Object Oriented Programming (OOP) Concepts

Inheritance, Polymorphism, Abstract Class and Interface

JULY,2023

[expleo]

Object Oriented Programming Concepts

- Inheritance
- Polymorphism
- Abstraction
- Quiz

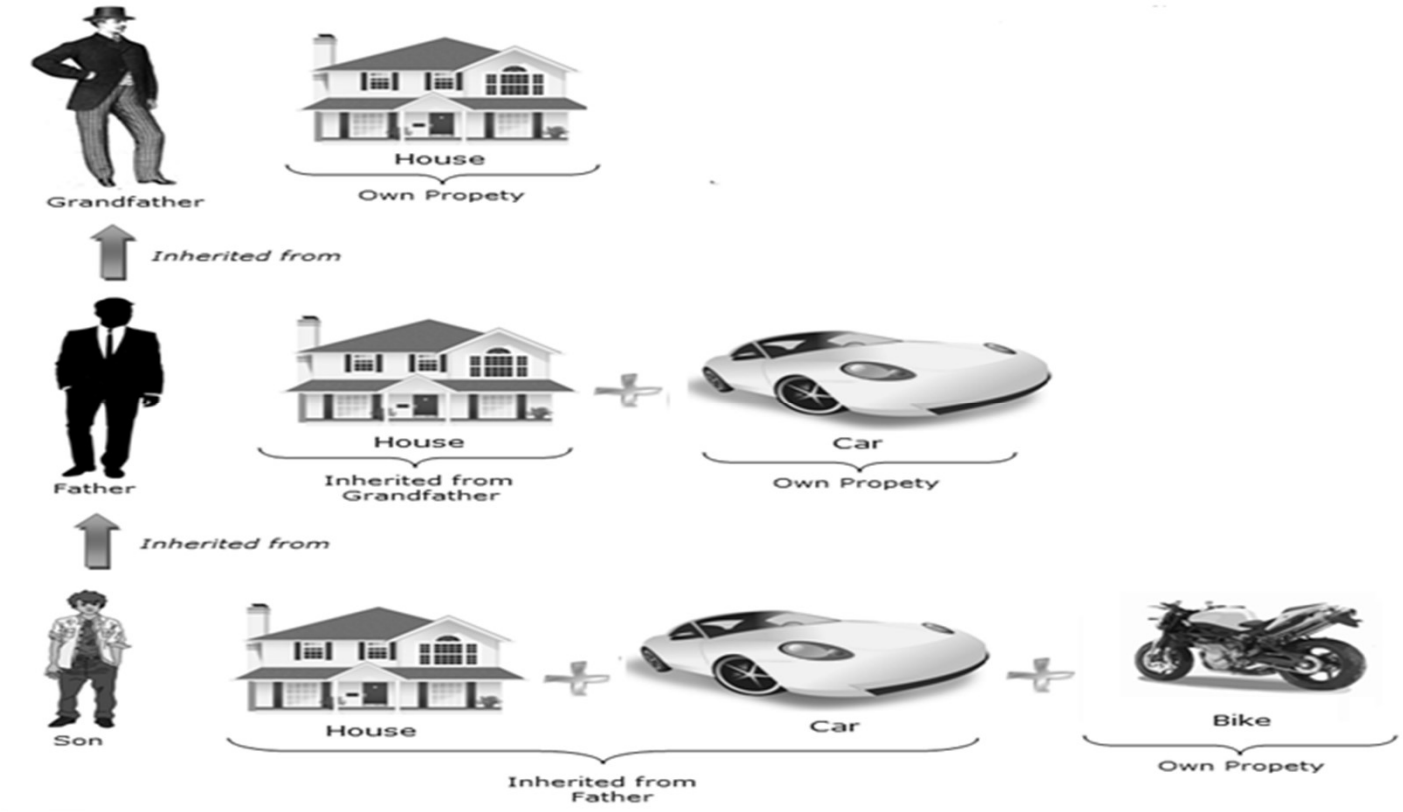
Inheritance

Introduction

- **Inheritance** is one of the important features of Object Oriented Programming which allows you to create **hierarchical classifications.**
- Inheritance represents the **IS-A relationship** which is also known as a ***parent-child*** relationship.
- Inheritance is the process of **acquiring the properties** from one class (**Parent class**) to other classes (**child classes**).
- With the help of inheritance, we can create a more **general class** (**Parent class**) at the top and it may then be inherited by other more **specific classes** (**Child classes**).
- **Child classes** will have the **properties of parent class** and it's **own attributes and behaviors** that are unique to it as well.

Inheritance

Introduction



Important Terminologies

- **Class:** Class is a **template or blueprint** of the objects. It defines the state (variables) and behaviour (methods) common to all objects of a certain kind.

- **SubClass/Child Class:** Subclass is a class which inherits the other class. It is also called a **derived class, extended class, or child class.**

- **Super Class/Parent Class:** Super class is the class from where a subclass inherits the features. It is also called a **base class or a parent class**.

Inheritance

Example

Parent Class

Super Hero Common characteristics

Speed
Strength
Durability
Agility

Spiderman IS-A Super Hero

Child Classes

•extrasensory and vision powers
•freezing breath

•Genius-level intellect
•detective
•strategist

•ability to cling
•spider-sense
•web-shooters

•Supersonic flight
Energy repulsor
•missile projection
Regenerative life support

Need

- **Reusability:** It is a mechanism which facilitates you to **reuse the fields and methods** of the existing class. As the name implies, the child **inherits characteristics** of the parent.

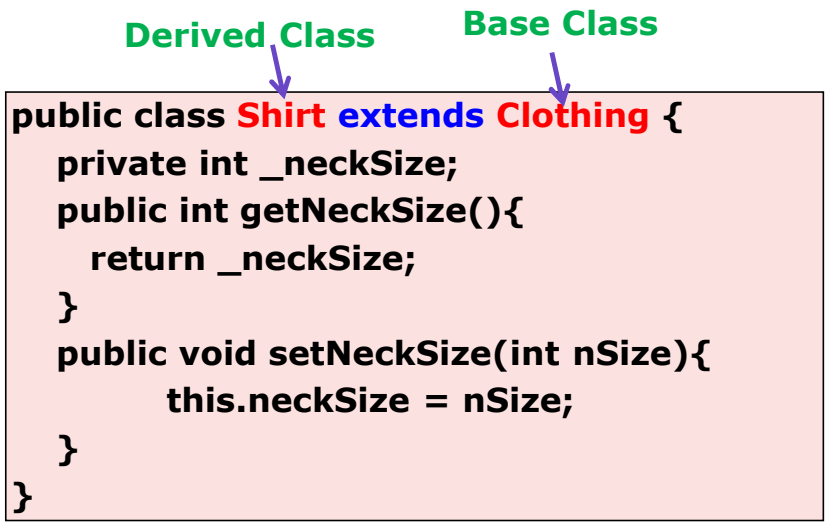
- There is **less code** duplication.

- **Code modification** can be done once for **all subclasses**.

The syntax of Java Inheritance

Syntax

```
class SubClass-name extends SuperClass-name {

   //methods and fields  of subclass


}
```



• The **extends** keyword denotes that a **subclass is derived from an existing super class**. It indicates that you are extending the functionality of an existing class.

Types of Inheritance

- **Below are the types of inheritance.**
  - –Single Inheritance
  - –Multilevel Inheritance
  - –Multiple Inheritance
  - –Hierarchical Inheritance
  - –Hybrid Inheritance

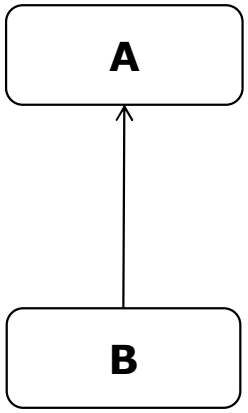
Note:

- Java supports Single, Multilevel and Hierarchical inheritance directly but Multiple and Hybrid inheritance can be achieved through Interfaces.

Single Inheritance

- Single inheritance is the concept of deriving the **properties and behaviours** from single base (parent) class

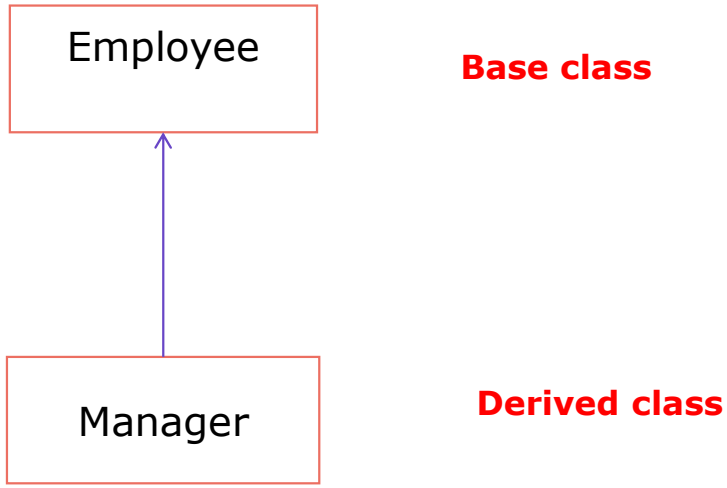
```
class A
{
 ....
 ....
 ....
}

class  B extends A
{
 ....
 ....
 ....
}
```



Base class

Derived class

Single Inheritance - Example



Single Inheritance – Example #1

```
/** This example demonstrates single inheritance concept **/

class Employee {                //parent class
    String empName;
    int empId;
    void setData(String name, int id){   // base class method
                    empName=name;
                    empId=id;
    }
    void displayData(){                        // base class method
                    System.out.println("Employee Name:"+empName);
                    System.out.println("ID:"+empId);
    }
}
```

Single Inheritance – Example #1

```
class Manager extends Employee {          //child class
    String empDept;
    void setDept(String dept) {           //sub class method
                    empDept = dept;
    }
    void displayDept(){                   //sub class method
                    System.out.println("Department:"+dept);
    }
}
```

Single Inheritance – Example #1

```
public class SingleInheritanceDemo{
    public static void main(String args[]){
                    Manager m=new Manager();          //child class object
                    m.setData("Arun", 123);  //access base class members
                    m.setDept("Marketing");
                    m. displayData();  //access base class members
                    m. displayDept();
    }
}
```

Output

Employee Name: Arun

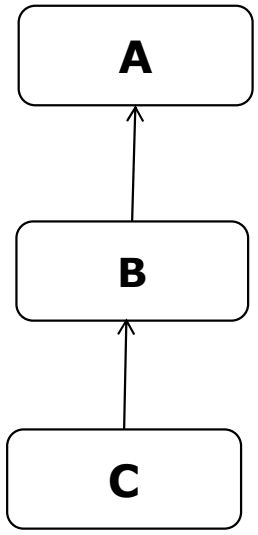
ID:123

Department: Marketing

Multilevel Inheritance

- In the case of multi-level inheritance, a subclass that is inheriting one **parent class will also act as the base class** for another class.

```
class A
{
  ....
 }
class  B extends A
{
  ....
  ....
}
class  C extends B
{
  ....
  ....
 }
```

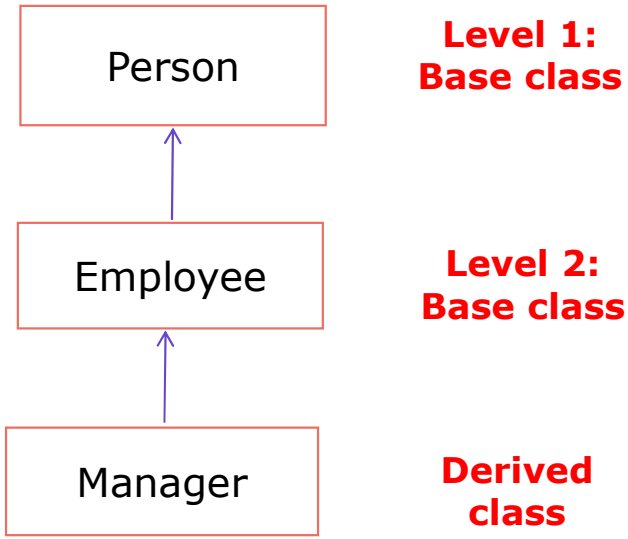


Level 1: Base class

Level 2: Base class

Derived class

Multilevel Inheritance - Example



Multilevel Inheritance- Example #1

```
/**   * This example demonstrates Multilevel inheritance concept **/

class Person {                    //Level 1:Base class
    String name;
    int age;
    void setPersonData(String name, int age){
                    this.name=name;
                    this.age=age;
    }
    void displayPersonData() {
                    System.out.println("Name:"+name);
                    System.out.println("Age:"+age);
    }
}
```

Multilevel Inheritance-Example #1

```
class Employee  extends Person{          //Level 2:Base class
    int empId;
    void setEmpData(String id){
                    empId=id;
    }
    void displayEmpData(){
                    System.out.println("ID:"+empID);
    }
}
```

Multilevel Inheritance- Example #1

```
class Manager extends Employee {          //child class
    String dept;
    float sal;
    void setManagerData(String depart, float salary){
                    dept = depart;
                    sal=salary;
    }
    void displayManagerData(){
                    System.out.println("Department:"+dept);
                    System.out.println("Salary :"+ sal);
    }
}
```

Multilevel Inheritance- Example #1

```
public class MlevelInherDemo{
    public static void main(String args[]) {
        Manager m=new Manager();          //child class object
                m.setPersonData("Arun", 34);
                m.setEmpData("M123");
                m.setManagerData("Marketing",60000);
                m.displayPersonData();
                m.displayEmpData();
                m.displayManagerData();
    }
}
```

Output

Name: Arun

Age:34

ID:M123

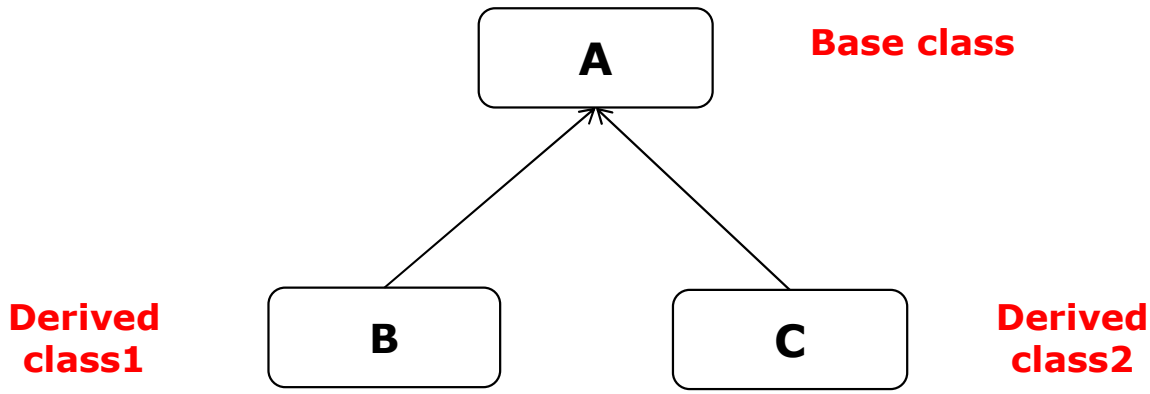
Department: Marketing

Salary :60000.0

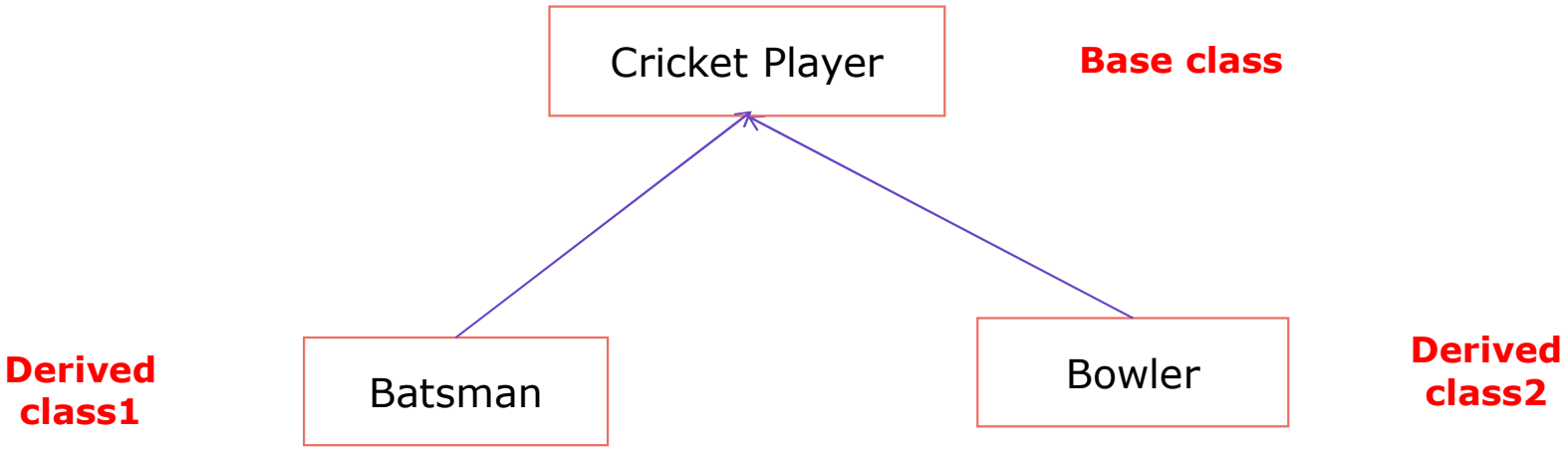
Hierarchical Inheritance

- In Hierarchical Inheritance concept, there is **one base class for multiple subclasses**.

```
class A
{
  ....
  ....
   }
class  B extends A
{
  ....
  ....
}
class  C extends A
{
  ....
  ....
 }
```



Hierarchical Inheritance - Example



Hierarchical Inheritance-Example

```
/**
 * This example demonstrates Hierarchical inheritance concept **/
class CricketPlayer {                    //Base class
   String playerName;
   String teamName;
    void setPlayerData(String playerName, String teamName){
      this.playerName=playerName;
      this.teamName=teamName;
    }
    void displayPlayerData() {
     System.out.println(" Player Name:"+playerName);
     System.out.println("Team Name:"+teamName);
    }
}
```

Hierarchical Inheritance-Example

```
class Batsman extends CricketPlayer {                    //Derived class 1
   int higestScore;
   float batAvg;

    void setBatsmanData(int higestScore, float batAvg){
      this.higestScore=higestScore;
      this.batAvg=batAvg;
    }

    void displayBatsmanData(){
     System.out.println(" Highest Score:"+higestScore);
     System.out.println("Batting Average:"+batAvg);
    }
}
```

Hierarchical Inheritance-Example

```
class Bowler extends CricketPlayer {              //Derived class  2
   int wickets;
   float bowlAvg;
    void setBowlerData(int wickets, float bowlAvg){
      this.wickets=wickets;
      this.bowlAvg=bowlAvg;
    }
    void displayBowlerData() {
     System.out.println(" No. of Wickets:"+wickets);
     System.out.println("Bowling Average:"+bowlAvg);
    }
}
```

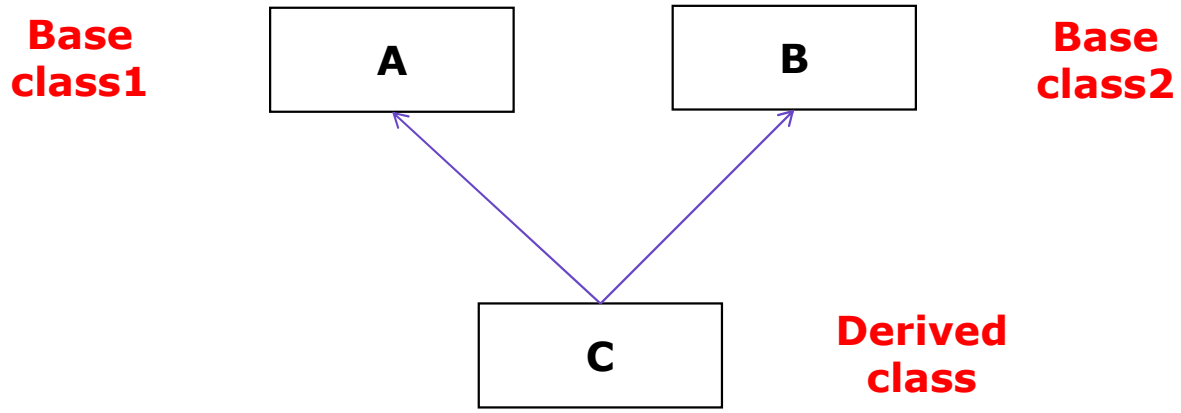
Hierarchical Inheritance- Example

```
public class HInherDemo{
  public static void main(String args[]){
     Batsman B1=new Batsman();       //Base Class Object
     Bowler B2=new Bowler();
     B1.setPlayerData("Sachin", "India");
     B1.setBatsmanData(200, 84.5);
     B2.setPlayerData("Bumra", "India");
     B2.setBowlerData(140, 6.75);
     B1.displayPlayerData();
     B1.displayBatsmanData();
     B2.displayPlayerData();
     B2.displayBowlerData();
   }
}
```

Output:
Player Name:Sachin
Team Name:India
Highest Score:200
Batting Average:84.5
Player Name:Bumra
Team Name:India
No. of Wickets:140
Bowling Average:6.75

Multiple Inheritance

- Multiple Inheritance concept, there is **more than one base classes** for a subclass.



Note:

- Due to **ambiguity issue** Java doesn't support multiple inheritance through class.
- Both multiple and hybrid inheritance supported through **interface concept**.

super Keyword

- The **super** keyword in Java is a **reference variable** which is used to refer **immediate parent class object.**
- Whenever you create the instance of subclass, an **instance of parent class** is created **implicitly** which is referred by super reference variable.
- Super keyword can be used at **variable, method and constructor** level.
- The super keyword is similar to 'this' keyword.
  - **Difference**: super keyword is used to refer the **members of super class.** But 'this' keyword used to refer current **class's instance and static variables**.

super Keyword: Example

```
/**
 * This example demonstrates the usage of super in variable level. **/
class ProjectLeader {                  //parent class
    String proleadName="Ram Kumar";
    int empId = 1000;
}
class Programmer extends ProjectLeader  {          //child class
    String progName;
    int empId;
    void setData(String name, int id){   // derived class method
      progName=name;
      empId=id;
    }
```

super Keyword: Example

```
void displayData(){   // derived class method
          System.out.println(" Programmer Name: "+ progName);
           System.out.println(" Programmer Id: "+ empId);
           System.out.println(" Project Leader Name: "+ proleadName);
           System.out.println(" Project Leader Id: "+ super.empId);  // access base class variable using super
    }
}
public class SuperVariableDemo{
  public static void main(String args[]){
    Programmer obj=new Programmer ();          //child class object
     obj.setData("Arun", 111);
     obj.displayData()
   }
}
```

Output:
Programmer Name: Arun
Programmer Id: 111
Project Leader Name: Ram Kumar
Project Leader Id: 1000

super Keyword: Example

```
/**
 * This example demonstrates the usage of super in method level. **/
class ProjectLeader {                //parent class
    String proleadName="Ram Kumar";
    int empId = 1000;
    void displayData(){   // base class method
            System.out.println("Project Leader Name: "+ proleadName);
            System.out.println("Project Leader Id: "+ empId);
   }
}
```

super Keyword: Example

```
class Programmer extends ProjectLeader  {          //child class
    String progName;
    int empId;
    void setData(String name, int id){   // derived class method
      progName=name;
      empId=id;
    }
    void displayData(){   // derived class method
          System.out.println("Programmer Name: "+ progName);
          System.out.println("Programmer Id: "+ empId);
          super.displayData();  //call base class method using super
    }
}
```

super Keyword: Example

```
public class SuperMethodDemo{
  public static void main(String args[]){
    Programmer obj=new Programmer ();          //child class object
     obj.setData("Arun", 111);
      obj.displayData();
   }
}
```

Output:

```
Programmer Name: Arun
Programmer Id: 111
Project Leader Name: Ram Kumar
Project Leader Id: 1000
```

Inheritance and Constructors

- **Can Constructors be inherited? -** No
- **Reasons:** Constructors are special and have same name as class name. So if **constructors** were inherited in child class then child class would contain a **parent class constructor** which is **against the constraint** that constructor should have **same name as class name**.
- Constructors are invoked in the **order of their derivation i.e. first base class then derived class constructor.**
- Constructor of base class with **no argument** gets **automatically called** in derived class constructor.
- But in case of **parameterized base class constructor** call using **super keyword**. In this case, **Base class constructor** call must be the **first line in** derived class constructor.

Inheritance and Constructors: Example

```
/**
* This example demonstrates order of invocation of constructors **/
class Base {
   Base( )
   {
      System.out.println("Inside Base's Constructor");
   }
}
class Derived1 extends Base {
   Derived1( )
   {
     System.out.println("Inside Derived1's Constructor");
   }
}
```

Inheritance and Constructors: Example

```
class Derived2 extends Derived1 {
  Derived2()  {
    System.out.println("Inside Derived2's Constructor");
  }
}

class OrderOfConstructorCallDemo{
  public static void main(String args[]) {
    Derived2 obj = new Derived2();
  }
}
```

Output:

Inside Base's Constructor
Inside Derived1's Constructor
Inside Derived2's Constructor

Polymorphism

Introduction

• **Polymorphism** is one of the important features of Object Oriented Programming.

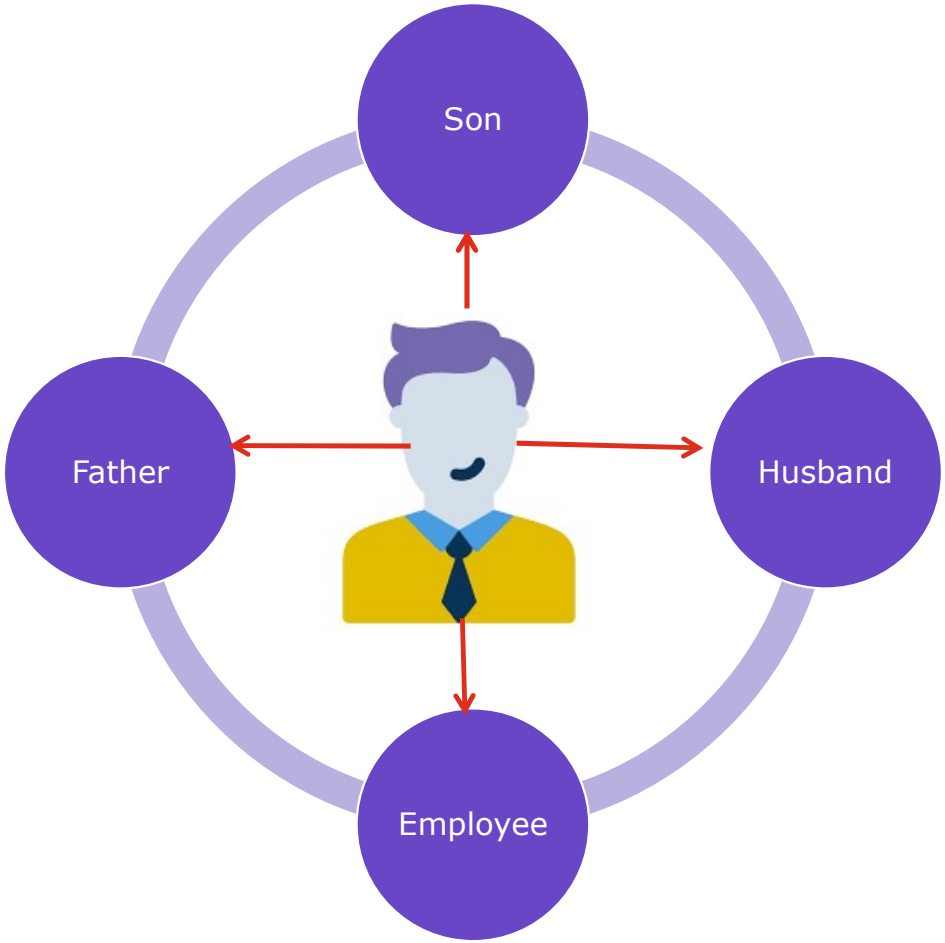
• **Polymorphism** can be defined as the ability of an object to **take many forms**. Simply , polymorphism allows performing the **same action in different ways**.

Example:

• A person at the same time can have **different characteristics**. Like a man at the same time is a **father, a husband, an employee**. So the same **person posses different behaviour in different situations.**

Polymorphism

Introduction



Introduction

- There are **two types of polymorphism** in java:
    1. **Static** Polymorphism also known as **compile time** polymorphism

       Example : Method Overloading

    2. **Dynamic** Polymorphism also known as **runtime** polymorphism

       Example: Method Overriding

Compile time polymorphism: Method Overloading

- When a **type of the object** is determined at the **compile time** (by the compiler), it is known as **compile time polymorphism** or ***static binding*** *or* **Early Binding.** That means Java would be able to **understand which function to be called**, at the **compile time itself.**
- **Example:** Method Overloading
- **Method overloading** : If the class contains two or more methods having the **same name** but **different arguments**, then it is called as ***method overloading*.**
- The compiler will be able to make the call to a **correct method**, depending on the **actual number** of **arguments**, its **data type** and **the sequence they are passed in**.

Example: int add(int, int)

double add(double, double);

float add(float, int, float);

Compile time polymorphism: Method Overloading

- **In Java, method overloading mainly used in the inbuilt classes.**

Method	Use
void println()	Terminates the current line by writing the line separator string
void println(boolean x)	Prints a boolean value and then terminates the line
void println(char x)	Prints a character and then terminates the line
void println(char[] x)	Prints an array of characters and then terminates the line

Note:

Because of ambiguity, method overloading is not possible by **changing the return type** of method only.

For Example: int add(int,int);

double add(int,int);

Method Overloading: Example

```
/**
 * This program demonstrates method overloading*/
class Adder {
      static int add(int a, int b) {
            return a + b;
      }
      static double add(double a, double b) {
            return a + b;
      }
}
class MainClass {
    public static void main(String[] args) {
        System.out.println(Adder.add(5, 6));
        System.out.println(Adder.add(12.8, 9.2));
   }
}
```

Output:
11
22.0

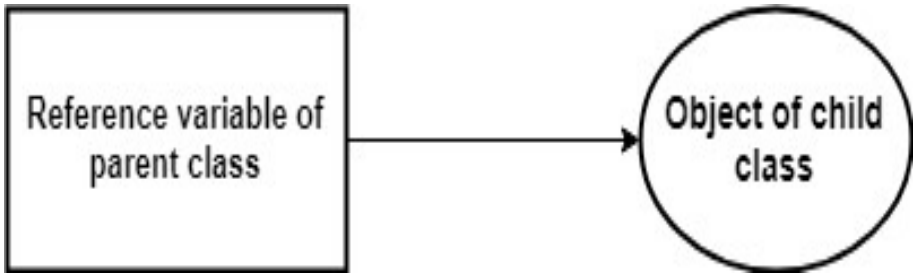
Runtime polymorphism: Method Overriding

- The process of binding the code associated with the **function call during runtime** rather than compile-time is known as **Runtime Polymorphism** or **Dynamic Binding** or **Late Binding.**
- **Example: Method Overriding**
- **Method Overriding:** If subclass (child class) has **same name**, **same parameters** and **same return type** as a method in **super-class** then the method in the subclass is said to ***override*** the method in the super-class.
- In simple words, the function has **same name and same signature** is known as **method overriding**.
- Overridden method in Java application is actually **called and executed** at **run time** only. In method overriding the derived class can give its **own specific implementation** to an inherited method.

Runtime polymorphism: Method Overriding

- In this process, the overridden method would be called through the **reference variable of a super class**. The determination of the method, which is to be called, is based on the object, being referred to by the reference variable (**upcasting**). That's why it is also called as **Dynamic Method Dispatch**.

- **Upcasting** means assigning child class reference to parent class reference**.**



Note:

- They must have the same argument list.
- They must have the same return type.
- Constructors cannot be overridden.

Polymorphism

Upcasting

```
/*** This abstract code illustrates the upcasrting. */

class A {

}


class B extends A {

}


 class Demo {
       public static void main(String[] args) {
            A a = new B();      //upcasting
       }
 }
```

Polymorphism

Method Overriding-Example

```
/**
 * This example demonstrates method overriding
**/
 class Vehicle {
    void run() {
     System.out.println("Vehicle is running");
    }
}
class Truck extends Vehicle {
    void run()    {
     System.out.println("Truck is running");
    }
}
```

Method Overriding-Example

```
class OverrideDemo
{
   public static void main(String args[])
  {

     Vehicle obj = new Vehicle();
     obj.run();                    //Vehicle class run () method invoked

     Vehicle obj = new Truck();
     obj.run();                    //Truck class run () method invoked

  }
}
```

Output:

Vehicle is running

Truck is running

final Keyword

- **final keyword** can be used in context of **behavioural restriction** on
  - –Variables
  - –Methods
  - –Classes
- When a **variable** is made as final then it **behaves as constant**
- If a **method** is set as final then it **cannot be overridden** by the sub classes. It restricts overriding.
- Similarly when a **class** is set as final then it **prevents from being inherited**.

Polymorphism

final variable

```
/**
 * This program demonstrates the use of final keyword */
public class Sample  {
  final double pi;
  public Sample()   {
   pi = 3.14;
 }
 public Sample(double pi)  {
   this.pi = pi;
 }
```

Polymorphism

final variable

```
    public static void main(String[] args) {
            Sample obj = new Sample(22.0/7.0);
            System.out.println(obj.pi);
    }
}
```

```
Output:
3.142857142857143
```

Polymorphism

final Method

```
/**
 * This program demonstrates the use of final keyword
*/
class Base  {
   public final void display(String s) {
     System.out.println(s);
   }
}
class Sample extends Base {
 public void display(String s)   {
   System.out.println(s);
  }
```

Polymorphism

final Method

```
   public static void main(String args[]) {
         Sample obj = new Sample();
         obj.display("TRY ME");
 }
}
```

Output:
Compile Time Error : Cannot override the final method from Base

Polymorphism

final Class

```
/**
 * This program demonstrates the use of final keyword */
final class Base  {
   public final void display(String s)    {
     System.out.println(s);
   }
}
class Sample extends Base {
 public void display(String s) {
   System.out.println(s);
  }
```

Polymorphism

final Class

```
    public static void main(String args[]) {
            Sample obj = new Sample();
            obj.display("TRY ME");
    }
}
```

Output:
Compile Time Error : The type Sample cannot subclass the final class Base

Abstraction

Introduction

- **Abstraction** is the one of the important feature of Object Oriented Programming.
- It is a process of **hiding the implementation** details and showing only the **functionality** to the user.
- In other words, it **shows only essential things** to the user and **hides the other internal details**. **For example**, sending SMS, it shows only the area where you can type the text and send the message. The internal processing about the message delivery will be hidden.
- There are **two ways** to achieve abstraction in Java,

  1. Abstract Class
  2. Interface

Abstraction

Introduction

- An abstraction includes the **essential details** relative to the **perspective of the user.**

Abstract Class: Need

- An **abstract class** is a restricted class, that cannot be used to create objects. That means we are forcing the programmer to inherit the abstract class and use it, but not directly.
- A class which is declared with **abstract keyword**, becomes an **abstract class**. **Abstract class** can have **abstract methods** and as well as **concrete methods**.

```
Public abstract class Shape {
   void draw() {
      System.out.println("drawing...");      //Concrete Method
   }
   public abstract void area();              //Abstract Methods
   public abstract void perimeter();
}
```

Note: Abstract method is a method which will not have the body, whereas concrete methods must have the body.

Abstract Class

Points to remember:

- For abstract class we cannot create object. (**Why?)**
- Abstract class can contain both abstract and concrete methods.
- It can have constructors and static methods also.
- It can have final methods, which will force the programmer not to **change the body of the methods** in the subclass.
- It contain abstract methods that **must be implemented** later by any **non abstract subclasses.**

Abstraction

Abstract Class

```
/**
 * This program demonstrates runtime polymorphism using abstract class
*/
abstract class Shape {
   void draw() {
      System.out.println("drawing...");
   }
   abstract void area();
   abstract void perimeter();
}
class Rectangle extends Shape {
   private int length, breadth;
   Rectangle(int length, int breadth){
         this.length = length;
         this.breadth = breadth;
   }
```

Abstract Class

```
@Override
   void area() {
      System.out.println("Area of Rectangle: " + (length * breadth));
   }
  @Override
   void perimeter() {
      System.out.println("Perimeter of Rectangle: " + (2 * (length + breadth)));
   }
}
class Square extends Shape {
   private int side;
   Square(int side){
         this.side = side;
   }
```

Abstract Class

```
@Override
    void area() {
        System.out.println("Area of Square: " + (side * side));
    }
  @Override
    void perimeter() {
        System.out.println("Perimeter of Square: " + (4 * side));
    }
}
class Circle extends Shape {
    private double radius;
    final static double PI = 3.14;
    Circle(double radius){
            this.radius = radius;
    }
```

Abstraction

Abstract Class

```
@Override
   void area() {
      System.out.println("Area of Circle: " + (PI * radius * radius));
   }

   @Override
   void perimeter() {
      System.out.println("Perimeter of Circle: " + (2 * PI * radius));
   }
}
```

Abstraction

Abstract Class

```
class MainTest {
   public static void main(String args[]) {
      Shape s;
      s = new Rectangle(3,5);
      s.area();
      s.perimeter();

      s = new Square(5);
      s.area();
      s.perimeter();

      s = new Circle(4.5);
      s.area();
      s.perimeter();    }
}
```

Output:
Area of Rectangle: 15
Perimeter of Rectangle: 16
Area of Square: 25
Perimeter of Square: 20
Area of Circle: 63.585
Perimeter of Circle: 28.26

Abstraction

Interface

- Another way to achieve Abstraction in java is **interface**. Unlike abstract class an **interface** is used for **full abstraction.**
- **Interface** is a reference type in java. It is very similar to class but it is **not a class**. In interface all the **variables** declared in **final static variables by default** and **methods** are **public abstract by default**.
- From **Java 8 onwards** interface also support **default methods** and **static methods**, which may have implementation details.
- **Default methods** were introduced **to provide backward compatibility for old interfaces** so that they can have **new methods without effecting existing code**.

Syntax : interface interfaceName { ... }

Interface: Need

- It is used to achieve **full abstraction (100%).**

- **Multiple Inheritance** can be achieved with interfaces, because the **class can implement multiple interfaces.**

Note:

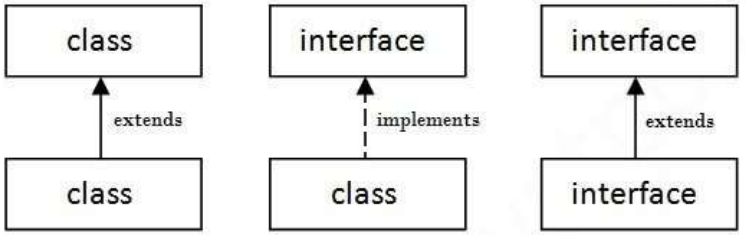
We cannot create object for interface.

The class which is implementing the interface, must define all the methods of interface. If it fails to define any method of the interface, the class becomes abstract class. if one of the methods is a default method, it needs not be redefined.

Abstraction

Interface

Points to Remember:



- Interface can extend one or more other interface.
- Interface cannot implement a class.
- Interface can be nested inside another interface.
- An interface cannot contain a constructor. **(Why?)**
- When a class implements an interface, it is like **signing an agreement**. The agreement indicates that the class will **implement the methods defined by the interface**.

Interface: Example

```
/**
 * This program demonstrates interface concepts
*/
interface Vehicle {
   void changeGear(int);
   void speedUp(int);
   void applyBrakes(int);
}
 class Bicycle implements Vehicle{
   int speed;
   int gear;
   //Abstract Method Implementation
   public void changeGear(int newGear){
      gear = newGear;
   }
```

Interface: Example

```
class Bike implements Vehicle {
   int speed;
   int gear;
   //Abstract Method Implementation
   public void changeGear(int newGear){
      gear = newGear;
   }
   //Abstract Method Implementation
   public void speedUp(int increment){
      speed = speed + increment;
   }
    //Abstract Method Implementation
   public void applyBrakes(int decrement){
      speed = speed - decrement;
   }
```

Interface: Example

```
//Abstract Method Implementation
   public void speedUp(int increment){
      speed = speed + increment;
   }
 //Abstract Method Implementation
   public void applyBrakes(int decrement){
      speed = speed - decrement;
   }

   public void printStates() {
       System.out.println("Speed: " + speed + " Gear: " + gear);
   }
}
```

Interface: Example

```
public void printStates() {
        System.out.println("Speed: " + speed + " Gear: " + gear);
    }
}

class MainClass {
        public static void main (String[] args) {
        Bicycle bicycle = new Bicycle();  //Bicycle Class Object Creation
        bicycle.changeGear(3);
        bicycle.speedUp(2);
        bicycle.applyBrakes(1);
        System.out.println("Bicycle present state :");
        bicycle.printStates();
```

Interface: Example

```
        Bike bike = new Bike(); //Bike Class Object Creation
        bike.changeGear(2);
        bike.speedUp(3);
        bike.applyBrakes(3);

        System.out.println("Bike present state :");
        bike.printStates();
}
}
```

Output:
Bicycle present state :
Speed: 1 Gear: 3
Bike present state :
Speed: 0 Gear: 2

Quiz

1. We have to use the concept of inheritance when there is a "IS-A" relationship between two classes. (Yes/No)

a) Yes **b) No**

a) Yes

Quiz

2. What is the keyword used for inheriting a class in Java?

a) extends

b) implements

c) instanceOf

d) None

a) extends

Quiz

3. To prevent from being inherited, the _____ keyword is used before the class

a) static

b) final

c) this

d) private

b) final

Quiz

4. Can constructors of a base class be inherited to its sub classes?

a) Yes

b) No

b) No

Quiz

5. What is the term which is used to denote the concept of re-using and re-defining the method of a parent class in a subclass?

a) overloading

b) overriding

c) extending

d) None

b) overriding

Quiz

6. Which one is the correct way of inheriting class A by class B

a) class B extends A

b) class A extends B

c) class A inherits B

d) Class B inherits A

a) class B extends A

Quiz

7. What is the output after the following code has been executed?

```
class Base
   {
      int i;
      void display()
      {
         System.out.println(i);
      }
   }
   class Derived extends Base
   {
      int j;
      void display()
      {
         System.out.println(j);
      }
   }
 public class inheritance_demo
 {
    public static void main(String args[])
    {
       Derived obj = new Derived();
       obj.i=5;
       obj.j=10;
       obj.display();
    }
}
```

Quiz

8. What is the output after the following code has been executed?

```
class Base
   {
      int i;
      void display()
      {
         System.out.println(i);
      }
   }
   class Derived extends Base
   {
      int j;
      void display()
      {
         super.display();
         System.out.println(j);

      }
   }
   public class inheritance_demo
   {
      public static void main(String args[])
      {
         Derived obj = new Derived();
         obj.i=5;
         obj.j=10;
         obj.display();
      }
   }
```

Quiz

9. What is the output after the following code has been executed?

```
class Base
  {
      public Base()
     {
            System.out.print("Base");
      }
    }
  public class Derived extends Base
  {
      public Derived()
      {
            this(“Java");
            System.out.print("Derived");
      }
      public Derived(String s)
      {
            System.out.print(s);
      }
      public static void main(String[] args)
      {
            new Derived();
      }
}
```

Quiz

11. What does the name Polymorphism translate to?

a) Many forms

b) Many changes

c) Two forms

d) None of the above

a) Many forms

Quiz

12. What are the two types of Polymorphism?

a) compile time and runtime

b) Constructor and method

c) derive and base

d) encapsulation and Inheritance

a) compile time and runtime

Quiz

14. The "is a" relationship between super class and sub class is commonly referred as:

a) Inheritance

b) Overriding

c) Constructor

d) None

a) Inheritance

Quiz

15. When does method overloading is determined?

a) At runtime

b) At Compile time

c) At coding time

d) None

b) At Compile time

Quiz

16. Which inheritance in java programming is not supported?

a) Single Inheritance

b) Multilevel Inheritance

c) Multiple inheritance using classes

d) Multiple Inheritance using interfaces

c) Multiple inheritance using classes