Class and Objects

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

Or

Collection of objects is called class. It is a logical entity. Once a class has been defined, we can create any number of objects belongs to that class.

Each object associated with data and Methods.

For Example:

"Fruit" is a class and Mango, Apple and Banana are Objects.

- Fields
- Methods
- Constructors
- Blocks
- Nested class and interface

Once a class has been defined, we can create any number of objects belongs to that class. Each object associated with data and Methods.

For Example "Fruit" is a class and Mango, Apple and Banana are objects.

```
class <class_name>

{

field; method;

}
```

Example:1

```
class Student
{
 int id; //field or data member or instance variable
String name;
public static void main(String args[])
{
Student s1=new Student();//creating an object of Student
//Printing values of the object
System.out.println(s1.id); //accessing member through reference variable
System.out.println(s1.name);
}
}
```

Output:

```
O
Null
```

Example: 2

```
class Student
{
int id;
String name;
}
class TestStudent3
{
public static void main(String args[])
{
Student s1=new Student();  //Creating objects
Student s2=new Student();
s1.id=101;   //Initializing objects
s1.name="Sonu";
s2.id=102;
s2.name="Monu";
System.out.println(s1.id+" "+s1.name);  //Printing data
System.out.println(“ \n”);
System.out.println(s2.id+" "+s2.name);
}
}
```

Output:

```
    101                 Sonu
     102                Monu
```

Multiple Classes:

We can have multiple classes in different Java files or single Java file. If you define multiple classes in a single Java source file, it is a good idea to save the file name with the class name which has main() method.

```
//Java Program to demonstrate having the main method in
//another class
//Creating Student class.

class Student
{
int id=1122;
String name="Amith";
}
//Creating another class TestStudent1 which contains the main method
class Student123
{
public static void main(String args[])
{
Student s1=new Student();
System.out.println(s1.id);
System.out.println(" \n");
System.out.println(s1.name);
}
}
```

Output:

```
1122
Amith
```

Object in Java:

How to create the objects in different ways:

The Object is the real-time entity having some state and behavior. In Java, Object is an instance of the class having the instance variables as the state of the object and the methods as the behaviour of the object. The object of a class can be created by using the new keyword in Java Programming language.

Objects are run time entities in an object-oriented System. They may represent a person, place, Place, Bank Account, a table of data or anu other item. that has state and behavior is known as an object.

For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.

Java provides five ways to create an object.

1. Using **new** Keyword
2. Using **clone()** method
3. Using **newInstance()** method of the **Class** class
4. Using **newInstance()** method of the **Constructor** class
5. Using **Deserialization**

1.Using new keyword:

Using the new keyword in java is the most basic way to create an object. This is the most common way to create an object in java. By using this method we can call any constructor we want to call (no argument or parameterized constructors).

```
class Parul
{
// Declaring and initializing string
String name = " PARUL UIVERSITY @ Vadodara";
public static void main(String[] args) // Main driver method
{
Parul obj = new Parul();  // using new keyword
// Print and display the object
System.out.println(obj.name);
}
}
```

Output:

```
PARUL UIVERSITY @ Vadodara
```

2.Using Object Cloning – clone() method:

The clone() method is used to create a copy of an existing object, in order to the clone() method the corresponding class should have implemented a Cloneable interface which is again a Marker Interface.

Example:

```
public class CreateObjectExample3 implements Cloneable
{
protected Object clone() throws CloneNotSupportedException
{
//invokes the clone() method of the super class
return super.clone();
}
String str = "New Object Created";
public static void main(String[] args)
{
//creating an object of the class
CreateObjectExample3 obj1 = new CreateObjectExample3();
//try catch block to catch the exception thrown by the method
try
{
//creating a new object of the obj1 suing the clone() method
CreateObjectExample3 obj2 = (CreateObjectExample3) obj1.clone();
System.out.println(obj2.str);
}
catch (CloneNotSupportedException e)
{
e.printStackTrace();
}
}
}
```

Output:

```
New Object Created
```

3.Using newInstance() Method of Class class:

The **newInstance()** method of the Class class is also used to create an object. It calls the default constructor to create the object. It returns a newly created instance of the class represented by the object. It internally uses the newInstance() method of the Constructor class.

Example:

```
public class CreateObjectExample4
{
void show()
{
System.out.println("A New Object Created.");
}
public static void main(String[] args)
{
try
{
//creating an instance of Class class
Class cls = Class.forName("CreateObjectExample4");
//creates an instance of the class using the newInstance() method
CreateObjectExample4 obj = (CreateObjectExample4) cls.newInstance();
//invoking the show() method
obj.show();
}
catch (ClassNotFoundException e)
{
e.printStackTrace();
}
catch (InstantiationException e)
{
e.printStackTrace();
}
catch (IllegalAccessException e)
{
e.printStackTrace();
}
}
}
```

Output:

```
A New Object Created.
```

4.Using newInstance() Method of Constructor class:

It is similar to the **newInstance()** method of the **Class** class. It is known as a reflective way to create objects. The method is defined in the **Constructor** class which is the class of **java.lang.reflect package.** We can also call the parameterized constructor and private constructor by using the **newInstance()** method.

```
import java.lang.reflect.*;
class Parul
{
// Member variables of this class
private String name;
// Constructor of this class
Parul() {}
// Method 1
// To set name of the string
public void setName(String name)
{
// This method refers to current object itself
this.name = name;
}
// Main driver method
public static void main(String[] args)
{
// Try block to check for exceptions
try
{
Constructor<Parul> constructor= Parul.class.getDeclaredConstructor();
Parul r = constructor.newInstance();
// Custom passing
r.setName("PIET @ Parul University");
System.out.println(r.name);
}
// Catch block to handle the exceptions
catch (Exception e)
{
```

```
// Display the exception on console
// using printStackTrace() method
e.printStackTrace();
}
}
}
```

Output:

```
PIET @ Parul University
```

5.Using deserialization

Whenever we serialize and then deserialize an object, JVM creates a separate object. In **deserialization**, JVM doesn't use any constructor to create the object. To deserialize an object we need to implement the Serializable interface in the class.

Serialization

```
import java.io.*;
class Persist
{
public static void main(String args[])
{
Try
{
//Creating the object
Student s1 =new Student(211,"ravi");
//Creating stream and writing the object
FileOutputStream fout=new FileOutputStream("f.txt");
ObjectOutputStream out=new ObjectOutputStream(fout);
out.writeObject(s1);
out.flush();
//closing the stream
out.close();
System.out.println("success");
}
catch(Exception e)
{
System.out.println(e);
}
}
}
```

Output:

```
Success
```

Deserialization:

```
import java.io.*;
class Depersist
{
public static void main(String args[])
{
Try
{
//Creating stream to read the object
ObjectInputStream in=new ObjectInputStream(new FileInputStream("f.txt"));
Student s=(Student)in.readObject();
//printing the data of the serialized object
System.out.println(s.id+" "+s.name);
//closing the stream
in.close();
}
catch(Exception e)
{
System.out.println(e);
}
}
}
```

Output:

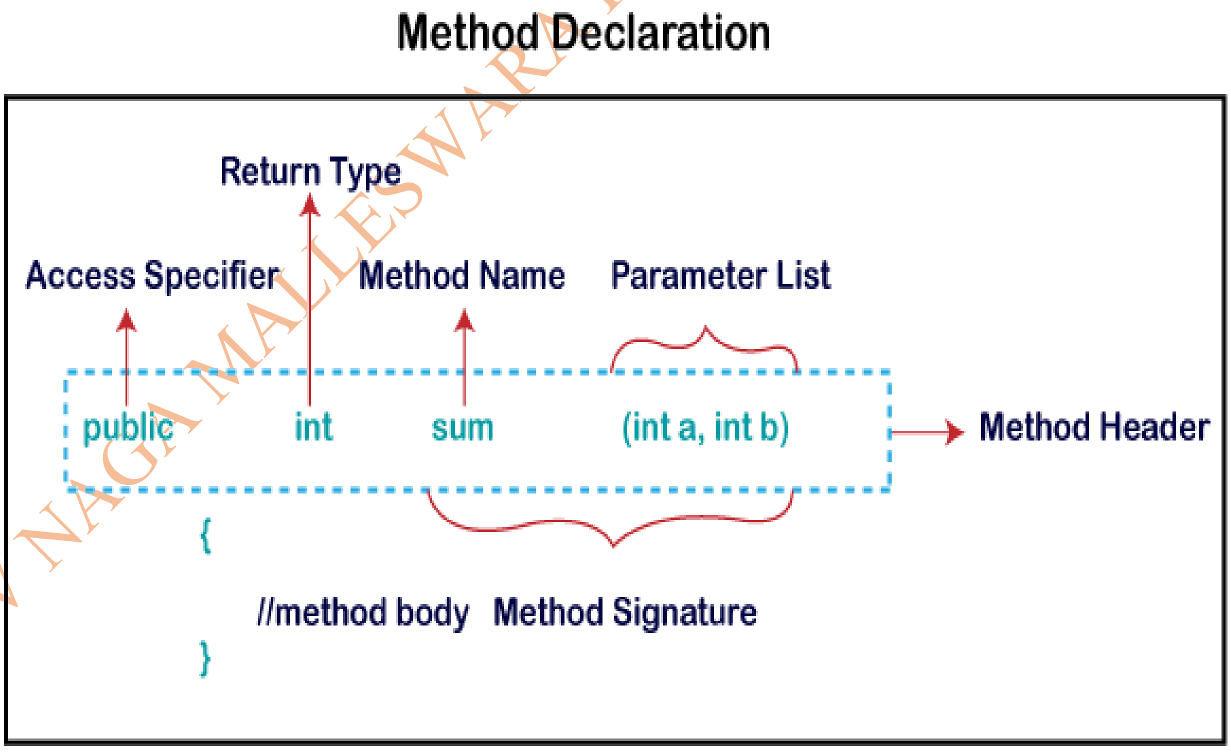
```
111
Ram
```

What is a method in Java?

- A **method** is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation.
- It is used to achieve the **reusability** of code.
- We write a method once and use it many times. We do not require to write code again and again.
- It also provides the **easy modification** and **readability** of code, just by adding or removing a chunk of code.
- The method is executed only when we call or invoke it.
- The most important method in Java is the **main()** method.

Method Declaration:

The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments. It has six components that are known as **method header**, as we have shown in the following figure.



Method Signature:

Every method has a method signature. It is a part of the method declaration. It includes the **method name** and **parameter list**.

Access Specifier:

Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides **four** types of access specifier:

- **Public:** The method is accessible by all classes when we use public specifier in our application.
- **Private:** When we use a private access specifier, the method is accessible only in the classes in which it is defined.
- **Protected:** When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.
- **Default:** When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.

Return Type:

Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.

Method Name:

It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be **subtraction().** A method is invoked by its name.

Parameter List:

It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.

Method Body:

It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

What is a method and explain different types of Methods in Java

There are two types of methods in Java:

- Predefined Method
- User-defined Method

Predefined Method:

- In Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods.
- It is also known as the **standard library method** or **built-in method**.
- We can directly use these methods just by calling them in the program at any point.
- Some pre-defined methods are **length(), equals(), compareTo(), sqrt(),** etc. When we call any of the predefined methods in our program, a series of codes related to the corresponding method runs in the background that is already stored in the library.
- Each and every predefined method is defined inside a class.
- Such as **print()** method is defined in the **java.io.PrintStream** class.
- It prints the statement that we write inside the method.
- For example, **print("Java")**, it prints Java on the console

Example:1

```
public class mul
{
    public static void main(String[] args)
      {
        int x=5;
        System.out.println(x*1);
        System.out.println(x*2);
        System.out.println(x*3);
        System.out.println(x*4);
        System.out.println(x*5);
        System.out.println(x*6);
        System.out.println(x*7);
        System.out.println(x*8);
        System.out.println(x*9);
        System.out.println(x*10);
    }
}
```

Example:2

```
public class mul
{
static int mul(int a)
{
System.out.println(5*a);
return 5*a;
}
public static void main(String[] args)
{
for(int i=1;i<=10;i++)
{
mul(i);
}
}
}
```

User-defined Method:

The method written by the user or programmer is known as **a user-defined** method. These methods are modified according to the requirement.

How to Create a User-defined Method

Let's create a user defined method that checks the number is even or odd. First, we will define the method.

```
//user defined method
public static void findEvenOdd(int num)
{
//method body
if(num%2==0)
System.out.println(num+" is even");
else
System.out.println(num+" is odd");
}
```

- We have defined the above method named findevenodd().
- It has a parameter **num** of type int.
- The method does not return any value that's why we have used void.
- The method body contains the steps to check the number is even or odd.
- If the number is even, it prints the number **is even**, else prints the number **is odd**.

Example:

```
import java.util.Scanner;
public class EvenOdd
{
public static void main (String args[])
{
//creating Scanner class object
Scanner scan=new Scanner(System.in);
System.out.print("Enter the number: ");
//reading value from the user
int num=scan.nextInt();
//method calling
findEvenOdd(num);
}
}
```

Output :

```
Enter the number: 12
12 is a even number
```

Types:

1.Static Methods: it is implement the behaviour of the class.

2. Instance Methods (Non-Static): it is used to implement behaviour of each instance of class. (create with Object)

Static Method:

- A static method in Java is a method that is part of a class rather than an instance of that class.
- Every instance of a class has access to the method.
- Static methods have access to class variables (static variables) without using the class's object (instance).
- Only static data may be accessed by a static method

- We can also create a static method by using the keyword **static** before the method name.
- The main advantage of a static method is that we can call it without creating an object.
- It can access static data members and also change the value of it. It is used to create an instance method.
- It is invoked by using the class name. The best example of a static method is the **main()** method.

Example of static method

```
public class Display
{
public static void main(String[] args)
{
show();
}
static void show()
{
System.out.println("It is an example of static method.");
}
}
```

Output:

```
It is an example of static method
```

Non-Static Example:

```
class Student
{
int rollno;
String name;
void insertRecord(int r, String n)
{
rollno=r;
name=n;
}
void displayInformation()
{
System.out.println(rollno+" "+name);
}
}
class TestStudent4
{
public static void main(String args[])
{
Student s1=new Student();
Student s2=new Student();
s1.insertRecord(111,"William");
System.out.println(" \n");
s2.insertRecord(222,"Smith");
s1.displayInformation();
s2.displayInformation();
}
}
```

Output:

```
111     William
222     Smith
```

```
import java.io.*;
class Geek
{
public static String geekName = "";
public static void geek(String name)
{
geekName = name;
}
}
class GFG
{
public static void main(String[] args)
{
// Accessing the static method geek()
// and field by class name itself.
Geek.geek("Abhinav");
System.out.println(Geek.geekName);
// Accessing the static method geek()
// by using Object's reference.
Geek obj = new Geek();
obj.geek("Amith");
System.out.println(obj.geekName);
}
}
```

Output:

```
Abhinav
Amith
```

Method Overloading:

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.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int, int) for two parameters, and b(int, int, int) for three parameters

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

1.Method Overloading: changing no. of arguments:

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

```
class Adding
{
static int add(int a, int b)
{
return a+b;
}
static int add(int a, int b, int c)
{
return a+b+c;
}
}
class TestOverloading1
{
public static void main(String[] args)
{
System.out.println(Adding.add(100,200));
System.out.println(Adding.add(100,200,300));
}
}
```

Output:

```
300

600
```

2.Method Overloading: changing data type of arguments

In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives three double arguments.

```
class Adding
{
static int add(int a, int b)
{
return a+b;
}
static double add(double a, double b, double c)
{
return a+b+c;
}
}
class TestOverloading2
{
public static void main(String[] args)
{
System.out.println(Adding.add(100,200));
System.out.println(Adding.add(100.55,200.55,300.33));
}
}
```

Output:

```
300
601.4300000000001
```

Method Overriding:

“If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.”

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

In other words, if a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Rules for Java Method Overriding

- The method must have the same name as in the parent class
- The method must have the same parameter as in the parent class.
- There must be an IS-A relationship (inheritance).

Understanding the problem without method overriding

```
    //Java Program to demonstrate why we need method overriding
    //Here, we are calling the method of parent class with child
    //class object.
    //Creating a parent class
class Vehicle
{
void run()
{
System.out.println("Vehicle is running");
}
}
class Bike extends Vehicle  //Creating a child class
{
public static void main(String args[])
{
Bike obj = new Bike();  //creating an instance of child class
obj.run();  //calling the method with child class instance
}
}
```

Output:

```
Vehicle is running
```

//Java Program to illustrate the use of Java Method Overriding
//Creating a parent class.

Example:

```
class Vehicle
{
//defining a method
void run()
{
System.out.println("Vehicle is running");
}
}
//Creating a child class
class Bike2 extends Vehicle
{
//defining the same method as in the parent class
void run()
{
System.out.println("Bike Is Running Safely");
}
public static void main(String args[])
{
Bike2 obj = new Bike2();//creating object
obj.run();//calling method
}
}
```

Output:

```
Bike Is Running Safely
```

Constructor:

A constructor in java is similar to a method that is invoked when an object of the class is created. Unlike Java methods, a constructor has the same name as that of the class and does not have any return type.

Types of Java constructors

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

Default Constructor:

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

```
class Bike1
{

Bike1()  //creating a default constructor
{
System.out.println("Bike is created");
}
//main method
public static void main(String args[])
{
//calling a default constructor
Bike1 b=new Bike1();
}
}
```

Output:

```
Bike is created
```

1.Parameterized Constructor:

A constructor which has a specific number of parameters is called a parameterized constructor. The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Example:

```
class Student4
{
int id;
String name;
Student4(int i, String n)  //creating a parameterized constructor
{
id = i;
name = n;
}
void display()  //method to display the values
{
System.out.println(id+" "+name);
}
public static void main(String args[])
{
Student4 s1 = new Student4(111,"Karan");  //creating objects and passing values
Student4 s2 = new Student4(222,"Aryan");  //calling method to display the values of object
Student4 s3 = new Student4(333,"Sony");
Student4 s4 = new Student4(444,"Mona");
s1.display();
s2.display();
s3.display();
s4.display();
}
}
```

Output:

```
111 Karan
222 Aryan
333 Sony
444 Mona
```

2.Constructor overloading in java:

The constructor overloading can be defined as the concept of having more than one constructor with different parameters list, so that every constructor can perform a different task.

```
public class Student5
{
    int id;
    String name;
    int age;
    Student5(int i, String n)  //creating two arg constructor-1
    {
    id = i;
    name = n;
    }
    Student5(int i, String n, int a)  //creating three args constructor  -2
   {
    id = i;
    name = n;
    age=a;
    }
    void display()
    {
    System.out.println(id+" "+name+" "+age);
    }
    public static void main(String args[])
   {
    Student5 s1 = new Student5(111,"Sonu");
    Student5 s2 = new Student5(222,"Monu",25);
    s1.display();
    s2.display();
   }
  }
```

Output:

```
111 Sonu
222 Monu 25
```

Note: Class name and Method name both are same

Difference between constructor and method in Java

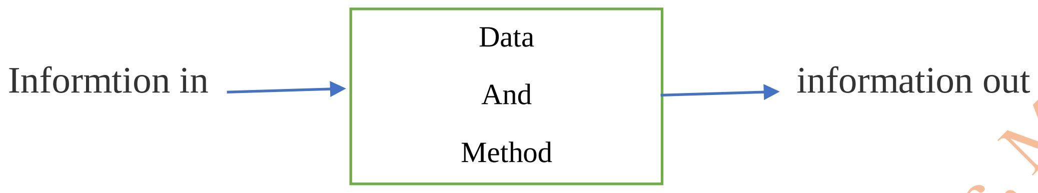
There are many differences between constructors and methods. They are given below.

Java Constructor	Java Method
A constructor is used to initialize the state of an object.	A method is used to expose the behavior of an object.
A constructor must not have a return type.	A method must have a return type.
The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor in a class.	The method is not provided by the compiler in any case.
The constructor name must be same as the class name.	The method name may or may not be same as the class name.

Encapsulation:

Wrapping up data and methods into a single unit is known as Encapsulation. The data not accessible from outside the world and only access those methods, which are wrapped in the class, it can access it.

Methods are interface between the object data and program.



Example:

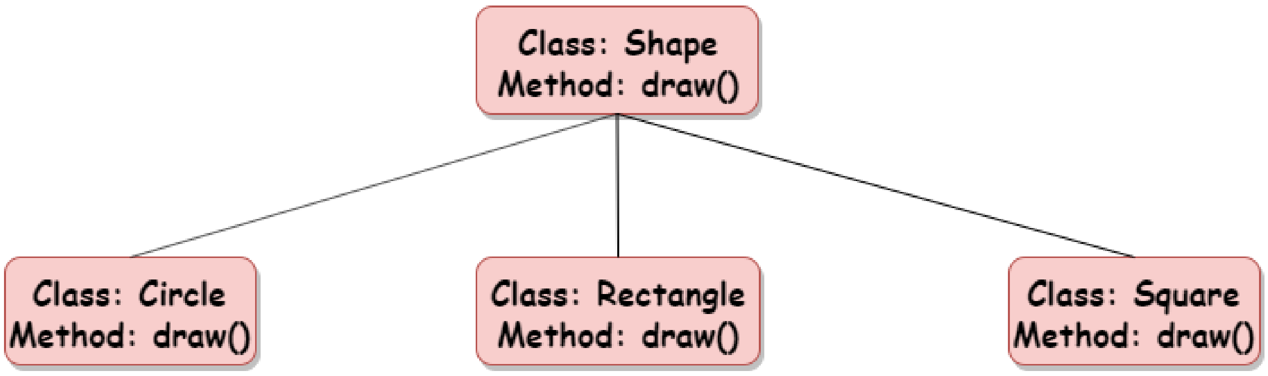
```
class Area
{
int length;
int breadth;  // fields to calculate area

Area(int length, int breadth) // constructor to initialize values
{
this.length = length;
this.breadth = breadth;
}
// method to calculate area
public void getArea()
{
int area = length * breadth;
System.out.println("Area: " + area);
}
}
class Main
{
public static void main(String[] args)
{
// create object of Area
// pass value of length and breadth
Area rectangle = new Area(5, 6);
rectangle.getArea();
}
}
```

Output:

```
30
```

Polymorphism:

Polymorphism is another important opps concepts, polymorphism means it is ability to take more than one form, an operation may exhibit different behaviour in different instances. The behaviour depends upon the types of data used in the operation.



Example:

```
class Shape
{
void draw()  // Method //
{
System.out.println("drawing...");
}
}
class Circle extends Shape
{
void draw()  // Method //
{
System.out.println("drawing Circle...");
}
}
class Rectangle extends Shape
{
void draw()
{
System.out.println("drawing Rectangle...");
}
}
```

```
class Square extends Shape
{
void draw()
{
System.out.println("drawing Square…. ");
}
}
class TestPolymorphism2
{
public static void main(String args[])
{
Shape s;
s=new Circle();
s.draw();
s=new Rectangle();
s.draw();
s=new Square();
s.draw();
}
}
```

Output:

```
drawing Circle...

drawing Rectangle...

drawing Square……
```

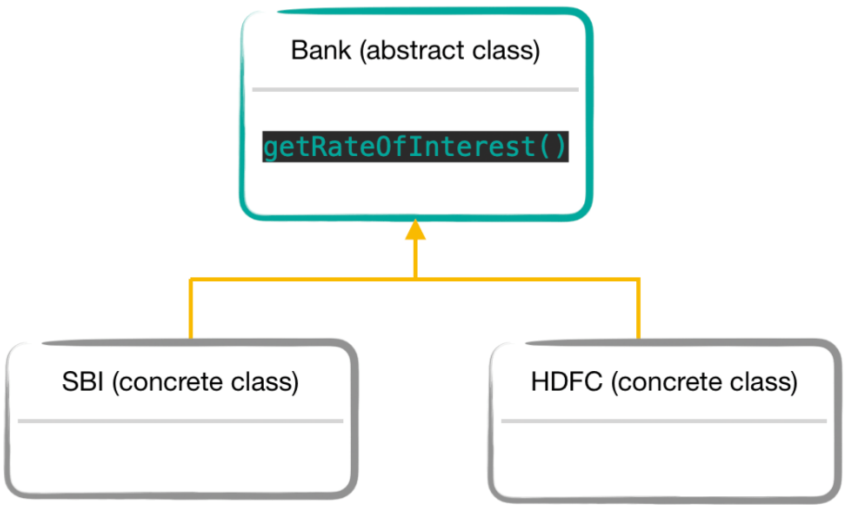
Abstraction:

Abstraction refers to the act of representing essential feature without including the background details or explanation. Classes use the abstraction concepts and defined a list of abstract attributes.

Example:

for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Example:



```
abstract class Bank
{
abstract int getRateOfInterest();
}
class SBI extends Bank
{
int getRateOfInterest()
{
return 5;
}
}
class HDFC extends Bank
{
int getRateOfInterest()
{
return 9;
}
}
class TestBank
{
public static void main(String args[])
{
Bank b;
b=new SBI();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
b=new HDFC();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
}
}
```

Output:

```
SBI       Rate of Interest is: 5 %
HDFC   Rate of Interest is: 9 %
```

Inheritance:

Inheritance is an important pillar of OOP (Object-Oriented Programming). It is the mechanism in java by which one class is allowed to inherit the features (fields and methods) of another class. In Java, inheritance means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and fields of that class.

Inheritance in Java: Why do we need it?

- **Code Reusability:** The code written in the Superclass is common to all subclasses. Child classes can directly use the parent class code.
- **Method Overriding:** Method Overriding is achievable only through Inheritance. It is one of the ways by which java achieves Run Time Polymorphism.
- **Abstraction:** The concept of abstract where we do not have to provide all details is achieved through inheritance. Abstraction only shows the functionality to the user.

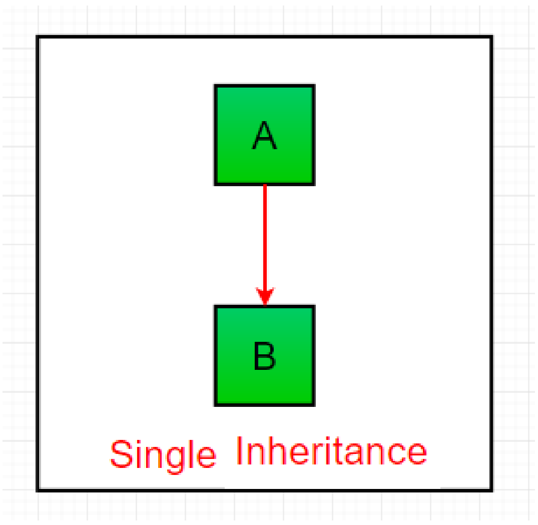
Types of inheritance in java:

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.

Single Inheritance Example:

When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.



Example:

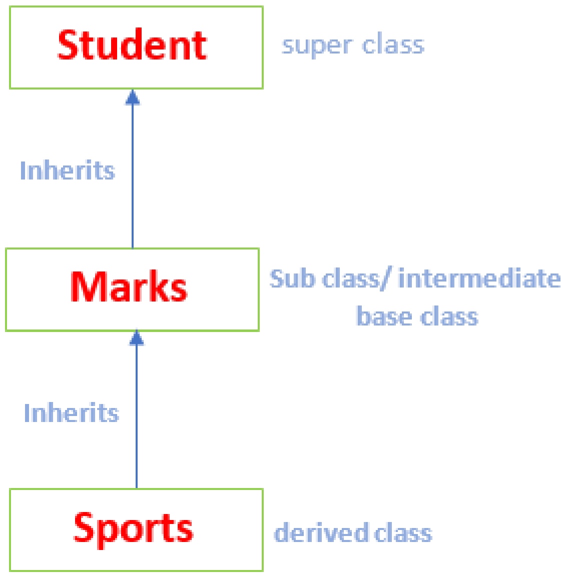
```
class Employee
{
float salary=40000;
}
class Programmer extends Employee
{
int bonus=10000;
public static void main(String args[])
{
Programmer p=new Programmer();
System.out.println("Programmer salary is:"+p.salary);
System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

Output:

```
Programmer salary is:40000.0

Bonus of Programmer is:10000
```

Multilevel Inheritance Example

When there is a chain of inheritance, it is known as *multilevel inheritance*. Multilevel Inheritance in java involves inheriting a class, which already inherited some other class. Multilevel Inheritance in Java is a type of inheritance in which a class that is already inherited by another class, inherits another class.



Multi-level Inheritance

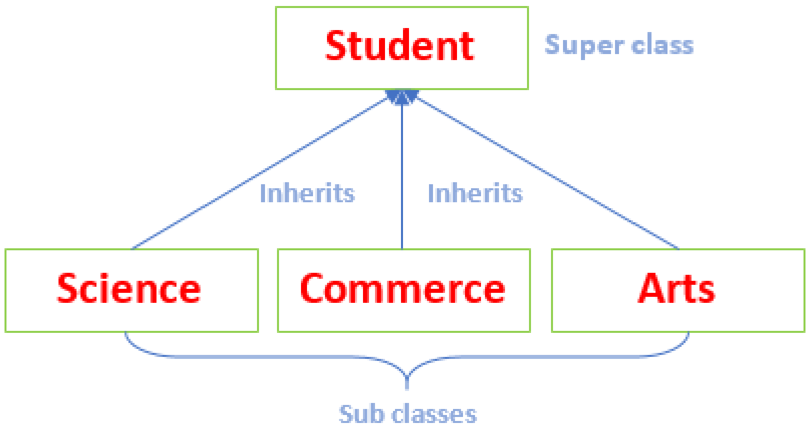
```
class Student
{
int reg_no;
void getNo(int no)
{
reg_no=no;
}
void putNo()
{
System.out.println("registration number= "+reg_no);
}
}
//intermediate sub class
class Marks extends Student
{
float marks;
void getMarks(float m)
{
marks=m;
}
void putMarks()
{
System.out.println("marks= "+marks);
}
}
class Sports extends Marks  //derived class
{
float score;
void getScore(float scr)
{
score=scr;
}
void putScore()
{
System.out.println("score= "+score);
}
}
class MultilevelInheritanceExample
{
public static void main(String args[])
{
Sports ob=new Sports();
ob.getNo(123);
ob.putNo();
ob.getMarks(78);
ob.putMarks();
ob.getScore(68);
ob.putScore();
}
}
```

Output:

```
registration number= 123
marks= 78.0
score= 68.0
```

Hierarchical Inheritance:

When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below.



Hierarchical Inheritance

Syntax:

```
Class Parent
{
        //Data members and member functions of Parent Class
}
Class Derived-1 : access modifier Parent
{
        //Data members and member functions of Derived-1 class
}
Class Derived-2 : access modifier Parent
{
        //Data members and member functions of Derived-2 class
}
```

Example:

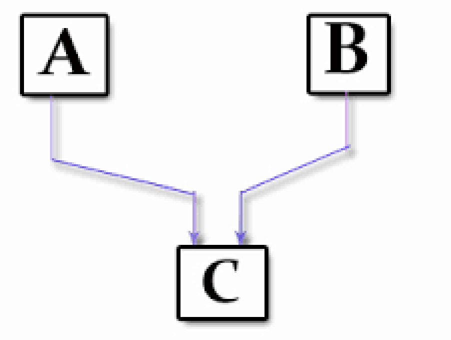
```
class Student
{
public void methodStudent()
{
System.out.println("The method of the class Student invoked.");
}
}
class Science extends Student
{
public void methodScience()
{
System.out.println("The method of the class Science invoked.");
}
}
class Commerce extends Student
{
public void methodCommerce()
{
System.out.println("The method of the class Commerce invoked.");
}
}
class Arts extends Student
{
public void methodArts()
{
System.out.println("The method of the class Arts invoked.");
}
}
class HierarchicalInheritanceExample
{
public static void main(String args[])
{
Science sci = new Science();
Commerce comm = new Commerce();
Arts art = new Arts();
//all the sub classes can access the method of super class
sci.methodStudent();
comm.methodStudent();
art.methodStudent();
}
}
```

Output:

```
The method of the class Student invoked.

The method of the class Student invoked.

The method of the class Student invoked.
```

What is a multiple inheritance:

When a class inherits properties from many classes, this is known as multiple inheritance. Multiple Inheritance is not supported by Java. But using Interfaces, Multiple Inheritance is possible in Java.



MULTIPLE INHERITANCE

Example:

```
interface A
{
public abstract void execute1();
}
interface B
{
public abstract void execute2();
}
class C implements A, B
{
public void execute1()
{
System.out.println("Hai... I am from execute1");
}
public void execute2()
{
System.out.println("Hai…. I am from execute2");
}
}
class Main
{
public static void main(String[] args)
{
C obj = new C(); // creating object of class C
obj.execute1(); //calling method execute1
obj.execute2(); // calling method execute2
}
}
```

Output:

```
Hai... I am from execute1
Hai…. I am from execute2
```

Note:

- Abstract keyword is a non-access modifier, it is used for class and methods.

  Abstract class:

- Abstract class is a restricted class that can’t be used to create object (to access, it must be inherit from another class).

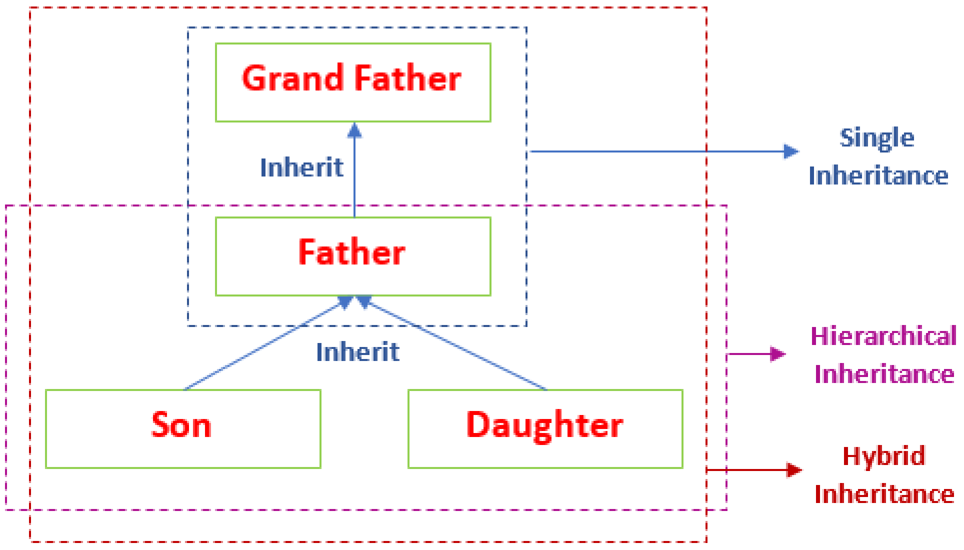
  Abstract method:

- It is only used in an abstract class and it does not have a body.

Hybrid inheritance:

The hybrid inheritance is the composition of two or more types of inheritance. The main purpose of using hybrid inheritance is to modularize the code into well-defined classes. It also provides the code reusability.

- Single and Multiple Inheritance (not supported but can be achieved through interface)
- Multilevel and Hierarchical Inheritance
- Hierarchical and Single Inheritance
- Multiple and Multilevel Inheritance.



Hybrid Inheritance

```
class GrandFather
{
public void show()
{
System.out.println("I am grandfather.");
}
}
//inherits GrandFather properties
class Father extends GrandFather
{
public void show()
{
System.out.println("I am father….");
}
}
//inherits Father properties
class Son extends Father
{
public void show()
{
System.out.println("I am son….");
}
}
//inherits Father properties
class Daughter extends Father
{
public void show()
{
System.out.println("I am a daughter….");
}
public static void main(String args[])
{
Daughter obj = new Daughter();
obj.show();
}
}
```

Output: I am a daughter.

Super keyword:

- The super keyword invokes immediately Super Class members.
- The super keyword refers to super class objects.
- It is used to call super class methods and to access the super class constructor.
- The main common use of super keyword is to eliminate the confusion super class and subclass, that have the methods with the same name.
- It is majorly used in the following contexts:

- Use of super with variables
- Use of super with methods
- Use of super with constructors

1. Use of super with variables:

This scenario occurs when a derived class and base class has the same data members. In that case, there is a possibility of ambiguity for the JVM. We can understand it more clearly using this code

Example:

```
class Vehicle
{
int maxSpeed = 120;
}
class Bike extends Vehicle // sub class Car extending vehicle
{
int maxSpeed = 180;
void display()
{
System.out.println("Maximum Speed: " + super.maxSpeed);
}                          // print maxSpeed of base class (vehicle)
}
class MyBike // Driver Program
{
public static void main(String[] args)
{
Bike Bullet = new Bike ();
Bullet.display();
}
}
```

Output:

```
Maximum Speed: 120
```

2. Use of super with methods:

This is used when we want to call the parent class method. So, whenever a parent and child class have the same-named methods then to resolve ambiguity we use the super keyword. This code snippet code(source code-it will return same results) helps to understand the said usage of the super keyword.

Example:

```
class Person
{
void message()
{
System.out.println("This is person class\n");
}
}
class Student extends Person // Subclass Student
{
void message()
{
System.out.println("This is student class");
}
// Note that display() is
// only in Student class
void display()
{
// will invoke or call current
// class message() method
message();
// will invoke or call parent
// class message() method
super.message();
}
}
class Test // Driver Program
{
public static void main(String args[])
{
Student st = new Student();
st.display();  // calling display() of Student
}
}
```

Super class& Subclass has same method

Output:

```
This is student class
This is person class
```

3. Use of super with constructors:

The super keyword can also be used to access the parent class constructor. One more important thing is that ‘super’ can call both parametric as well as non-parametric constructors depending upon the situation. Following is the code snippet to explain the above concept:

Example:

```
class Person
{
Person()
{
System.out.println("Person class Constructor");
}
}
// subclass Student extending the Person class
class Student extends Person
{
Student()
{
// invoke or call parent class constructor
 super();
System.out.println("Student class Constructor");
}
}
class Test // Driver Program
{
public static void main(String[] args)
{
Student s = new Student();
}
}
```

Output:

```
Person class Constructor
Student class Constructor
```

Note:

A snippet represents a snippet of java source code a passed to JShell. It as associated only with the JShell instance that created it. An instance of snippet (including subclass). it will return same results.