Outline

- Basic Concepts
  - Inheritance
  - Method overloading and overriding
  - Encapsulation
  - Polymorphism
  - Abstract classes and Interfaces

Inheritance

What is Inheritance?

- Inheritance is a mechanism in which **one class acquires** the **property of another class.**
- 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.**
  - In addition, **you can add new fields and methods** to your current class as well.
- **Inheritance in Java** is a mechanism in which **one object acquires all the properties and behaviors of a parent object.**

Inheritance

What is Inheritance?

- **The class** which **inherits the properties of other** is known as **subclass** (**derived class, child class**)
- **The class** whose **properties are inherited** is known as **superclass (base class, parent class).**
- Therefore, **a subclass is a specialized version of a superclass**
  - It inherits all of the instance variables and methods defined by the superclass and add its own, unique elements.
- Inheritance represents **the IS-A relationship** which is also known as **a parent-child relationship.**

Inheritance

Terms used in Inheritance

- **Super class and base class** are synonyms of **Parent class.**
- **Sub class** and **derived class** are synonyms of **Child class.**
- **Properties and fields** are synonyms of **Data members**.
- **Functionality** is a synonym of **method**.

Inheritance

What is extends Keyword?

- **extends** is the **keyword used to inherit the properties of a class.**
- **The syntax of Java Inheritance**

```
class Super
{
    .....
    .....
}
class Sub extends Super
{
    .....
    .....
}
```

Example 1

```
class Calculation {
    int z;
    public void addition(int x, int y) {
        z = x + y;
        System.out.println(z);}
    public void Subtraction(int x, int y) {
        z = x - y;
        System.out.println(z);}}
public class My_Calculation extends Calculation {
    public void multiplication(int x, int y) {
        z = x * y;
        System.out.println(z);}
    public static void main(String args[]) {
        int a = 20, b = 10;
        My_Calculation demo = new My_Calculation();
        demo.addition(a, b);
        demo.Subtraction(a, b);
        demo.multiplication(a, b);}}
```

Inheritance

- The Output of **Example 1** is
  30
  10
  200

Example 2

```
class Teacher {
  String job = "Teacher";
  String collegeName = "ACT";
  void does(){
    System.out.println("Teaching");}}
public class PhysicsTeacher extends Teacher{
  String mainSubject = "Physics";
  public static void main(String args[]){
    PhysicsTeacher obj = new PhysicsTeacher();
    System.out.println(obj.collegeName);
    System.out.println(obj.job);
    System.out.println(obj.mainSubject);
    obj.does();
  }
}
```

Inheritance

- The Output of **Example 2** is
  ACT
  Teacher
  Physics
  Teaching

Example 3

```
class A {
int i;
private int j;
void setij(int x, int y) {
i = x;
j = y;}}
class B extends A {
int total;
void sum() {
total = i + j; }
class Access {
public static void main(String args[]) {
B subOb = new B();
subOb.setij(10, 12);
subOb.sum();
System.out.println("Total is " + subOb.total);
}}}
```

Inheritance

- The Output of **Example 3** is
  error: j has private access in A

Super Keyword in Java

- The **super keyword** in Java is used in **subclasses to access superclass members (attributes, constructors and methods).**
- The **super keyword** in Java is a reference variable which is used to **refer immediate parent class object.**
- The **super keyword** refers to **superclass (parent) objects.**

Super Keyword in Java

- Usage of Java super Keyword objects.
  1. super can be used to refer immediate parent class instance variable.
  2. super can be used to invoke immediate parent class method
  3. super() can be used to invoke immediate parent class constructor.

Super Keyword in Java

- super is used to refer **immediate parent class instance variable.**
  - We can use **super keyword** to access the **data member or field of parent class**.
  - It is used **if parent class and child class have same fields.**

Example 1

```
class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);
System.out.println(super.color);
}
}
class TestSuper1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}}
```

Inheritance

- The Output of **Example 1** is
  black
  white

Example 2

```
class Vehicle {
    int maxSpeed = 120;
}
class Car extends Vehicle {
    int maxSpeed = 180;
    void display()
    {
        System.out.println("Maximum Speed: "+ super.maxSpeed);
    }
}
class Test {
    public static void main(String[] args)
    {
        Car small = new Car();
        small.display();
    }}
```

Inheritance

- The Output of **Example 2** is
  Maximum Speed: 120

Super Keyword in Java

- super can be used to **invoke parent class method**
  - The super keyword can also be used to **invoke parent class method.**
  - It should be used **if subclass contains the same method as parent class.**

Inheritance

Example 1

```
class Animal{
void eat()
{
System.out.println("eating...");} }
class Dog extends Animal
{
void bark(){System.out.println("barking...");}
void eat(){System.out.println("eating bread...");}
void work()
{
super.eat();
eat();
}
}
class TestSuper2{
public static void main(String args[]){
Dog d=new Dog();
d.work();  }}
```

Inheritance

- The Output of **Example 1** is
  eating...
  eating bread...

Super Keyword in Java

- super is used to invoke parent class constructor.
  - The super keyword can also be used to invoke the parent class constructor.

Example 1

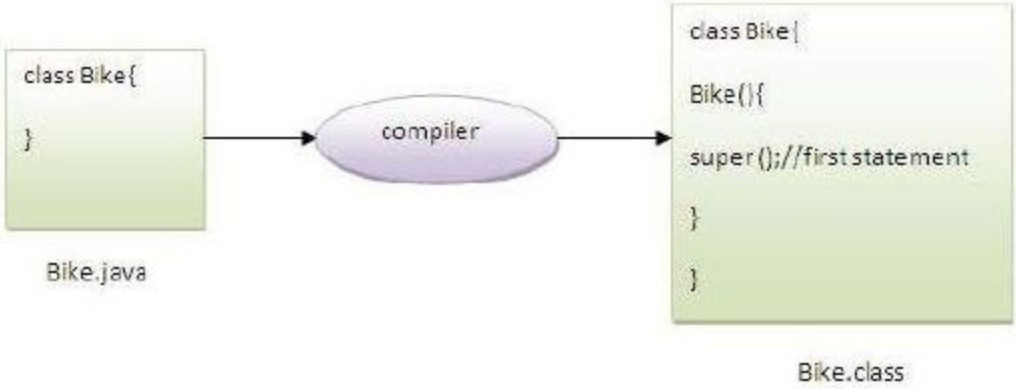
```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");
}
}
class TestSuper3{
public static void main(String args[]){
Dog d=new Dog();
}}
```

Inheritance

- The Output of **Example 1** is
  animal is created
  dog is created

Super Keyword in Java

- **Note: super() is added in each class constructor automatically by compiler if there is no super()**



Example 2

```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
System.out.println("dog is created");
}
}
class TestSuper4{
public static void main(String args[]){
Dog d=new Dog();
}}
```

Inheritance

- The Output of **Example 2** is
  animal is created
  dog is created

Example 3

```
class Person {
    void message(){
        System.out.println("This is person class\n");}}
class Student extends Person {
    void message() {
        System.out.println("This is student class");}
    void display(){
        message();
        super.message();}}
class Test {
    public static void main(String args[])
    {
        Student s = new Student();
        s.display();
    }
}
```

Inheritance

- The Output of **Example 3** is
  This is student class
  This is person class

Example 4

```
class Animal {
  Animal() {
    System.out.println("I am an animal");}
  Animal(String type) {
    System.out.println("Type: "+type);}}
class Dog extends Animal {
  Dog() {
    super("Animal");
    System.out.println("I am a dog"); }}
class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
  }
}
```

Inheritance

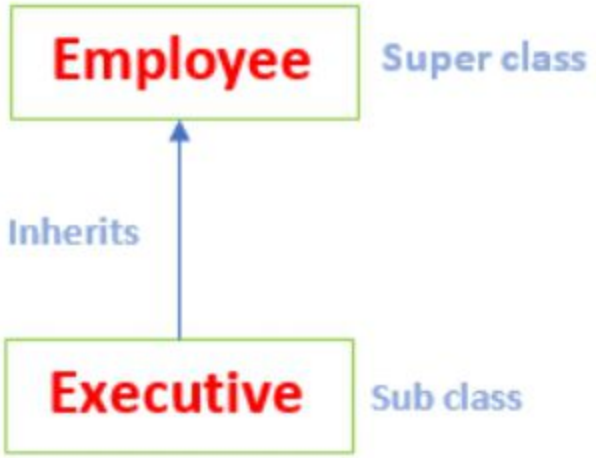
- The Output of **Example 3** is
  Type: Animal
  I am a dog

Types of Inheritance in Java

- Java supports the following four types of inheritance:
  1. **Single Inheritance**
  2. **Multi-level Inheritance**
  3. **Hierarchical Inheritance**
  4. **Hybrid Inheritance**

Single Inheritance

- In **single inheritance**, a sub-class is **derived from only one super class**.
- It inherits the properties and behavior of a **single-parent class**.
- Sometimes it is also known **as simple inheritance.**



Single Inheritance

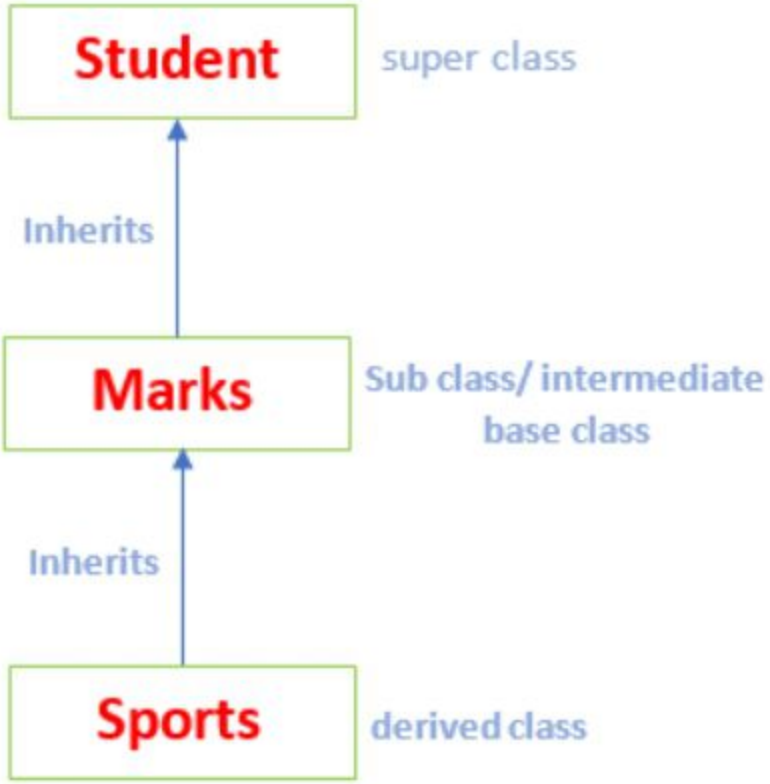
Example 1

```
class Employee
{
float salary=34534*12;
}
public class Executive extends Employee
{
float bonus=3000*6;
public static void main(String args[])
{
Executive obj=new Executive();
System.out.println("Total salary credited: "+obj.salary);
System.out.println("Bonus of six months: "+obj.bonus);
}
}
```

Multi-level Inheritance

- In multi-level inheritance, **a class is derived from a class which is also derived from another class** is called **multi-level inheritance.**
- In simple words, we can say that a class that has more than one parent class is called multi-level inheritance.

Multi-level Inheritance



Example 1

- The class

```
class Student  {
int reg_no;
void getNo(int no)  {
reg_no=no;  }
void putNo()  {
System.out.println("registration number= "+reg_no);  }  }
class Marks extends Student  {
float marks;
void getMarks(float m)  {
marks=m;  }
void putMarks()  {
System.out.println("marks= "+marks);  }  }
class Sports extends Marks  {
float score;
void getScore(float scr)  {
score=scr;  }
void putScore()  {
System.out.println("score= "+score);  }  }
```

Example 1

- **The main method**

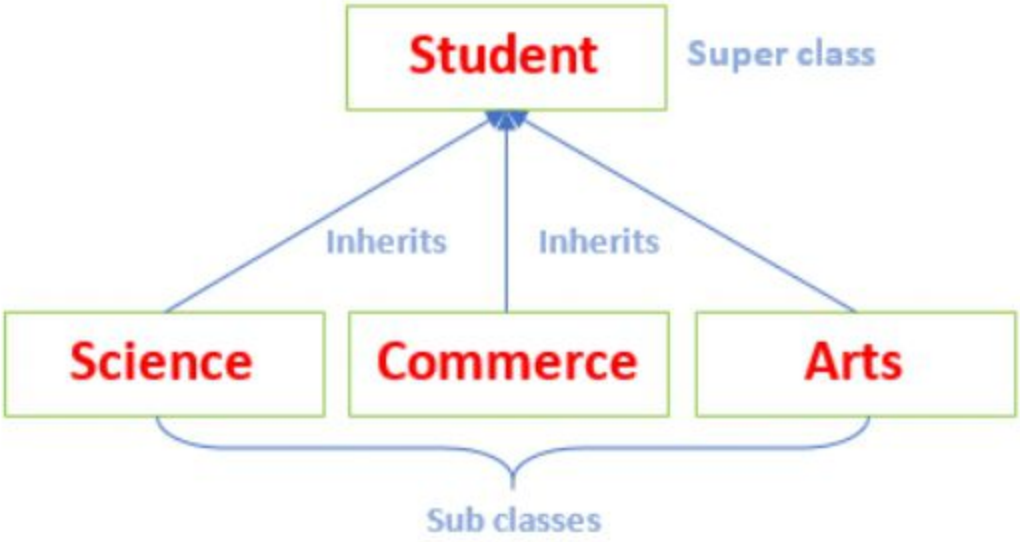
```
public class MultilevelInheritanceExample   {
public static void main(String args[])  {
Sports ob=new Sports();
ob.getNo(987);
ob.putNo();
ob.getMarks(78);
ob.putMarks();
ob.getScore(68.7);
ob.putScore();  }  }
```

Inheritance

- The Output of **Example 1** is
  registration number= 987
  marks= 78.0
  score= 68.7

Hierarchical Inheritance

- When **two or more classes** inherits **a single class**, it is known **as hierarchical inheritance.**
- **If a number of classes** are derived from **a single base class**, it is called **hierarchical inheritance.**



Example 1

- **The class**

```
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.");
} }
```

Inheritance

- The Output of **Example 1** is
  The method of the class Student invoked.
  The method of the class Student invoked
  The method of the class Student invoked.

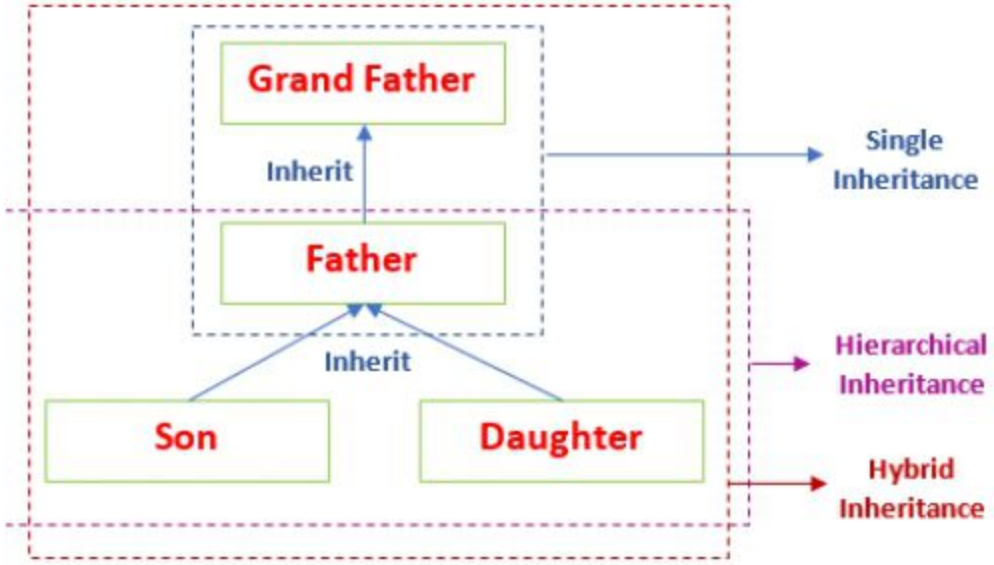
Example 1

- **The Main method**

```
public class HierarchicalInheritanceExample
{
public static void main(String args[])
{
Science sci = new Science();
Commerce comm = new Commerce();
Arts art = new Arts();
sci.methodStudent();
comm.methodStudent();
art.methodStudent();
}
}
```

Hybrid Inheritance

- Hybrid means **consist of more than one**.
- Hybrid inheritance is the **combination of two or more types of inheritance.**



Example 1

```
class GrandFather {
public void show() {
System.out.println("I am grandfather."); } }
class Father extends GrandFather {
public void show() {
System.out.println("I am father."); } }
class Son extends Father {
public void show() {
System.out.println("I am son."); } }
public 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(); } }
```

Inheritance

- The Output of **Example 1** is
  I am daughter.

Method overloading and overriding

Method overriding

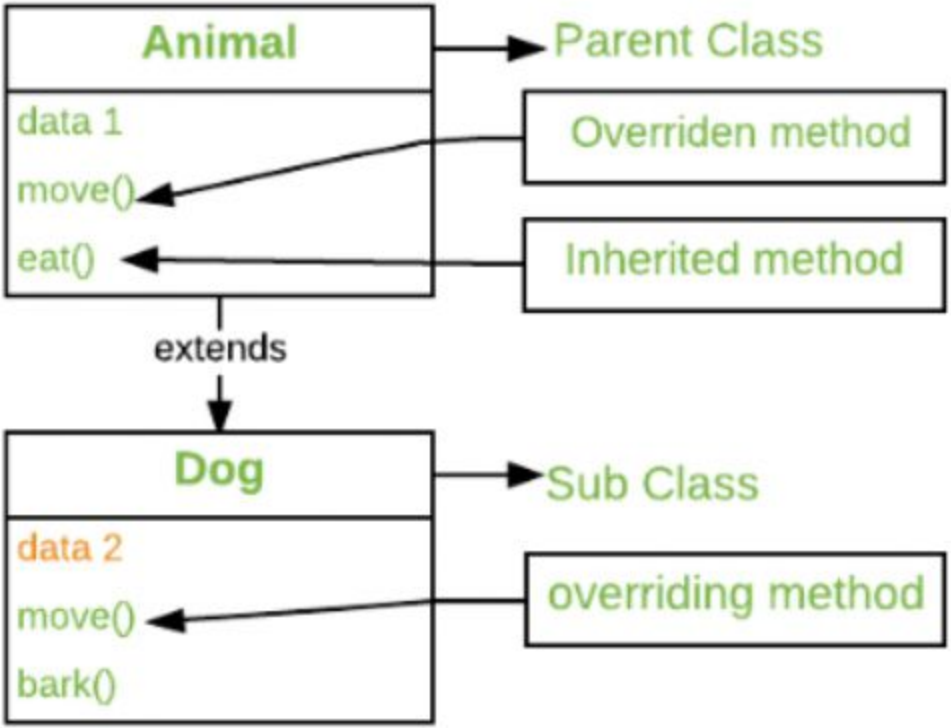
- when **a method in a subclass** has the **same name** and **type signature** as a method in its **superclass**, then the **method in the subclass is said to override the method in the superclass.**
- When an **overridden method is called** from **within a subclass**, **it will always refer to the version of that method defined by the subclass.**
  - The version of the method **defined by the superclass will be hidden.**

Method Overriding

Laws of Method Overriding in JAVA:

1. The method name should be common and the same as it is in the parent class.
2. The method must have the same parameter as in the parent class.
3. There must be an inheritance connection between classes.
4. If it declared the methods as static or final, then those methods cannot be overridden.

Method Overriding



Method Overriding

Example 1

```
class A {
int i, j;
A(int a, int b) {
i = a;j = b;}
void show() {
System.out.println("i and j: " + i + " " + j);}}
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;}
void show() {
System.out.println("k: " + k);}}
class Override {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show();
}}
```

Method Overriding

Example 2

```
class Parent {
    void show(){
        System.out.println("Parent's show()");}}
class Child extends Parent {
    void show() {
        System.out.println("Child's show()");}}
class Main {
    public static void main(String[] args)
    {
        Parent obj2 = new Child();
        obj2.show();
    }
}
```

Method Overriding

Example 1

```
class college {
 public void move() {
  System.out.println("College is open");
 }
}
class univ extends college {
 public void move() {
  System.out.println("University is open too");
 }
}
public class stud {
 public static void main(String args[]) {
  univ a = new college();
  college b = new univ();
  a.move();
  b.move();
 }
}
```

Method overloading and overriding

Method overloading

- **Method Overloading** is a feature that allows **a class to have multiple methods** with the **same name** but **with different number, sequence or type of parameters.**
- In short **multiple methods with same name but with different signatures.**
- **for example**
  - The signature of method **add(int a, int b)** having two int parameters is different from signature of method **add(int a, int b, int c)** having three int parameters.

Method overloading and overriding

Method overloading

- The following method is valid overloading signature

```
float add(int a, float b);
```

```
float add(int a, float b, int c);
```

```
float add(float a, int b);
```

```
float add(float a, float b);
```

Method overloading and overriding

Three ways to overload a method

- In order to overload a method, the parameter list of the methods must differ in either of these:
  1. Number of parameters.
  2. Data type of parameters.
  3. Sequence of Data type of parameters.

Method overloading and overriding

Three ways to overload a method

- **1. Different Number of parameters in signature**

```
class DisplayOverloading
{
  int add(int a, int b){
    int sum = a+b;
    return sum;}
  int add(int a, int b, int c){
    int sum = a+b+c;
    return sum;}}
class JavaExample
{
  public static void main(String args[])
  {
    DisplayOverloading obj = new DisplayOverloading();
    System.out.println(obj.add(10, 20));
    System.out.println(obj.add(10, 20, 30));}}
```

Method overloading and overriding

Three ways to overload a method

- **2. Data type of parameters are different**

```
class DisplayOverloading2{
  public int add(int a, int b){
    int sum = a+b;
    return sum;}
  public float add(float a, float b){
    float sum = a+b;
    return sum;}}
class JavaExample{
  public static void main(String args[]){
    DisplayOverloading2 obj = new DisplayOverloading2();
    System.out.println(obj.add(5, 15));
    System.out.println(obj.add(5.85, 2.58));
  }
}
```

Method overloading and overriding

Three ways to overload a method

- **3. Sequence of Data type of parameters.**

```
class MethodOverload {
public void printParam(char ch, int num)    {
System.out.println("Input character:" + ch + " ; " + "Input Number:" + num
    }
public void printParam(int num, char ch)      {
System.out.println("Input Number:" + num + " ; " + "Input Character:" + ch
    }
}
class Main
{
public static void main(String args[])
   {
MethodOverload obj = new MethodOverload();
obj.printParam(100, 'A');
obj.printParam('A', 100);   }}
```

Method overloading and overriding

```
class JavaExample{
  void disp(int a, double b){
    System.out.println("Method A");
  }
  void disp(int a, double b, double c){
    System.out.println("Method B");
  }
  void disp(int a, float b){
    System.out.println("Method C");
  }
  public static void main(String args[]){
    JavaExample obj = new JavaExample();
    obj.disp(100, 20.67f);
  }
}
```

Method overloading and overriding

Answer the following question?

- **What happen when the return type, method name and argument list same?**

Method overloading and overriding

```
class Demo
{
    public int myMethod(int num1, int num2){
        System.out.println("First myMethod of class Demo");
        return num1+num2; }
    public int myMethod(int var1, int var2){
        System.out.println("Second myMethod of class Demo");
        return var1-var2;}}
class Sample4
{
    public static void main(String args[])
    {
        Demo obj1= new Demo();
        obj1.myMethod(10,10);
        obj1.myMethod(20,12);}}
```

Method overloading and overriding

The output of the above program is

- **It will throw a compilation error: More than one method with same name and argument list cannot be defined in a same class.**

Method overloading and overriding

Answer the following question?

- **What happen when the return type is different. Method name and argument list same?**

Method overloading and overriding

```
class Demo2
{
    public double myMethod(int num1, int num2){
        System.out.println("First myMethod of class Demo");
        return num1+num2; }
    public int myMethod(int var1, int var2){
        System.out.println("Second myMethod of class Demo");
        return var1-var2;}}
class Sample5
{
    public static void main(String args[])
    {
        Demo2 obj2= new Demo2();
        obj2.myMethod(10,10);
        obj2.myMethod(20,12);
    }
}
```

Method overloading and overriding

The output of the above program is

- **It will throw a compilation error: More than one method with same name and argument list cannot be given in a class even though their return type is different. Method return type doesn't matter in case of overloading.**

Getter and Setter in Java

- **Getter** and **Setter** are methods used to **protect your data and make your code more secure.**
- **Getter(Accessor method)**
  - Getters are also called accessors
  - getters:a method that provides **access to the state of an object to be accessed.**
  - A get method signature:
    public returnType getPropertyName()
  - **Syntax they start with get, followed by the name of the variable, with the first letter in upper case:**

Encapsulation

- **Code :**

```
private int price;

    public int getPrice() { //getter method to retrieve price when called
        return price;
    }
```

Getter and Setter in Java

- **Setters are also called mutators.**
  - Mutator method(setters): **a method that modifies an object's state.**
  - A set method signature:
    public void setPropertyName(dataType propertyValue)
  - **Syntax they start with set, followed by the name of the variable, with the first letter in upper case:**

Encapsulation

- **Code :**

```
private int price;

    public void setPrice(int p) { //setter method to update the price when called
        this.price=p;
    }
```

Encapsulation

What is Encapsulation?

- Encapsulation is **a programming technique** that **binds the class members (variables and methods) together** and **prevents them from being accessed by other classes.**
- **Encapsulation** is the techniques of making **the fields in the class private** and providing **access to the fields using public method**.
- If a **fields is decleared private**,it **cannot be accessed by anyone outside the class**

Encapsulation

In Java, there are two steps to implement encapsulation. Following are the steps:

- Use the **access modifier 'private'** to declare the **class member variables.**
- **To access these private member variables and change their values**, we have to provide the **public getter and setter methods respectively.**

Encapsulation

Example 1

```
class Person {
  private int age;
  public int getAge() {
    return age;
  }
  public void setAge(int age) {
    this.age = age;}}
class Main {
  public static void main(String[] args) {
    Person p1 = new Person();
    p1.setAge(24);
    System.out.println("My age is " + p1.getAge());
  }
}
```

Encapsulation

Output of the above example is

- My age is 24

Encapsulation

Example 2

```
class Student {
    private int Student_Id;
    private String name;
    public int getId() {
        return Student_Id; }
    public void setId(int s_id) {
        this.Student_Id = s_id;}
    public String getname() {
        return name;}
    public void setname(String s_name) {
        this.name = s_name;}}
class Main{
    public static void main(String[] args) {
        Student s=new Student();
        s.setId (27);
        s.setname("Feven");
        System.out.println("Student Data:" + "\nStudent ID:" + s.getId()
                           + " Student Name:" + s.getname()); }}
```

Encapsulation

Output of the above example is

- **Student Data:**
  Student ID:27 Student Name:Feven

Encapsulation

Example 3

- **The Class**

```
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;}
```

Example 3

- **Main Method**

```
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());
}
}
```

Encapsulation

Output of the above example is

- **Name : James Age : 20**

Example 4

- **The Class**

```
class Student{
    private String name;
    private int rollno;
     public void setName(String n) {
        this.name = n;}
     public String getName() {
        return name; }
     public void setRollno(int r) {
        if(r <= 50) {
            this.rollno = r; }
        else {
            System.out.println("Roll number is Invalid");}}
    public int getRollno() {
        return rollno; }}
```

Example 4

- **Main Method**

```
public class Main {
    public static void main(String[] args) {
        Student g1 = new Student();
        g1.setName("Meron");
        System.out.println(g1.getName());
        g1.setRollno(51);
        System.out.println(g1.getRollno());  }
}
```

Encapsulation

Output of the above example is

- **Meron**
  Roll number is Invalid
  0

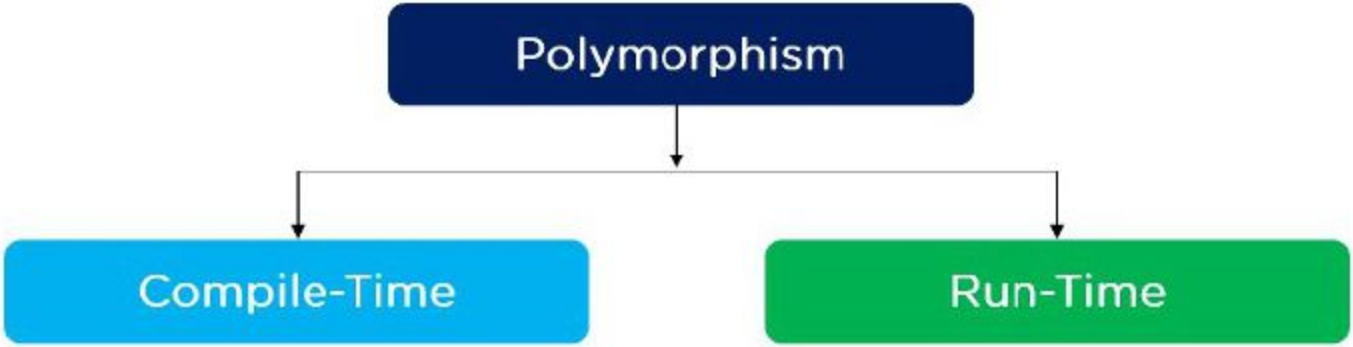
Polymorphism

What is Polymorphism?

- **Polymorphism** is derived from two **Greek words**: **poly** and **morphs**.
  1. **poly:**means **many**
  2. **morphs:** means **forms**
- So **polymorphism means many forms**.
- **Polymorphism** is the **ability of an object to take on many forms**
- In **programming languages** polymorphism is the **capability of an action** or **method to do different things based on the object that it is acting upon.**

Types of Polymorphism?

- There are two types of polymorphism in Java



Compile:Time Polymorphism In Java

- Compile-time polymorphism is also **known as "Static polymorphism".**
- whatever polymorphism is to be performed, is performed **at compile time.**
- the compile-time polymorphism is performed **using "Method Overloading".**
- At compile-time **compiler knows** which method to call by checking **the method signature**.
  - ⋆ So, this is called **complie-time polymorphism or static or early binding.**

Examples of Compile:Time Polymorphism in Java

- **Main Method**

```
class MethodOverload
{
public void printParam(char ch)
    {
System.out.println(ch);
    }
    public void printParam(char ch, int num)
    {
System.out.println("Character: " + ch + " ; "+ "Number:" +num);
    }
}
class Main
{
public static void main(String args[])
   {
MethodOverload obj = new MethodOverload();
obj.printParam('A');
obj.printParam('A',10);  }}
```

Examples of Runtime Polymorphism in Java

Output of the above example is

- **A**
  Character: A ; Number:10

Polymorphism

Run-time polymorphism

- Runtime polymorphism also called **Dynamic Method Dispatch**
- In this process, the call to an overridden method is resolved dynamically at runtime rather than at compile-time.
- You can achieve **Runtime polymorphism via Method Overriding.**
- Method Overriding is done when a child or a subclass has a method with the **same name, parameters, and return type as the parent** or the superclass
- **Java virtual machine** determine the **proper method to call at run time**,not at he compile time.
  - ⋆ So this is called **dynamic or late binding.**
- **Upcasting** is the Parent class's reference variable refers to the object of the child class.

Examples of Runtime Polymorphism in Java

- **The Class**

```
class Animal
{
  void eat()
  {
System.out.println("Animals Eat");}  }
class herbivores extends Animal{
  void eat(){
System.out.println("Herbivores Eat Plants");} }
class omnivores extends Animal{
  void eat(){
System.out.println("Omnivores Eat Plants and meat");} }
class carnivores extends Animal{
  void eat(){
System.out.println("Carnivores Eat meat");
}
}
```

Examples of Runtime Polymorphism in Java

- **Main Method**

```
class Main{
  public static void main(String args[]){
    Animal A = new Animal();
    Animal h = new herbivores();
  Animal o = new omnivores();
    Animal c = new carnivores();
    h.eat();
    A.eat();
    c.eat();
    o.eat();
```

Examples of Runtime Polymorphism in Java

Output of the above example is

- **Herbivores Eat Plants**
  Animals Eat
  Carnivores Eat meat
  Omnivores Eat Plants and meat

Java Abstraction

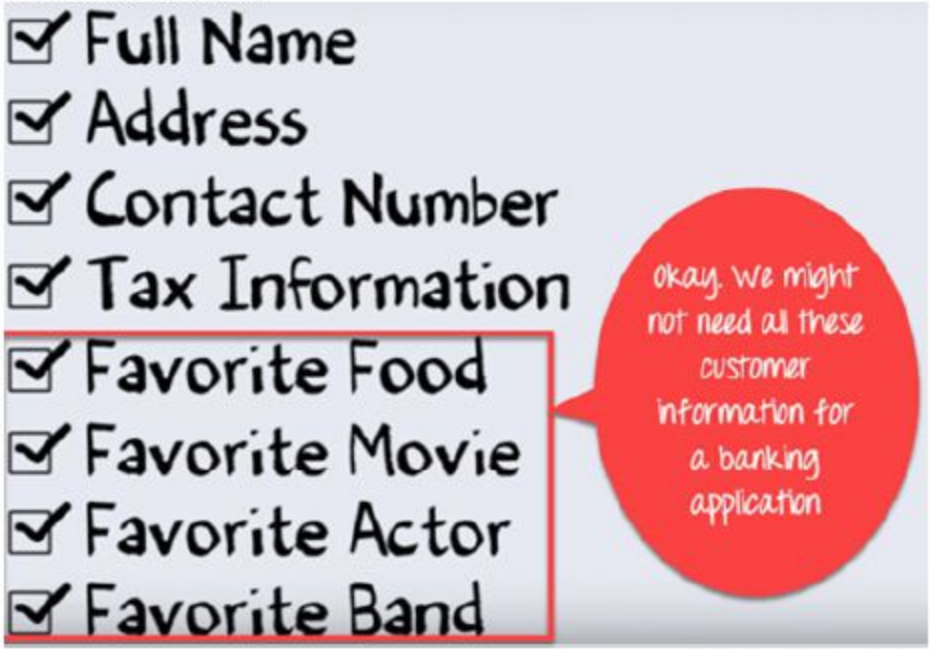
What is Abstraction in OOP?

- **Abstraction** is the concept of **object-oriented programming** that **"shows" only essential attributes and "hides" unnecessary information.**
- **The main purpose** of **abstraction** is **hiding the unnecessary details from the users.**
- Abstraction is a process of **hiding the implementation details** and showing only **functionality to the user**.
- It is one of the most important concepts of OOPs.

Java Abstraction

Abstraction in OOPs with example:

- Suppose you want to create a banking application and you are asked to collect all the information about your customer.
- There are chances that you will come up with following information about the customer



Abstraction in OOPs with example:

- But, not all of the above information is required to create a banking application.
- So, you need to select only the useful information for your banking application from that pool.
  - ⋆ Data like name, address, tax information, etc. make sense for a banking application which is an Abstraction example in OOPs
- Since we have **fetched/removed/selected the customer information from a larger pool**, the process is referred as Abstraction in OOPs.

Java Abstraction

Abstraction in OOPs with example:

- **However,** the **same information once extracted** can be used for **a wide range of applications**.
- **For instance,** you can use the **same data** for **hospital application, job portal application, a Government database**, etc.
  - ⋆ With **little or no modification**.
  - ⋆ Hence, it becomes your Master Data.
  - ⋆ This is an **advantage of Abstraction in OOPs.**

Java Abstraction

Abstraction in OOPs with example:

- **However,** the **same information once extracted** can be used for **a wide range of applications**.
- **For instance,** you can use the **same data** for **hospital application, job portal application, a Government database**, etc.
  - ⋆ With **little or no modification**.
  - ⋆ Hence, it becomes your Master Data.
  - ⋆ This is an **advantage of Abstraction in OOPs.**

Abstract Class

What is Abstract Class?

- **A class** which contains **the abstract keyword** in its declaration is **known as abstract class.**
- It is just like a **normal class but has two differences.**
  1. **We cannot create an object of this class**.
     Only objects of its **non-abstract (or concrete) sub-classes can be created.**
  2. It can have **zero or more abstract methods** .

Abstract Class

When to use Abstract class in Java?

- An abstract class can be used when we need to **share the same method to all non-abstract subclasses with their own specific implementations.**
- Whenever you need to implement a method **differently in different classes** we **use an abstract class.**
- Thus, **abstract class is useful** to make the **program more flexible and understandable.**

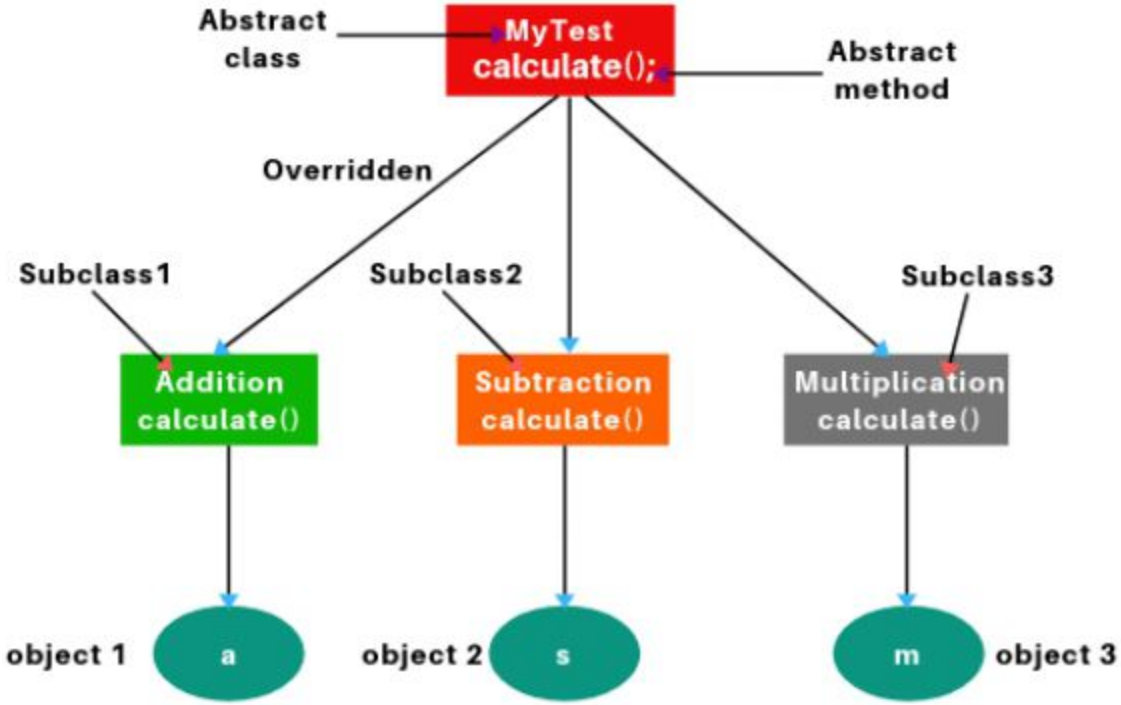
Abstract Class

When to use Abstract class in Java?

- **For example**, if I have a class called "Bank" which has a method namely **"CalcInterest"**, I can keep this method as abstract.
- **All** the banks have **a different way of calculating the interest**.
- So each **class would derive this method and implement it in their own way.**
- **This is why abstract classes are useful.**

Abstract Class

When to use Abstract class in Java?



Abstract Class

What are Abstract Methods?

- A method that is **declared with abstract modifier in an abstract class** and **has no implementation (means no body) is called abstract method in java**.
- It does not contain any body.
- Abstract method has simply **a signature declaration followed by a semicolon.**

Abstract Class

What are Abstract Methods?

- A method that is **declared with abstract modifier in an abstract class** and **has no implementation (means no body) is called abstract method in java**.
- It does not contain any body.
- Abstract method has simply **a signature declaration followed by a semicolon.**
- Since the **abstract method does not contain any body.**
  - ⋆ **Therefore**, it is also known **as incomplete method in java.**
- **Key point:** **A concrete method** is **a method** which has **always the body**.
  - ⋆ It is also called **a complete method in java.**

Abstract Class

What are Abstract Methods?

- A method that is **declared with abstract modifier in an abstract class** and **has no implementation (means no body) is called abstract method in java**.
- It does not contain any body.
- Abstract method has simply **a signature declaration followed by a semicolon.**
- Since the **abstract method does not contain any body.**
  - ⋆ **Therefore**, it is also known **as incomplete method in java.**
- **Key point:** **A concrete method** is **a method** which has **always the body**.
  - ⋆ It is also called **a complete method in java.**

What are Abstract Methods?

- **Syntax:**

```
abstract type MethodName(arguments); // No body
For example:
   abstract void msg(); // No body.
```

- An abstract class can have **both abstract and regular methods:**

```
abstract class Animal
 {
  public abstract void animalSound();
  public void sleep()
  {
    System.out.println("Zzz");
  }
}
```

Abstract Class

When to use Abstract method in Java?

- There are the following uses of abstract method in Java. They are as follows:
  1. An abstract method can be used when the same method has to perform different tasks depending on the object calling it.
  2. A method can be used as abstract when you need to be overridden in its non-abstract subclasses.

Abstract Class

Example: Java Abstract Class

```java
abstract class Language {
  public void display() {
    System.out.println("This is Java Programming");
  }
}

class Main extends Language {

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

Abstract Class

Example: Java Abstract Class

```java
abstract class Language {
  public void display() {
    System.out.println("This is Java Programming");
  }
}

class Main extends Language {

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

Abstract Class

Example: Java Abstract Class

```
abstract class Shape{
abstract void draw();
}
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle");}
}
class Circle1 extends Shape{
void draw(){System.out.println("drawing circle");}
}
class TestAbstraction1{
public static void main(String args[]){
Shape s=new Circle1();
s.draw();
}
}
```

Abstract Class

Example: Java Abstract Class

```
abstract class AbstractDemo{
   public void myMethod(){
      System.out.println("Hello");
   }
   abstract public void anotherMethod();
}
public class Demo extends AbstractDemo{

   public void anotherMethod() {
        System.out.print("Abstract method");
   }
   public static void main(String args[])
   {

      AbstractDemo obj = new AbstractDemo();
      obj.anotherMethod();
   }
}
```

Abstract Class

Example: Java Abstract Class

```
abstract class Bank{
abstract int getRateOfInterest();
}
class SBI extends Bank{
int getRateOfInterest(){return 7;}
}
class PNB extends Bank{
int getRateOfInterest(){return 8;}
}
class Main{
public static void main(String args[]){
Bank b;
b=new PNB();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
b=new SBI();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
 }}
```

Abstract Class

Reading Assignment

1. What is the difference between Abstract and Encapsulation?
2. What is Interface?
3. What is the difference between Abstract and Interface?