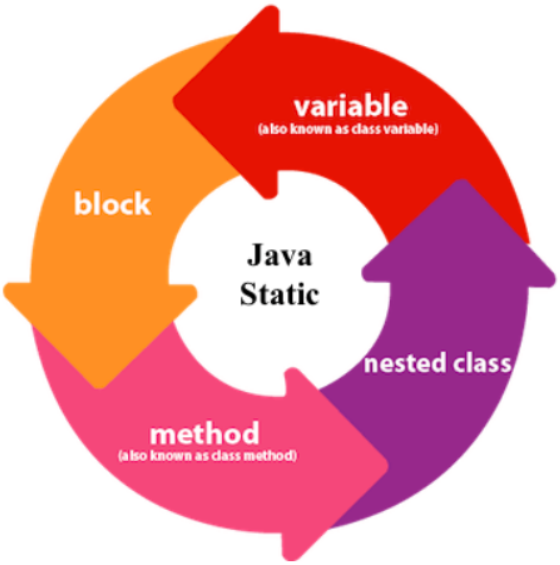
We will cover the following

- Static keyword
- Java static variable
- Java static method
- Java static block

Static keyword

The static keyword is used for memory management in java. The static keyword is used with methods, variables, blocks, and nested classes. Basically, the static keyword belongs to the class than an instance of the class. The static can be:

1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block
4. Nested class



Java static variable

When you use a static keyword with a variable, then it is known as a static variable.

- The static variable will be shared among all the objects of the class (which is not unique for each object),
- Memory is allocated to the static variable only once in the class area at the time of class loading.

Advantages of static variable

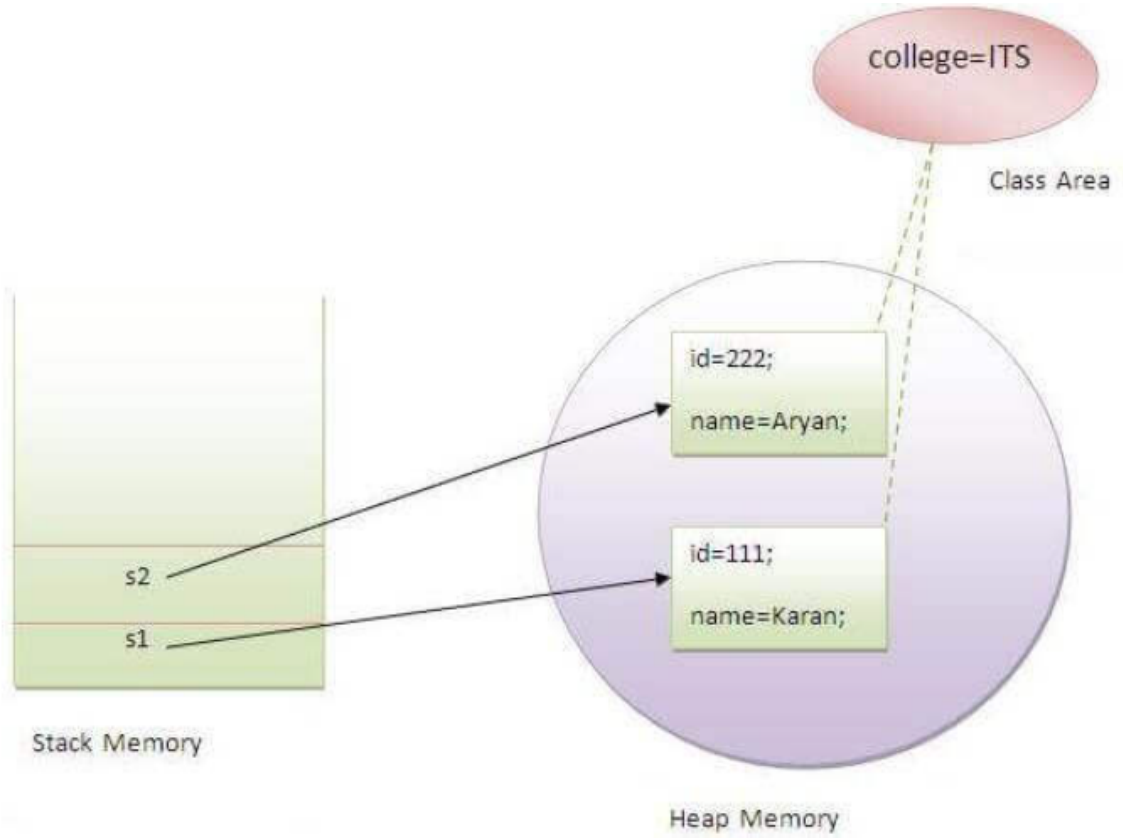
It makes your program **memory efficient** (i.e., it saves memory).

Just go through the code, read the comments, and have a look over the output to get a better insight into the use of static variables.

```java
//Java Program to demonstrate the use of static variable
class Student{
   int rollno;//instance variable
   String name;
   static String college ="ITS";//static variable
   //constructor
   Student(int r, String n){
   rollno = r;
   name = n;
   }
   //method to display the values
   void display (){System.out.println(rollno+" "+name+" "+college);}
}
//main class to show the values of objects
public class Main{
 public static void main(String args[]){
 Student s1 = new Student(111,"Karan");
 Student s2 = new Student(222,"Aryan");
 //we can change the college of all objects by the single line of code
 //Student.college="BBDIT";
 s1.display();
 s2.display();
 s1.college="MIT";
 System.out.println("s1 changed the college name now let us see is it changed for s2 too ?");
 s1.display();
 s2.display();
 System.out.println("yes, if any object change that variable that changes will be reflected to all objects");

 }
}
```

input

```
111 Karan ITS
222 Aryan ITS
s1 changed the college name now let us see is it changed for s2 too ?
111 Karan MIT
222 Aryan MIT
yes, if any object change that variable that changes will be reflected to all objects


...Program finished with exit code 0
```



Java static Method

If you use a static keyword with any method, it is called a static method.

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without creating an instance of a class.
- A static method can access static data members and can change the value of it.

```
class Calculate{
  static int cube(int x){
  return x*x*x;
  }

  public static void main(String args[]){
  int result=Calculate.cube(15);
  System.out.println(result);
  }
}
```

```
3375


...Program finished with exit code 0
```

You can notice in the above example that cube method of the calculate class is called directly using the name of the class because static methods belong to the class rather than the object itself.

Restrictions for the static method

1. The static method cannot use non-static data members or call the non-static method directly.
2. this and super cannot be used in a static context.

Main.java

```
class Main{
 int id=40;//non static

 public static void main(String args[]){
  System.out.println(id);
 }
}
```

input

```
Compilation failed due to following error(s).
Main.java:5: error: non-static variable id cannot be referenced from a static context
  System.out.println(id);
                     ^
1 error
```

You can see that id is a non-static member and the main method is static so the compiler throws a compilation error in this situation.

Java's main method static?

It is because the object is not required to call a static method. If it were a non-static method, JVM creates an object first then calls a main() method that will lead to the problem of extra memory allocation.

Java static block

- used to initialize the static data member.
- executed before the main method at the time of classloading.

```java
class Main{
  static{
      System.out.println("This block is executed first than main method");
  }
  public static void main(String args[]){
   System.out.println("Hello main");
  }
}

```

input

```
This block is executed first than main method
Hello main


...Program finished with exit code 0
```

NOTE:-
We can not execute a program without the main() method. One of the ways was the static block, but it was possible till JDK 1.6. Since JDK 1.7, it is not possible to execute a Java class without the main method.

Final

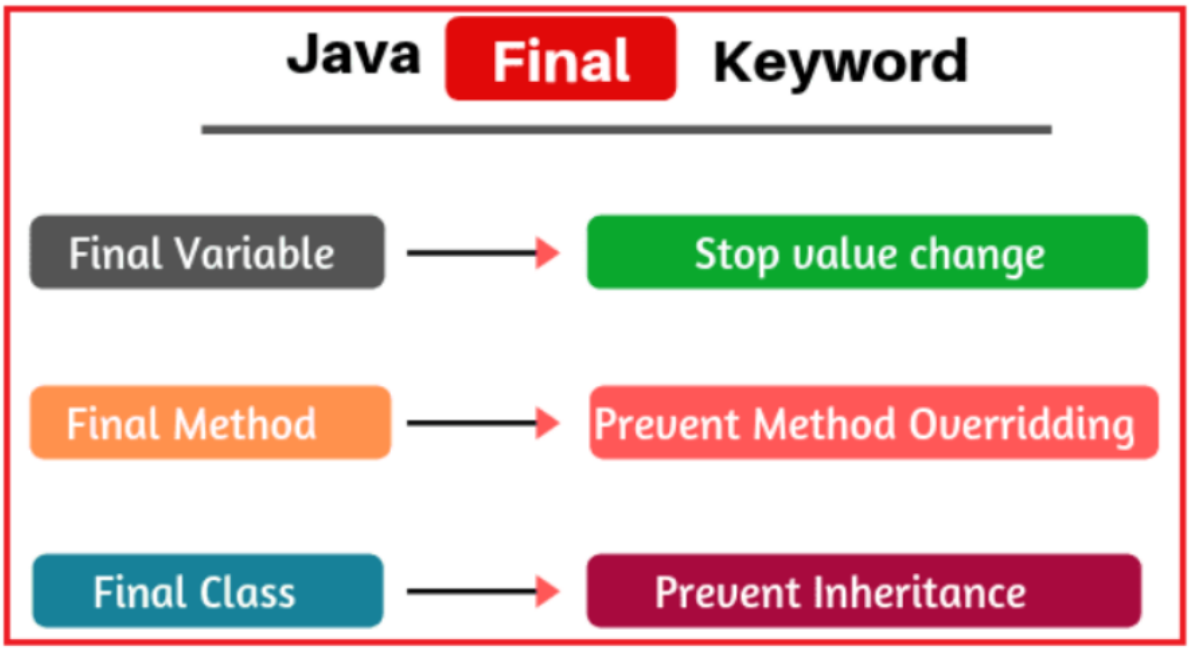
We will cover the following

- Final keyword
- Java Final variable
- Java Final method
- Java Final class

Final keyword

The final keyword in java is used to restrict the user. Java final keyword can be used with the following.

1. Variable
2. Method
3. class



Java Final variable

once the variable is declared as final then its value can not be changed.
E.g. here in the example below, you can see that I have declared the variable speedlimit and initialized it with a value of 90. Now, with the run method's help, when I am trying to change its value, the compiler throws the error.

```
1  class rollsroyace{
2   final int speedlimit=90;//final variable
3   void run(){
4    speedlimit=400;
5   }
6   public static void main(String args[]){
7   rollsroyace mycar=new  rollsroyace();
8   mycar.run();
9   }
10 }//end of class
```

input

Compilation failed due to following error(s).

```
Main.java:4: error: cannot assign a value to final variable speedlimit
  speedlimit=400;
  ^
1 error
```

Java Final Method

if you declare your method as final, you can not override that method.
Eg.

```
1  class Bike{
2    final void run(){System.out.println("running the bike ");}
3  }
4
5  class Honda extends Bike{
6     void run(){System.out.println("running safely with 80kmph");}
7
8     public static void main(String args[]){
9     Honda honda= new Honda();
10    honda.run();
11    }
12 }
13
```

input

Compilation failed due to following error(s).

```
Main.java:6: error: run() in Main cannot override run() in Bike
   void run(){System.out.println("running safely with 80kmph");}
        ^
  overridden method is final
1 error
```

Here you can notice that the run method of the Bike class can not be redefined in its child class as the run method is declared as final. So the compiler is throwing the error.

Java Final Class

If you make any class as final, you cannot extend that class.

```
1  final class Bike{
2     void run(){System.out.println("running the bike ");}
3  }
4
5  class Honda extends Bike{
6     void run(){System.out.println("running safely with 80kmph");}
7
8     public static void main(String args[]){
9     Honda honda= new Honda();
10    honda.run();
11    }
12 }
13
```

input

Compilation failed due to following error(s).

```
Main.java:5: error: cannot inherit from final Bike
class Honda extends Bike{
                    ^
1 error
```

Here you can notice that the Bike class is declared as final, so we are not able to override it; that's why the compiler is throwing the error.

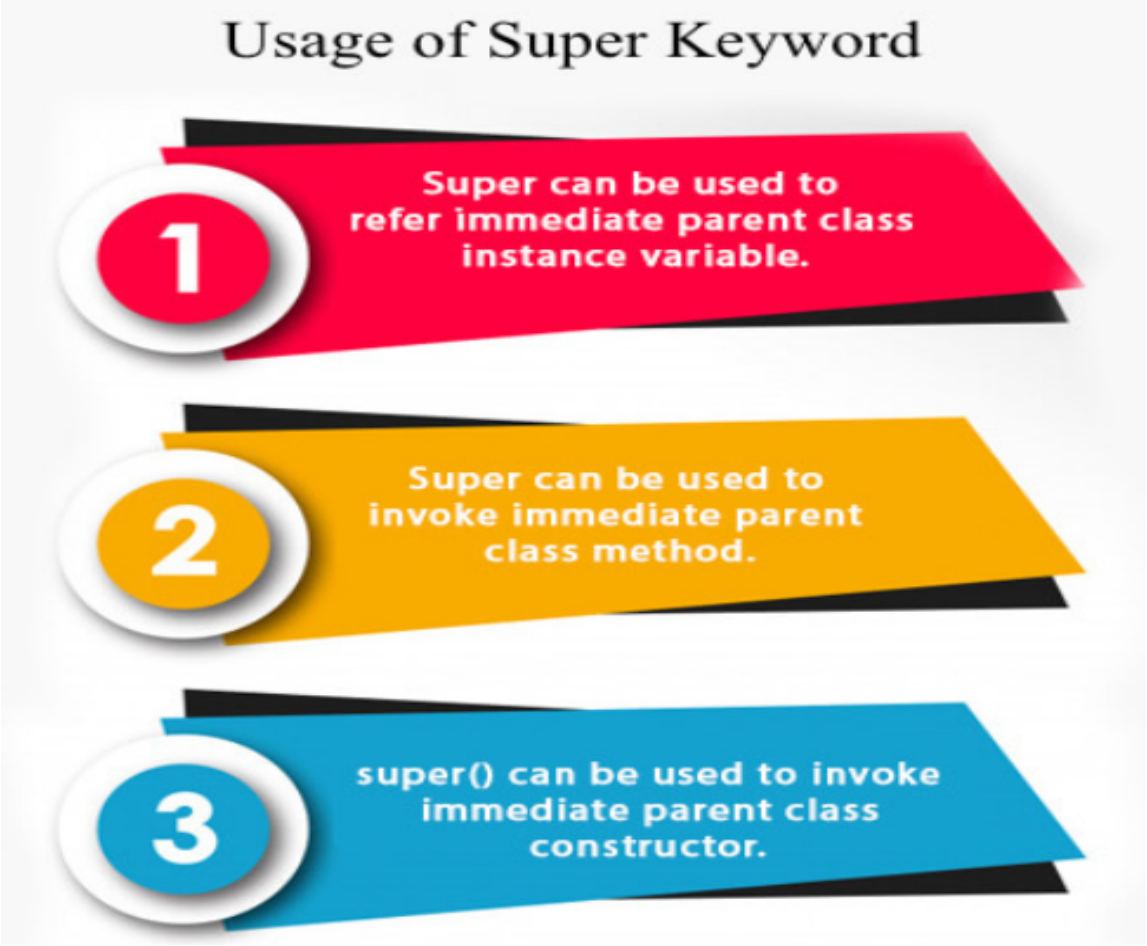
Super keyword

We will cover the following

- Super Keyword
- refer to the immediate class variable
- Invoke the immediate parent class method
- Invoke the immediate parent class constructor

Super keyword

The **super** keyword in Java is a reference variable that is used to refer to the immediate parent class object.



Refer to the immediate parent class variable

```
class Car
{

String color = "white";

}

class Audi extends Car
{

String color = "black";

void printColor ()
  {

    System.out.println ("current class: "+color);   //prints color of Audi class
    System.out.println ("parent class: "+super.color);  //prints color of Car class
   }
}
class Main

{

public static void main (String args[])
  {

    Audi d = new Audi ();

    d.printColor ();

  }

}
```

input

```
current class: black
parent class: white


...Program finished with exit code 0
```

Here you can see that super.color prints the value of the color variable of the immediate parent class.

Invoke the immediate parent class method

```
1  class Animal
2  {
3    void eat ()
4    {
5      System.out.println ("eating... from parent");
6    }
7  }
8  class Dog extends Animal
9  {
10   void bark ()
11   {
12     System.out.println ("barking...");
13   }
14   void work ()
15   {
16
17     super.eat ();// calling the parent class method
18
19     bark ();
20
21   }
22 }
23
24 class Main
25 {
26
27   public static void main (String args[])
28   {
29
30     Dog d = new Dog ();
31
32       d.work ();
33
34 }}
35
```

```
eating... from parent
barking...


...Program finished with exit code 0
```

Here you can see that super.eat() calls the eat method of the parent class.

Invoke the immediate parent class method

```
1  class Animal
2  {
3    Animal ()
4    {
5      System.out.println ("calling constructor... from parent");
6    }
7  }
8  class Dog extends Animal
9  {
10   Dog(){
11       super();//calling parent constructor
12   }
13 }
14
15 class Main
16 {
17
18   public static void main (String args[])
19   {
20
21     Dog d = new Dog ();
22
23   }
24 }
25
```

```
calling constructor... from parent


...Program finished with exit code 0
```

super() is used here to call the parent class constructor.

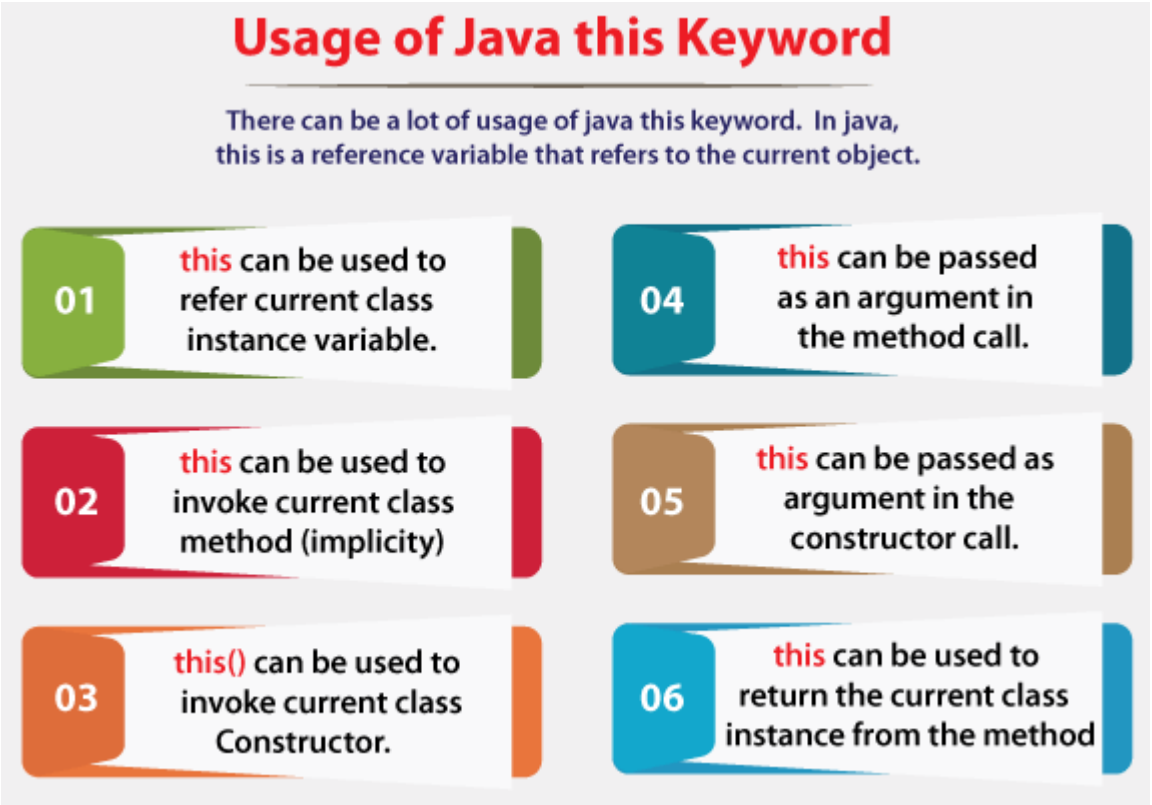
this keyword

We will cover the following

- this keyword
- refer to current class instance variable
- this: to invoke the current class method
- this: to invoke the current class constructor
- this: to pass as an argument in the method
- this keyword can be used to return the current class instance

this keyword

There can be a lot of usage of **Java this keyword**. In Java, this is a **reference variable** that refers to the current object.



this: refer to a current class instance variable

there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

Let's take the case when we do not use this keyword then what happens actually

```
class Student{

    int rollno;

    String name;

    float fee;

    Student (int rollno, String name, float fee){

    rollno = rollno;

    name = name;

    fee = fee;

    }
    void display (){
        System.out.println (rollno + " " + name + " " + fee);
    }
}
class Main{

public static void main (String args[]){

Student s1 = new Student (111, "ankit", 5000f);

s1.display ();

}}
```

```
0 null 0.0


...Program finished with exit code 0
```

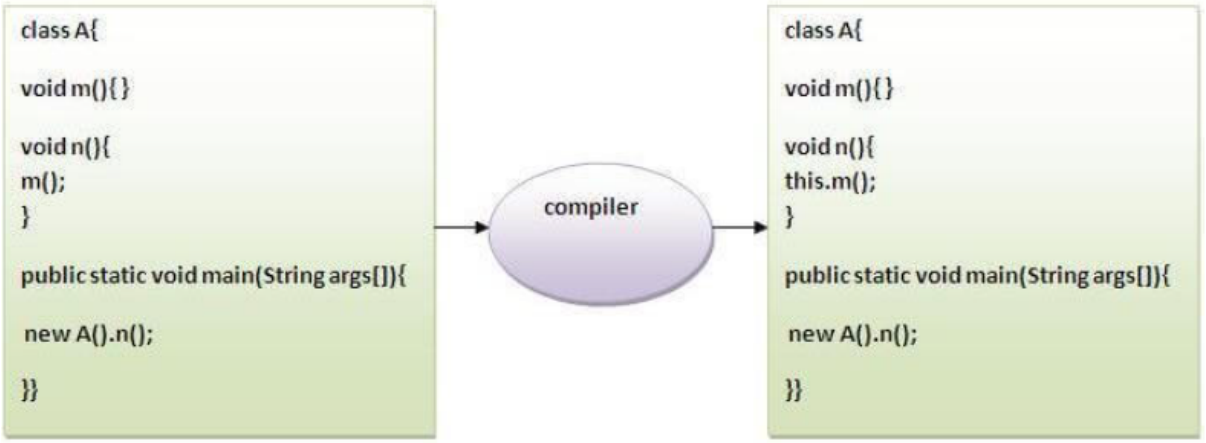
now you can notice that the values of roll no, name and fee do not change even we have assigned the values in the constructor. this is because the compiler is not able to distinguish local variables and instance variables because of the same names. this issue gets resolved using this keyword.

```java
class Student{

    int rollno;

    String name;

    float fee;

    Student (int rollno, String name, float fee){

    this.rollno = rollno;

    this.name = name;

    this.fee = fee;

    }
    void display (){
        System.out.println (rollno + " " + name + " " + fee);
    }
}
class Main{

public static void main (String args[]){

Student s1 = new Student (50, "vinay", 5500f);

s1.display ();

}}
```

```
50 vinay 5500.0


...Program finished with exit code 0
```

this: to invoke the current class method

this can be used to invoke the current class method. But If you don't use this keyword, the compiler automatically adds this keyword while invoking the method. Let's see the example




```java
class thiskeyworddemo
{

    void myfunction (){
        System.out.println ("hello you are myfunction");
      }
    void call (){

    System.out.println ("hello you are call");

          //myfuntion();//same as this.myfuntion()
          this.myfunction ();

    }
}
class Main{

    public static void main (String args[]){
    thiskeyworddemo demo = new thiskeyworddemo ();
    demo.call();

    }

}
```

```
hello you are call
hello you are myfunction


...Program finished with exit code 0
```

this: to invoke the current class constructor

The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

```
class thiskeyworddemo
{

    thiskeyworddemo (){
        this(10);
      }

    thiskeyworddemo (int k){
        System.out.println ("you have called the parameter constructor with value "+k);
      }
}
class Main{

    public static void main (String args[]){
    thiskeyworddemo demo = new thiskeyworddemo ();
  }

}
```

input

```
you have called the parameter constructor with value 10


...Program finished with exit code 0
```

this: to pass as an argument in the method

The this keyword can also be passed as an argument in the method. It is mainly used in the event handling. Let's see the example:

```
class thiskeyworddemo
{
     int testValue=3493;
     void func1 (){
         System.out.println ("hello you are myfunction");
         func2(this);
       }
     void func2 (thiskeyworddemo obj){
         System.out.println (obj.testValue);
     }
}
class Main{
     public static void main (String args[]){
     thiskeyworddemo demo = new thiskeyworddemo ();
     demo.func1();

     }

}
```

```
hello you are myfunction
3493


...Program finished with exit code 0
```

this keyword can be used to return the current class instance

We can return this keyword as a statement from the method. In such a case, the return type of the method must be the class type (non-primitive). Let's see the example:

```
class thiskeyworddemo
{
    int testValue=3493;
    void func1 (){
        System.out.println ("hello happy learning!!");

     }
    thiskeyworddemo func2 (){
        return this;
    }
}
class Main{
    public static void main (String args[]){
    thiskeyworddemo demo = new thiskeyworddemo ();
    thiskeyworddemo returnedobject = demo.func2();
     returnedobject.func1();
    }

}
```

```
hello happy learning!!


...Program finished with exit code 0
```

Interview Questions

1. What is a static keyword in Java?

Static is a Non-Access Modifier. Static can be applied to variable, method, nested class, and initialization blocks (static block).

2. Why main() method is declared as static?

If our **main()** method is not declared as static then the JVM has to create an object first and call which causes the problem of having extra memory allocation.

3. Can constructors be static in Java?

In general, a static method means that "The Method belongs to the class and not to any particular object" but a constructor is always invoked with respect to an object, so it makes no sense for a constructor to be **static**.

4. Can we use this to refer static members?

Yes, it's possible to access the static variables of a class using this but it's discouraged and as per best practices this should be used on nonstatic reference.

5. What are all the differences between this and the super keyword?

- This refers to the current class object whereas super refers to the superclass object
- Using this we can access all non-static methods and variables. Using super we can access superclass variables and methods from sub-class.
- Using this(); call we can call other constructors in the same class. Using super we can call superclass constructor from sub-class constructor.

6. What is a final method?

When a method is declared as **final**, then it is called a **final method,** The subclass can call the final method of the parent class but cannot **override** it.

7. Can a main() method be declared final?

Yes, the **main()** method can be declared as final and cannot be **overridden**.

8. What is a Static Final variable in Java?

When have declared a variable as **static final** then the variable becomes a **CONSTANT**. Only one copy of the variable exists which cannot be changed by any instance.

Encapsulation

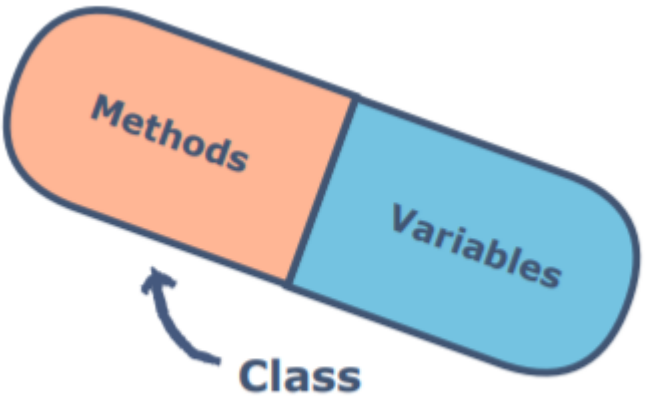
We will cover the following

- Definition
- Advantages of encapsulation
- An example of encapsulation

Definition:-

Encapsulation is a fundamental programming technique in OOP used to achieve data hiding.

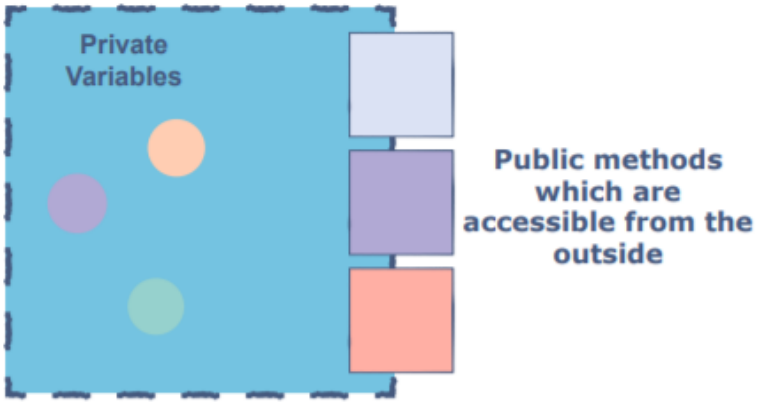
Depending upon this **unit**, objects are created. Encapsulation is normally done to hide the state and representation of an object from the outside. A class can be thought of as a **capsule** having methods and data members inside it.



As a rule of thumb, a good convention is to declare all the data members or instance variables of a class private. This will restrict direct access from the code outside that class.

At this point, a question can be raised that if the methods and variables are encapsulated in a class then “how can they be used outside of that class”?

Well, the answer to this is simple. One has to implement methods to let the outside world communicate with this class. These methods can be getters, setters, and any other custom methods implemented by the programmer.



Advantages of Encapsulation

- Classes are easier to change and maintain.
- We can specify which data member we want to keep hidden or accessible.
- We decide which variables have read/write privileges (increases flexibility).

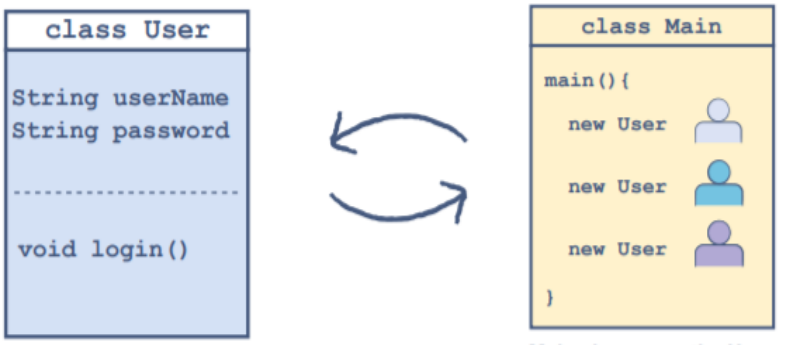
An Example of encapsulation:-

As discussed earlier, encapsulation refers to the concept of binding data and the methods operating on that data in a single unit also called a class.

The goal is to prevent this bound data from any unwanted access by the code outside this class.

Let's understand this using an example of a very basic User class. Consider that we are up for designing an application and are working on modeling the log-in part of that application. We know that a user needs a username and a password to log into the application.

A very basic User class will be modeled as: Having a field for the userName Having a field for the password A method named login() to grant access Whenever a new user comes, a new object can be created by passing the userName and password to the constructor of this class. class User String userName String password void login()



Main class uses the User class to create new Objects

How can we implement Encapsulation in java:-

```java
/* File name : EncapTest.java */
public class EncapTest {
   private String name;
   private String idNum;
   private int age;

   public int getAge() {
      return age;
   }

   public String getName() {
      return name;
   }

   public String getIdNum() {
      return idNum;
   }

   public void setAge( int newAge) {
      age = newAge;
   }

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

   public void setIdNum( String newId) {
      idNum = newId;
   }
}
```

Note:-
Here you can see getter and setter methods we have used here in order to change the private data members of the class because these variables are only accessible to class methods and variables and any outside class member or object can not change these private variables. so this is the concept of encapsulation.

```java
/* File name : RunEncap.java */
public class RunEncap {

   public static void main(String args[]) {
      EncapTest encap = new EncapTest();
      encap.setName("James");
      encap.setAge(20);
      encap.setIdNum("12343ms");

      System.out.print("Name : " + encap.getName() + " Age : " + encap.getAge());
   }
}
```

output of above code:-

```
Name : James Age : 20
```

Inheritance

We will cover the following:-

- Definition
- Is-A relationship
- Has-A relationship
- Types of Inheritance
- single Inheritance
- multilevel Inheritance
- hierarchical Inheritance
- multiple inheritance not supported by java?

Definition:-

Inheritance is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of oops

Inheritance provides a way to create a new class from an existing class. The new class is an extended version of the existing class such that it inherits all the non-private fields (variables) and methods of the existing class. The existing class is used as a starting point or as a base to create the new class.

The Is-A Relationship

After reading the above definition, the next question that comes to your mind is What is the use case of inheritance? Well, the answer is that wherever we come across an IS-A relationship between objects, we can use inheritance.

Existing Class	Derived Class
Shape	Square
Programming Language	Java
Vehicle	Car

The Has-A relationship

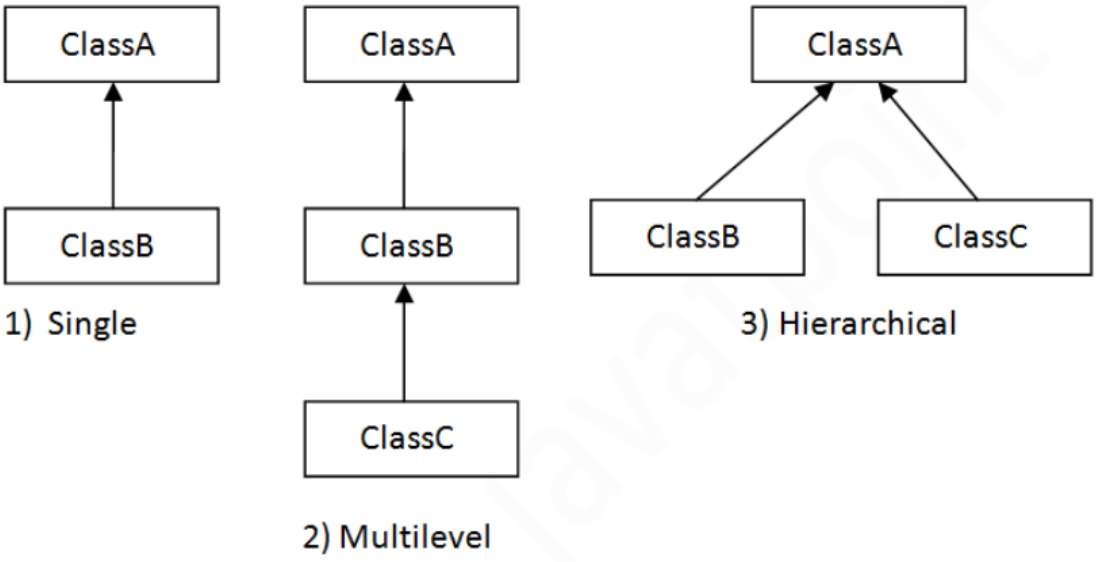
here you can see these are not is-A examples instead these are Has-A relationships between them.

Types of Inheritance:-

On the basis of class, there can be three types of inheritance in java: single, multilevel, and hierarchical.

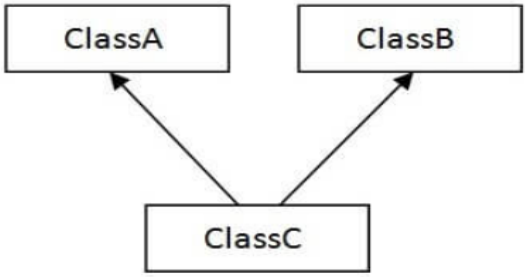
In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.

1. Single
2. Multi-level
3. Hierarchical
4. Multiple
5. Hybrid

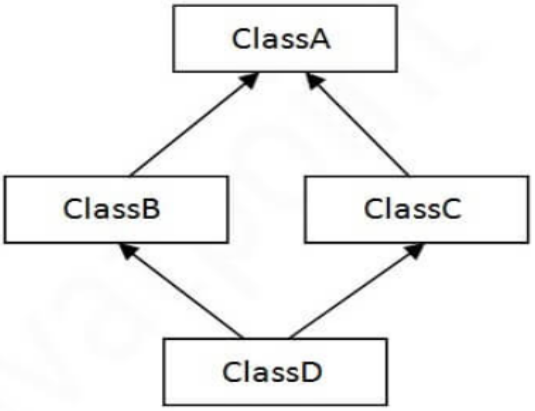


Note: - Multiple inheritance is not supported in Java through the class.

When one class inherits multiple classes, it is known as multiple inheritance. For Example:



4) Multiple



5) Hybrid

Single Inheritance

When a class inherits another class, it is known as single inheritance.

```
1  class Animal
2  {
3
4      void eat (){
5          System.out.println (" Is eating the break");
6      }
7  }
8
9  class Dog extends Animal
10 {
11
12   void bark (){
13     System.out.println (" Is barking over the thief");
14   }
15
16 }
17 class Main
18
19 {
20
21     public static void main (String args[]){
22
23         Dog d = new Dog ();
24
25         d.bark ();
26
27         d.eat ();
28     }
29
30 }
```

```
Is barking over the thief
Is eating the break


...Program finished with exit code 0
```

Multilevel inheritance

When there is a chain of inheritance, it is known as multilevel inheritance.

```
class Animal{
    void eat (){
        System.out.println (" Is eating the break");}
}

class Dog extends Animal{
  void bark (){
    System.out.println (" Is barking over the thief");}
}
class childDog extends Dog{
     void weep(){
        System.out.println(" Is weeping...");}
}

class Main{
   public static void main (String args[]){
        childDog d=new childDog();
        d.weep();
        d.bark();
        d.eat();
    }
}
```

```
Is weeping...
Is barking over the thief
Is eating the break


...Program finished with exit code 0
```

Hierarchical inheritance

When two or more classes inherit a single class, it is known as hierarchical inheritance. here in the example below, you can see two different classes are inherited through the same single class.

```java
class Animal{
    void eat (){
        System.out.println (" Is eating the breakfast");}
}

class Dog extends Animal{
  void bark (){
    System.out.println (" Is barking over the thief");}
}
class Cat extends Animal{
    void meow(){
        System.out.println(" Is meowing due to fear");}
}

class Main{
   public static void main (String args[]){
        Cat c=new Cat();
        c.eat();
        c.meow();
    }
}
```

```
Is eating the breakfast
Is meowing due to fear


...Program finished with exit code 0
```

multiple inheritance is not supported by java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

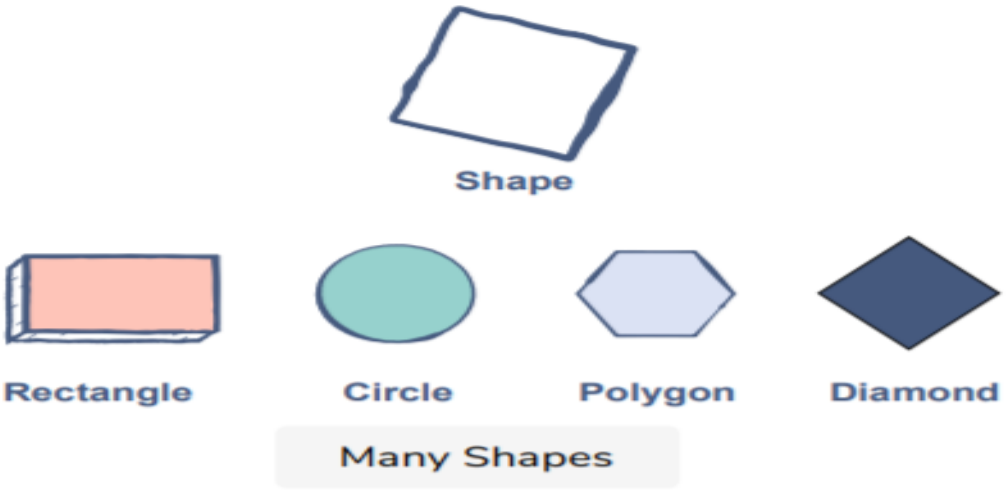
Polymorphism

We will cover the following:-

- Definition
- Types of polymorphism
- Compile-time polymorphism
- Method overloading with example
- Run-time polymorphism.
- Method overriding with example

Definition:-

The word Polymorphism is a combination of two Greek words, Poly means many and Morph means forms.
In programming, polymorphism refers to the same object exhibiting different forms and behaviors. For example, take the Shape Class. The exact shape you choose can be anything. It can be a rectangle, a circle, a polygon, or a diamond. So, these are all shapes but their properties are different. This is called Polymorphism



Types of polymorphism

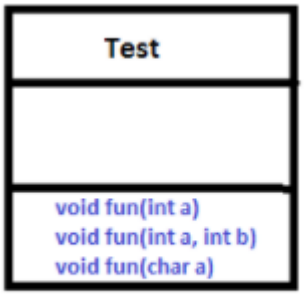
- Compile-time polymorphism
- Runtime polymorphism

Compile-time polymorphism

It is also known as static polymorphism. This type of polymorphism is achieved by Method Overloading or operator overloading
Note:- operator overloading is not supported by java.

Method overloading:-

when we have more than one function/method in the same class with the same name and number of arguments. then these functions are known as overloaded functions. Functions can be overloaded by a change in the number of arguments or/and a change in the type of arguments.
Note:-different return type is not considered as overloading.



Overloading

Here, you can see there are three functions in the same class having the same name and the same number of arguments. So these functions are overloaded.

Here we will look at an example of the same with the help of a program written in java. We have overloaded multiply function using different arguments types, and a number of arguments are different in each function.

Main.java

Main.java

```java
// Helper class
class Helper {

    // Method with 2 integer parameters
    static int Multiply(int one, int two)
    {
        // Returns product of integer numbers
        return one * two;
    }

    // Method 2
    // With same name but with 2 double parameters
    static double Multiply(double one, double two)
    {
        // Returns product of double numbers
        return one * two;
    }

   static int Multiply(int p, int q,int r)
    {
        // Return product
        return p * q * r;
    }
}

// Class 2
// Main class
class Main {

    // Main driver method
    public static void main(String[] args)
    {
        // Calling method by passing
        // input as in arguments
        System.out.println(Helper.Multiply(5,4));
        System.out.println(Helper.Multiply(6,4,8));
        System.out.println(Helper.Multiply(5.5, 6.3));
    }
}
```

```
20
192
34.65


...Program finished with exit code 0
```

Run-time polymorphism

It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. Method Overriding achieves this type of polymorphism. On the other hand, method overriding occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

Method overriding

In a simple language, when we have two classes, one is child class, and the other is parent class, and when we write the same function in both the child class and the parent class, the method is said to be overridden. This concept is known as runtime polymorphism because the compiler will decide at runtime to which function it will call during the program's execution.

Main.java

```java
// parent class
class vehicle {
    // Method of parent class
    void run()
    {
        System.out.println("vehicle is running ");
    }
}
// child class
class  car extends  vehicle{

    // Method
    void run() {
        System.out.println("car is running");
    }
}

// Main class
class Main {

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

        // Creating object
        vehicle pqr =new car();

        // Now we will be calling print methods
        // inside main() method
        pqr.run();

        //when the object behaves as vehicle
        vehicle abc=new vehicle();
        abc.run();
    }
}
```

```
car is running
vehicle is running


...Program finished with exit code 0
```

Here, you can easily understand that the run method is called at the runtime, according to whether the vehicle is behaving like a car, or the vehicle is behaving like the vehicle itself.

Abstraction

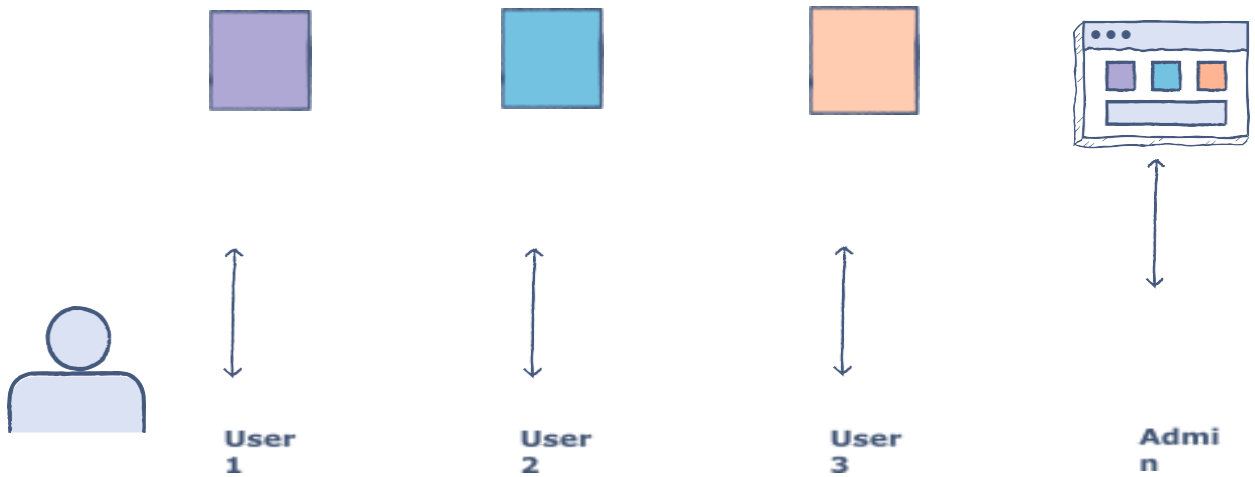
we will cover the following:-

- Definition
- an example from java
- abstract data types
- Rules and implementation of abstraction in java

Definition:-

Abstraction in Object-Oriented Programming refers to showing only the essential features of an object to the user and hiding the inner details to reduce complexity. It can be put this way that the user only has to know “what an object does?” rather than “how it does?”.

Real-world Examples #

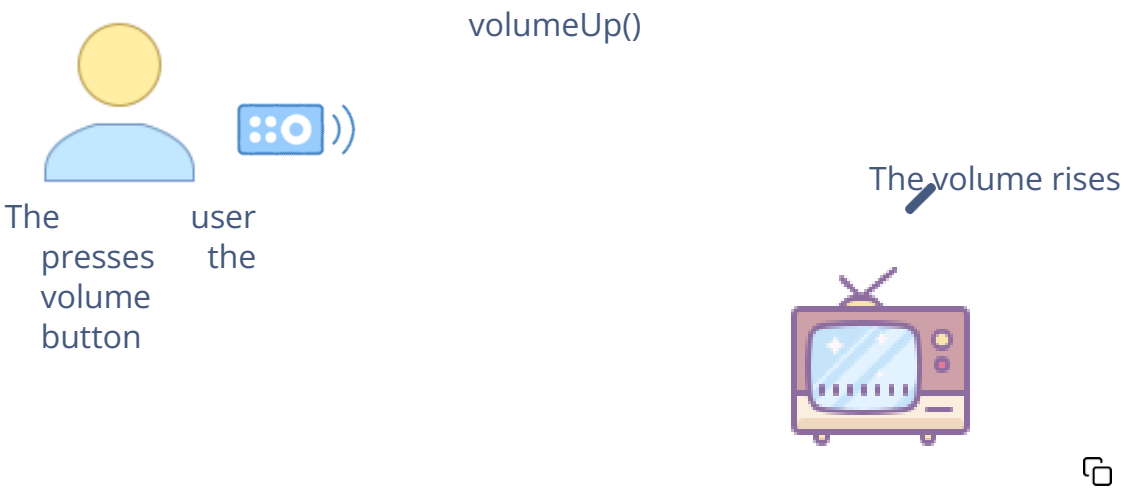


The above illustration of the users and the admin of an application is a good real-world example of abstraction

A user can only use and interact with the limited features of an application and is unaware of the implementation details or the way the application was developed. Usually, the users are only concerned with the functionality of an application.

An admin can have the access to a lot more features of the application and nothing is hidden from him. The admin can monitor the activity of the users, knows how the application was developed and can implement new features by deploying them in the application.

In the above example, the abstraction is being applied to the user but not to the admin.



Let's look into another example of abstraction. Take the Volume button on a television remote. With a click of a button, we request the T.V. to increase its volume. Let's say the button calls the Volume Up() function. The T.V. responds by producing a sound larger than before. How the inner circuitry of the T.V. implements this is oblivious to us, yet we know the exposed function needed to interact with the T.V.'s volume.

An Example from Java

In Java, one can very easily see abstraction in action. Let's take an example of Java Math class. There are a lot of in-built methods in this class that can be used by the programmer to get facilitated. Let's use a few methods in our code to access the in-built functionality:

```
class TestAbstraction {

        public static void main( String args[] ) {

            int min = Math.min(15,18);      //find min of two numbers

            double square = Math.pow(2,2); //calculate the power of a number

            System.out.println("The min of 15 & 18 is:
            "+ min); System.out.println("The square of
            2 is: " + square)


        }
    }
```

In the above code:

Math.min() find min of two num

Math.max() find max of two num

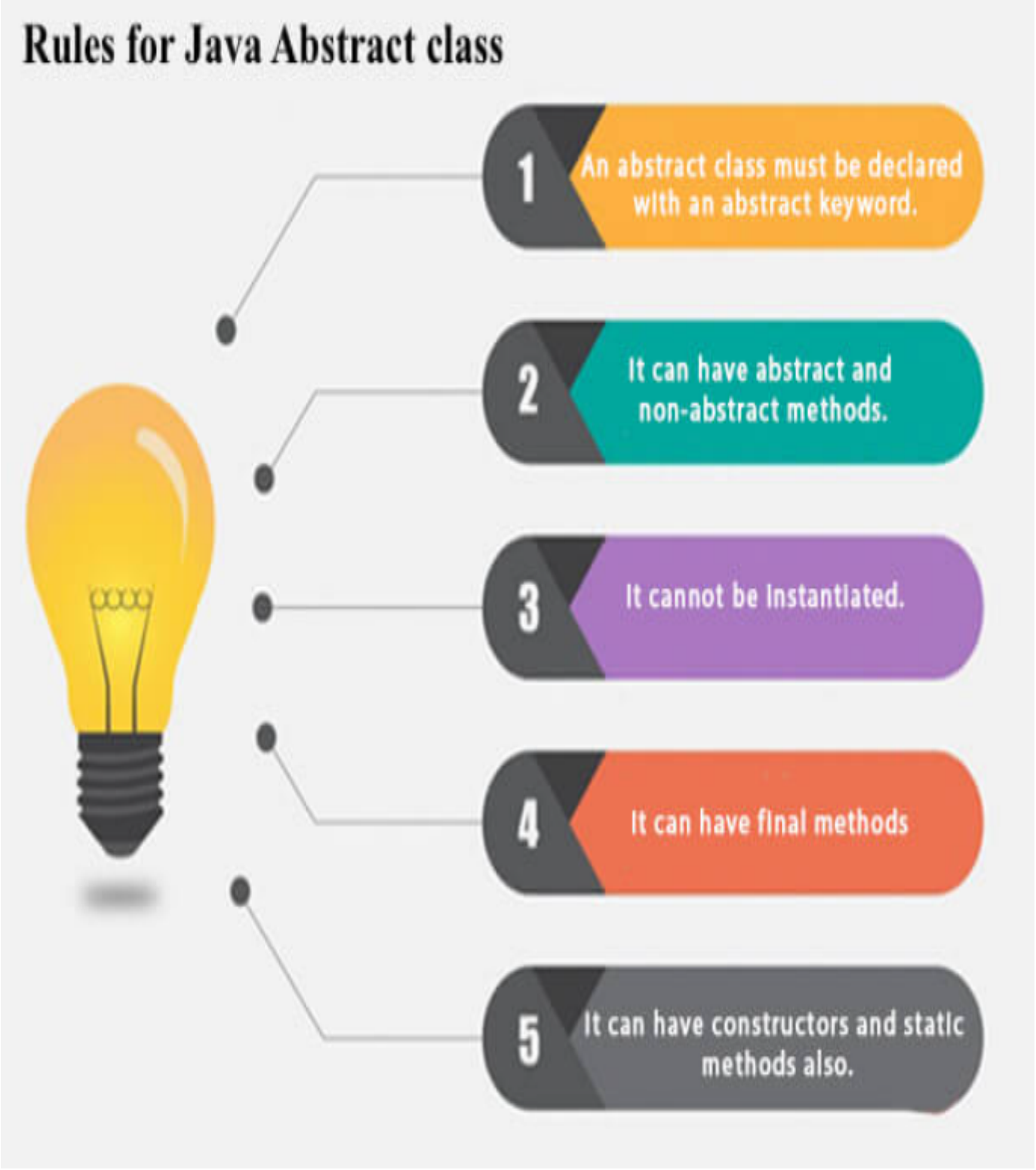
But the user doesn't have to know about the implementation of these two methods inside the Math class

Abstract Data Types

By the definition of abstract data types, the users only get to know the essentials i.e. the functionality of those data types, and the 'how the implementation should be done to achieve the specified functionality?' part is hidden.

An example of abstract data type can be built in stack class in java in which the user knows that it has pop push functions but the user doesn't know how there are implemented

Rules for java abstract class:-



How can we implement abstraction in java?

source code

```java
/* File name : Employee.java */
public abstract class Employee {
   private String name;
   private String address;
   private int number;

   public Employee(String name, String address, int number) {
      System.out.println("Constructing an Employee");
      this.name = name;
      this.address = address;
      this.number = number;
   }

   public double computePay() {
     System.out.println("Inside Employee computePay");
     return 0.0;
   }

   public void mailCheck() {
      System.out.println("Mailing a check to " + this.name + " " + this.address);
   }

   public String toString() {
      return name + " " + address + " " + number;
   }

   public String getName() {
      return name;
   }

   public String getAddress() {
      return address;
   }

   public void setAddress(String newAddress) {
      address = newAddress;
   }

   public int getNumber() {
      return number;
   }
}
```

when you will compiler this code by creating the object of the employee class you will get to know that this will fall into error as this violates the rules of abstraction.AS you can not create an object of the abstract class instead you can inherit it.

```java
/* File name : AbstractDemo.java */
public class AbstractDemo {

   public static void main(String [] args) {
      /* Following is not allowed and would raise error */
      Employee e = new Employee("rohan sharma", "Houston, TX", 43);
      System.out.println("\n Call mailCheck using Employee reference--");
      e.mailCheck();
   }
}
```

input

Compilation failed due to following error(s).

```
Main.java:6: error: cannot access Employee
      Employee e = new Employee("George W.", "Houston, TX", 43);
      ^
  bad source file: ./Employee.java
    file does not contain class Employee
    Please remove or make sure it appears in the correct subdirectory of the sourcepath.
1 error
```

Now we will follow the abstraction rules. we will not create the object directly instead we will inherit the abstract class first and implement the abstract methods of the parent class in the child class then we will create the object of the child class.

```java
/* File name : Salary.java */
public class Salary extends Employee {
   private double salary;   // Annual salary

   public Salary(String name, String address, int number, double salary) {
      super(name, address, number);
      setSalary(salary);
   }

   public void mailCheck() {
      System.out.println("Within mailCheck of Salary class ");
      System.out.println("Mailing check to " + getName() + " with salary " + salary);
   }

   public double getSalary() {
      return salary;
   }

   public void setSalary(double newSalary) {
      if(newSalary >= 0.0) {
         salary = newSalary;
      }
   }

   public double computePay() {
      System.out.println("Computing salary pay for " + getName());
      return salary/52;
   }
}
```

Main.java | Employee.java | Salary.java | AbstractDemo.java

```java

/* File name : AbstractDemo.java */
public class AbstractDemo {

   public static void main(String [] args) {
      Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);
      Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);
      System.out.println("Call mailCheck using Salary reference --");
      s.mailCheck();
      System.out.println("\n Call mailCheck using Employee reference--");
      e.mailCheck();
   }
}
```

```
Constructing an Employee
Constructing an Employee
Call mailCheck using Salary reference --
Within mailCheck of Salary class
Mailing check to Mohd Mohtashim with salary 3600.0

 Call mailCheck using Employee reference--
Within mailCheck of Salary class
Mailing check to John Adams with salary 2400.0


...Program finished with exit code 0
```

Interface

We will cover the following:-

- Definition
- Declaration
- why do we use an interface
- Relationship between classes and interface
- Multiple inheritance in java through the interface
- An Example of Multiple inheritance

Definition:-

An interface can be thought of as a contract that a class has to fulfill while implementing an interface. According to this contract, the class that implements an interface has to @Override all the abstract methods declared in that very interface.

An interface can be used to achieve 100% abstraction as it contains the method signatures/abstract methods(what to be done) and no implementation details (how to be done) of these methods. In this way, interfaces satisfy the definition of abstraction. The implementation techniques of the methods declared in an interface are totally up to to the classes implementing that interface.

Declaration

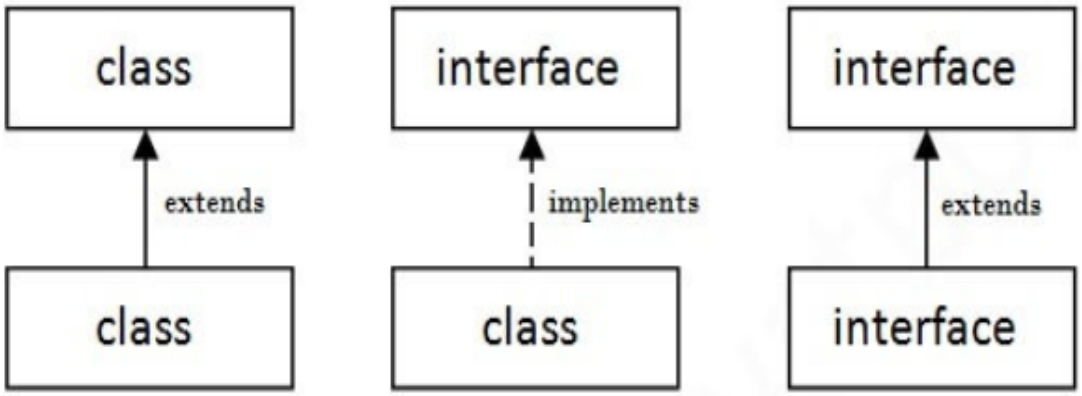
the interface is declared with the Interface keyword

```
interface interfaceName {
   // Code goes here
}
```

Why do we use an interface?

The relationship between classes and interfaces

a class extends another class, an interface extends another interface, but a class implements an interface.



Example of interface

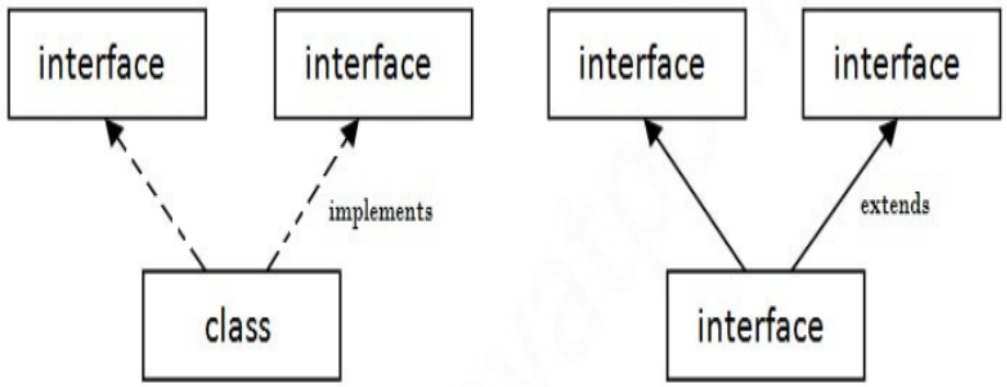
Main.java

```java
interface printer{
  void print();
}
class printable implements printer{
    public void print(){
        System.out.println("Hello");
    }
 }

public class Main{
    public static void main(String args[]){
    printable obj = new printable();
    obj.print();
 }
}
```

```
Hello


...Program finished with exit code 0
```

Multiple inheritance in java through the interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple Inheritance in Java

An Example of multiple inheritance using the interface

Main.java

```java
interface printer{
  void print();
}
interface Showable{
   void show();
}
class printable implements printer,Showable{
    public void print(){
        System.out.println("Hello");
    }
    public void show(){
        System.out.println("Welcome to codingninjas..!!");
    }
 }

public class Main{
    public static void main(String args[]){
    printable obj = new printable();
    obj.print();
    obj.show();
 }
}
```

```
Hello
Welcome to codingninjas..!!


...Program finished with exit code 0
```

Interview Questions

Q1.What is the difference between compile-time polymorphism and runtime polymorphism?

SN	compile-time polymorphism	Runtime polymorphism
1	In compile-time polymorphism, call to a method is resolved at compile-time.	In runtime polymorphism, call to an overridden method is resolved at runtime.
2	It is also known as static binding, early binding, or overloading.	It is also known as dynamic binding, late binding, overriding, or dynamic method dispatch.
3	Overloading is a way to achieve compile-time polymorphism in which, we can define multiple methods or constructors with different signatures.	Overriding is a way to achieve runtime polymorphism in which, we can redefine some particular method or variable in the derived class. By using overriding, we can give some specific implementation to the base class properties in the derived class.
4	It provides fast execution because the type of an object is determined at compile-time.	It provides slower execution as compare to compile-time because the type of an object is determined at run-time.
5	Compile-time polymorphism provides less flexibility because all the things are resolved at compile-time.	Run-time polymorphism provides more flexibility because all the things are resolved at runtime.

Q2.What is the Java instanceOf operator?

The instanceof in Java is also known as type comparison operator because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has a null value, it returns false.

Q3.What is the difference between abstraction and encapsulation?

Abstraction hides the implementation details whereas encapsulation wraps code and data into a single unit.

Q4.What are the differences between abstract class and interface?

Abstract class	Interface
An abstract class can have a method body (non-abstract methods).	The interface has only abstract methods.
An abstract class can have instance variables.	An interface cannot have instance variables.
An abstract class can have the constructor.	The interface cannot have the constructor.
An abstract class can have static methods.	The interface cannot have static methods.
You can extend one abstract class.	You can implement multiple interfaces.
The abstract class **can provide the implementation of the interface**.	The Interface **can't provide the implementation of the abstract class**.
The **abstract keyword** is used to declare an abstract class.	The **interface keyword** is used to declare an interface.
An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
An **abstract class** can be extended using keyword **extends**	An **interface class** can be implemented using keyword **implements**
A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

Example:
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Q5.What is a superclass?

A superclass—also called a base class—is a class that is a parent for more classes rather than objects. It usually contains the most basic code and data that will be used by every class and object under it. Using the above example, ‘beverage’ and ‘machine’ could be superclasses for ‘soda’ and ‘computer.’

Q6.What is a subclass?

A subclass is a class that falls under a superclass. It inherits from the superclass and is considered to have an “is-a” relationship with the superclass.

Q7.Are there any limitations of Inheritance?

Yes, with more powers comes more complications. Inheritance is a very powerful feature in OOPs, but it has some limitations too. Inheritance needs more time to process, as it needs to navigate through multiple classes for its implementation. Also, the classes involved in Inheritance - the base class and the child class, are very tightly coupled together. So if one needs to make some changes, they might need to do nested changes in both classes. Inheritance might be complex for implementation, as well. So if not correctly implemented, this might lead to unexpected errors or incorrect outputs.

Q8.What are the various types of inheritance?:

- Single inheritance
- Multiple inheritances
- Multi-level inheritance
- Hierarchical inheritance
- Hybrid inheritance

Q9. What is meant by static polymorphism?

Static Polymorphism is commonly known as the Compile time polymorphism. Static polymorphism is the feature by which an object is linked with the respective function or operator based on the values during the compile time. Static or Compile time Polymorphism can be achieved through Method overloading or operator overloading.