



Dr. Vishwanath Karad

**MIT WORLD PEACE
UNIVERSITY** | PUNE

TECHNOLOGY, RESEARCH, SOCIAL INNOVATION & PARTNERSHIPS

CET1042B: Object Oriented Programming with C++ and Java

SCHOOL OF COMPUTER ENGINEERING AND TECHNOLOGY

**S. Y. B. TECH. COMPUTER SCIENCE AND ENGINEERING
(CYBERSECURITY AND FORENSICS)**

CET1042B: Object Oriented Programming with C++ and Java

Teaching Scheme
Theory: 2 Hrs. / Week

Credits: 02 + 02 = 04
Practical: 4 Hrs./Week

Course Objectives

- 1) **Knowledge:** (i) Learn object oriented paradigm and its fundamentals.
- 2) **Skills:** (i) Understand Inheritance, Polymorphism and dynamic binding using OOP.
(ii) Study the concepts of Exception Handling and file handling using C++ and Java.
- 3) **Attitude:** (i) Learn to apply advanced concepts to solve real world problems.

Course Outcomes

- 1) Apply the basic concepts of Object Oriented Programming to design an application.
- 2) Make use of Inheritance and Polymorphism to develop real world applications.
- 3) Apply the concepts of exceptions and file handling to store and retrieve the data.
- 4) Develop efficient application solutions using advanced concepts.

Assignment 2

Implementation of inheritance using C++ and JAVA (use concept of interfaces).

Problem Statement_Assignment 2

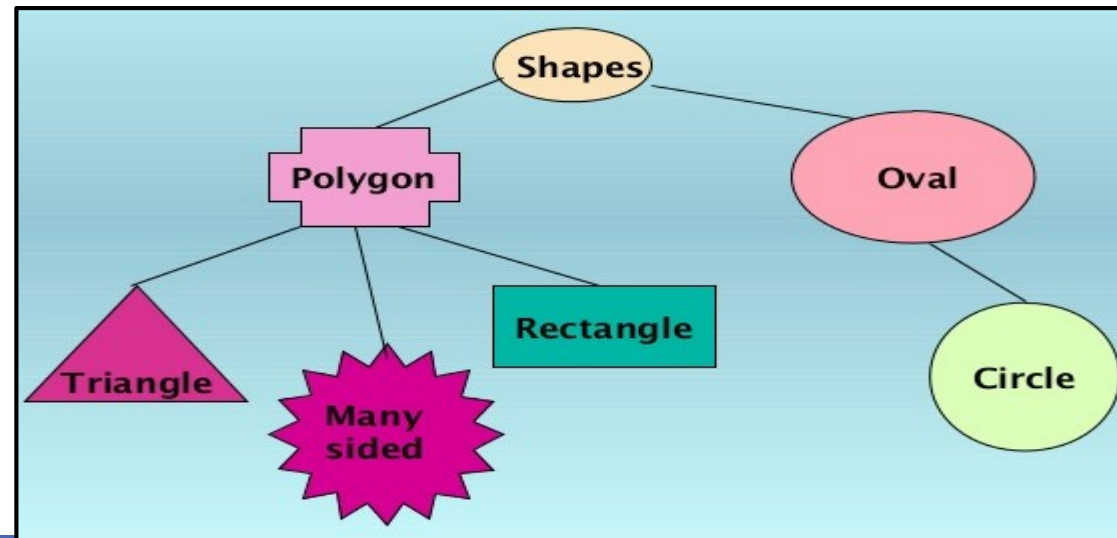
Design and develop inheritance for a given case study, identify objects and relationships and implement inheritance wherever applicable.

- **Employee** class has Emp_name, Emp_id, Address, Mail_id, and Mobile_no as data members.
- Inherit the following classes from employee class.
 - ✓ Programmer
 - ✓ Team Lead
 - ✓ Assistant Project Manager and
 - ✓ Project Manager
- Add Basic Pay as the member of all the inherited classes with 97% of Basic Pay as DA, 10 % of Basic Pay as HRA, 12% of Basic Pay as PF, 0.1% of Basic Pay for staff club fund.
- Generate pay slips for the employees with their gross and net salary.

Getting Introduced with Inheritance

Introduction to Inheritance

- It is the mechanism by which one class acquires the properties of another class
- Provides a way to create a new class from an existing class
- The new class is a specialized version of the existing class
- Inheritance establishes an "is a" relationship / a parent-child relationship between classes
- Allows sharing of the behavior of the parent class into its child classes
- Child class can add new behavior or override existing behavior from parent
- It allows a hierarchy of classes to be built, moving from the most general to the most specific



Example: Shape Taxonomy

Benefits of using inheritance

- Reusability:
reuse the methods and data of the existing class
- Extendibility:
extend the existing class by adding new data and new methods
- Modifiability:
modify the existing class by overloading its methods with newer implementations
- Saves memory space and time
- Increases reliability of the code
- Saves the developing and testing efforts

Terminologies used inheritance

❑ **superclass, base class, parent class:**

- Describe the parent in the relationship, which shares its functionality
- Defines all qualities common to any derived classes.

❑ **subclass, derived class, child class:**

- Describe the child in the relationship, which accepts functionality from its parent
- Inherits those general properties and adds new properties that are specific to that class.

❑ **extend, inherit, derive:**

become a subclass of another class

Class Derivation in C++

Any class can serve as a base class.....Thus a derived class can also be a base class

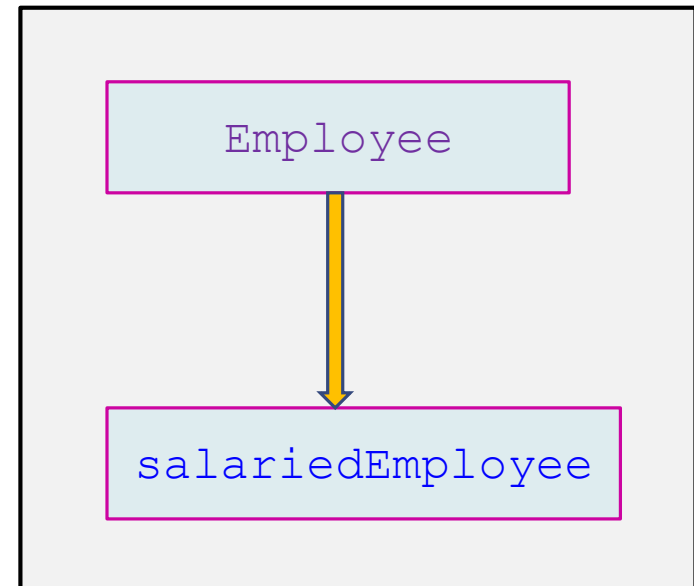
Syntax

```
class DerivedClassName:: specification BaseClassName
```

- DerivedClassName - the class being derived
- specification - specifies access to the base class members
 - public / protected / private - private by default

Example:

```
class Employee // base class
{
    . . .
};
class salariedEmployee : public Employee // derived class
{
    . . .
};
```



Member Access Control

Type 1: inherit as private

Base	Derived
private members	inaccessible
protected members	private members
public members	private members

Type 2: inherit as protected

Base	Derived
private members	inaccessible
protected members	protected members
public members	protected members

Type 3: inherit as public

Base	Derived
private members	inaccessible
protected members	protected members
public members	public members

Inheritance and Access Specifier

Access	public	Protected	Private
Members of the same class	Yes	Yes	Yes
Members of derived classes	Yes	Yes	No
Non-members	Yes	No	No

Types of Inheritance

1. Single Inheritance:

It is the inheritance hierarchy wherein one derived class inherits from one base class.

2. Multiple Inheritance:

It is the inheritance hierarchy wherein one derived class inherits from multiple base class(es)

3. Hierarchical Inheritance:

It is the inheritance hierarchy wherein multiple subclasses inherits from one base class.

4. Multilevel Inheritance:

It is the inheritance hierarchy wherein subclass acts as a base class for other classes.

5. Hybrid Inheritance:

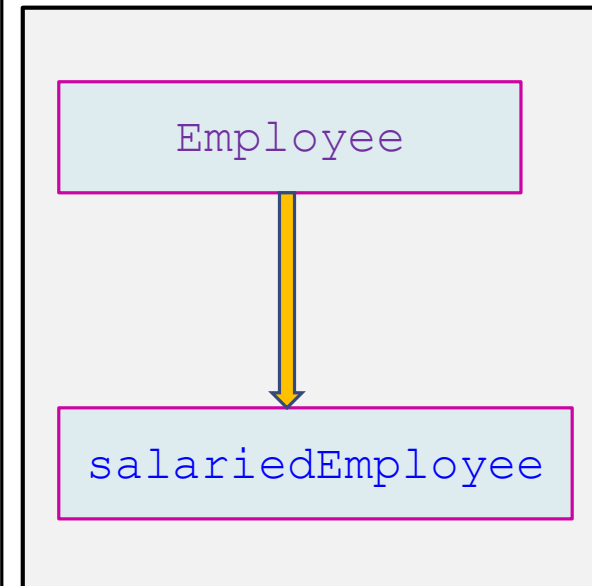
The inheritance hierarchy that reflects any legal combination of other four types of inheritance.

Single Inheritance

- There is only one base class and has only one derived class.

```
class Employee // Employee superclass
{
    ....
};

class SalariedEmployee : public Employee // SalariedEmployee subclass inherits class Employee
{
    private:
        double weeklySalary;
    public:
        ....
};
```



Multiple Inheritance

One derived class with multiple base classes

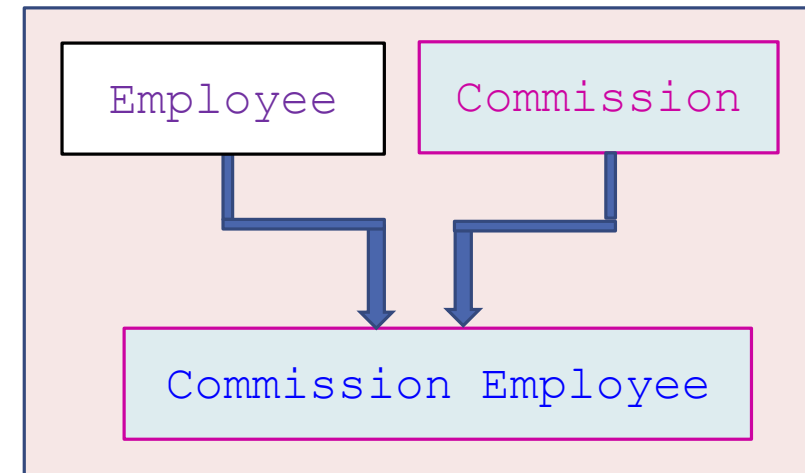
Syntax

```
class DerivedClassName:: access BaseClassName-1, access BaseClassName-2, .....
```

```
class Employee // Employee superclass
{
    private:
        string firstName, string lastName;
    public:
        .....
};

class Commission // Commission superclass
{
    public:
        void setCommissionRate(double rate)
        {
            commissionRate=(rate>0.0&&rate<1.0)?rate:0.0;
        }
};
```

```
class CommissionEmployee : public Employee, public Commission
{
    .....// derived class definition
};
```



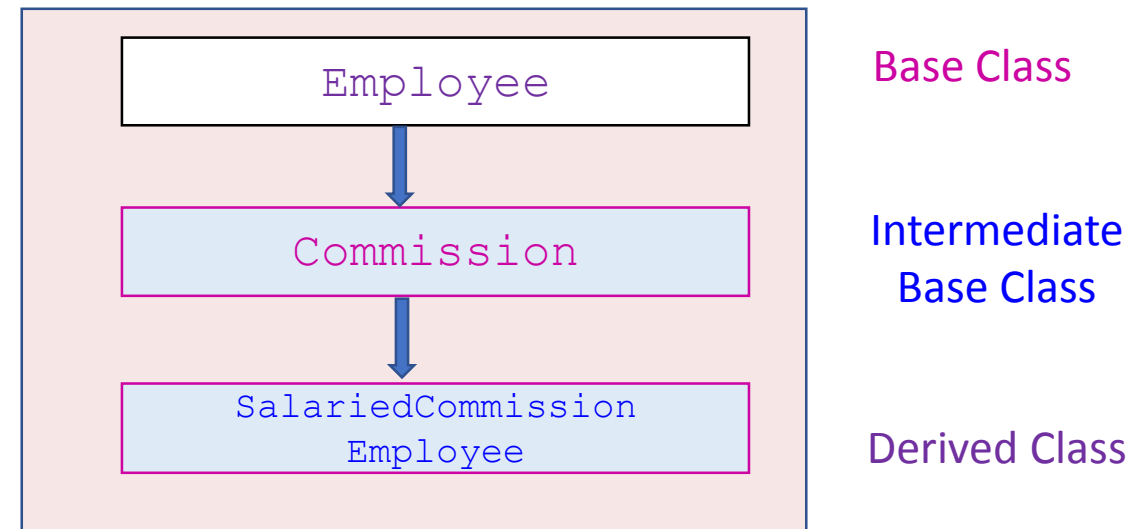
Multilevel Inheritance

Subclass can be created from another intermediate subclass

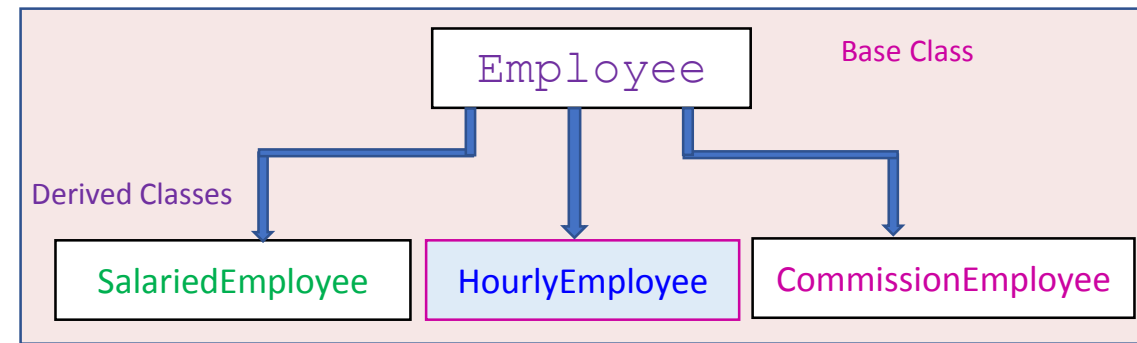
```
class Employee // Employee superclass
{
    private:
        string firstName, string lastName;
    public:
        .....
};

class Commission: public Employee // Commission Subclass
{
    public:
    void setCommissionRate(double rate)
    {
        commissionRate=(rate>0.0&&rate<1.0)?rate:0.0;
    }
};
```

```
class SalariedCommissionEmployee : public Commission
{
    .....// derived class definition
};
```



Hierarchical Inheritance



Multiple subclasses have only one base class

```
class Employee // Employee superclass
{
    private:
        string firstName, string lastName;
    public:
        .....
};

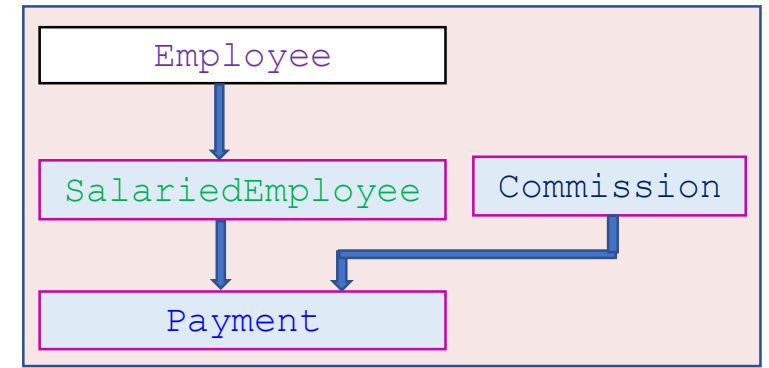
class SalariedEmployee : public Employee // Subclass
{
    public:
    void setWeeklySalary (double salary)
    {
        weeklySalary=salary<0.0?0.0:salary;
    }
};
```

```
class HourlyEmployee : public Employee // Subclass
{ public:
    void setHours(double hoursWorked)
    { .....
    }
};

class CommissionEmployee: public Employee //Subclass
{
    public:
    void setCommissionRate(double rate)
    {
        commissionRate=(rate>0.0&&rate<1.0)?rate:0.0;
    }
};
```

Hybrid Inheritance

Any legal combination of other four types of inheritance



```
class Employee // Employee superclass
{
    private:
        string firstName, string
lastName;
    public:
        .....
};
class SalariedEmployee : public Employee
{
    public:
    void setWeeklySalary (double salary)
    {
        weeklySalary=salary<0.0?0.0:salary;
    }
};
```

```
class Commission // Commission Subclass
{
    public:
    void setCommissionRate(double rate)
    {
        commissionRate=(rate>0.0&&rate<1.0)?rate:0.0;
    }
};
class Payment : public SalariedEmployee, public Commission
{
    double earnings()
    { ..... }
};
```


Problem Statement_Assignment 2

Design and develop inheritance for a given case study, identify objects and relationships and implement inheritance wherever applicable.

- **Employee** class has Emp_name, Emp_id, Address, Mail_id, and Mobile_no as data members.
- Inherit the classes:
 - ✓ Programmer
 - ✓ Team Lead
 - ✓ Assistant Project Manager and
 - ✓ Project Manager from employee class.
- Add Basic Pay as the member of all the inherited classes with 97% of Basic Pay as DA, 10 % of Basic Pay as HRA, 12% of Basic Pay as PF, 0.1% of Basic Pay for staff club fund.
- Generate pay slips for the employees with their gross and net salary.

Problem Statement_Assignment 2_Solution using C++

Demonstrating use of Inheritance in Java Programming

Inheritance in Java

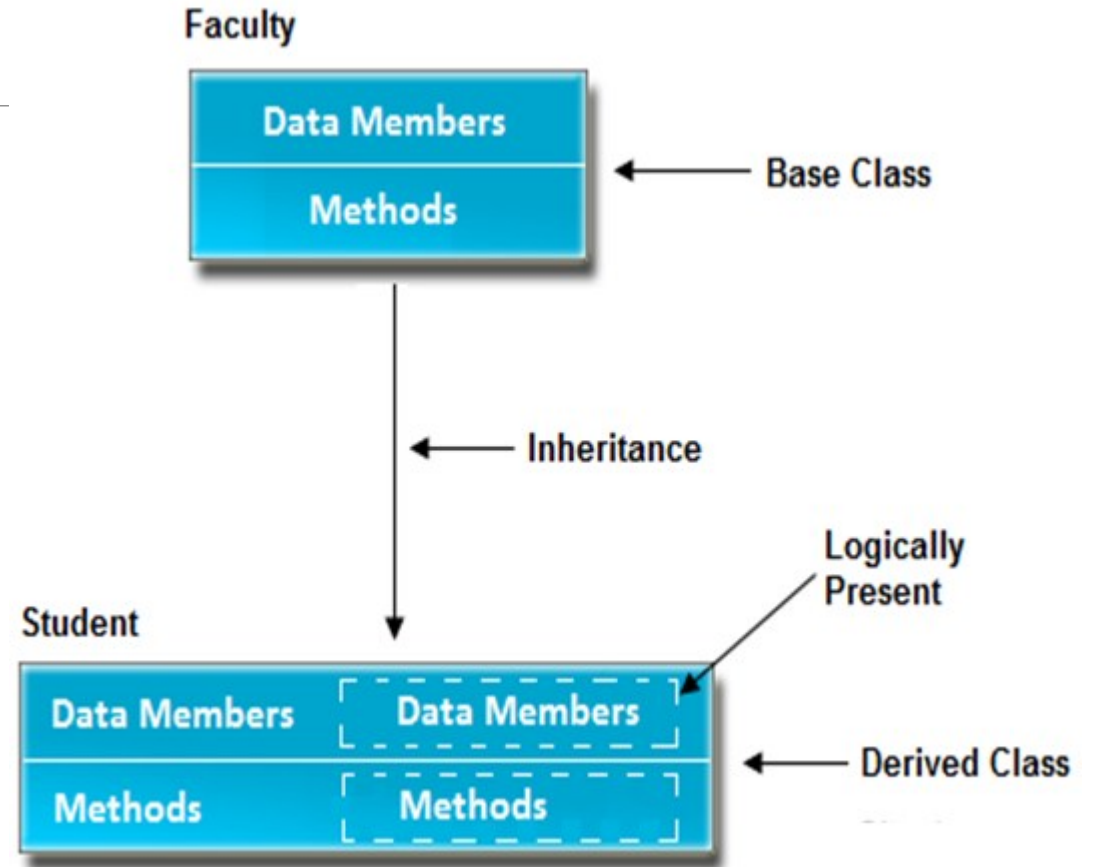
- Inheritance is an important pillar of OOP(Object-Oriented Programming).
- It is the mechanism in java by which one class is allowed to inherit the features(fields and methods) of another class.
- **Important terminology:**
- **Super Class:** The class whose features are inherited is known as superclass(or a base class or a parent class).
- **Sub Class:** The class that inherits the other class is known as a subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability:** Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Benefits of using inheritance

If we develop any application using concept of Inheritance then that application has following advantages:

- Requires less Application development time.
- Application takes less memory.
- Application execution time is less.
- Application performance is enhanced (improved).
- Redundancy (repetition) of the code is reduced or minimized so that we get consistent results and less storage cost.

Inheritance_Example



In the above diagram data members and methods are represented in broken line are inherited from faculty class and they are visible in student class logically.

Syntax of Inheritance in Java

The keyword used for inheritance is **extends**

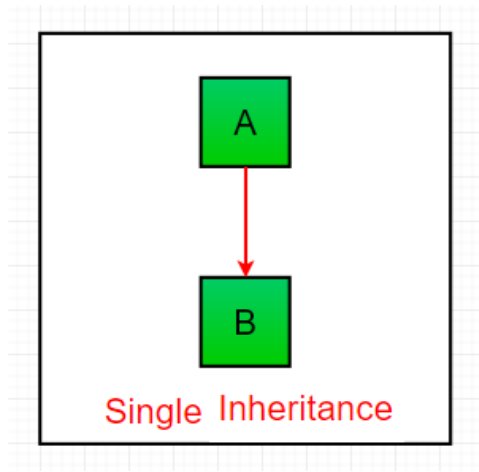
Syntax :

```
class derived-class_Name extends base-class_Name
{
    //methods and fields
}
```

Types of Inheritance in Java

1. Single Inheritance:

In single inheritance, subclasses inherit the features of one superclass. In the image below, class A serves as a base class for the derived class B.



Single Inheritance_Example

```
import java.io.*;
import java.lang.*;
import java.util.*;
```

```
class one {
    public void print_geek()
    {
        System.out.println("Geeks");
    }
}
```

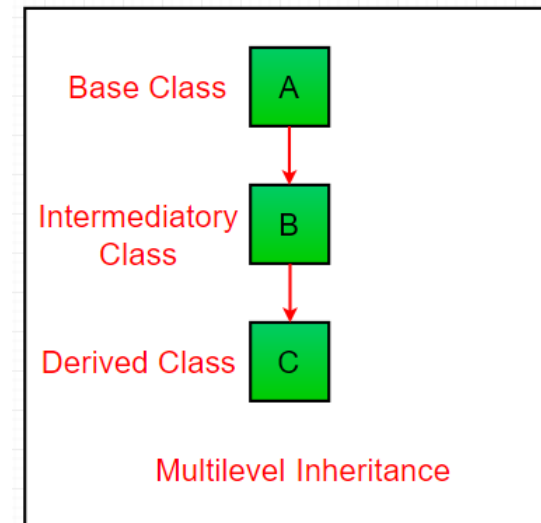
```
class two extends one {
    public void print_for() { System.out.println("for"); }
}
// Driver class
public class Main {
    public static void main(String[] args)
    {
        two g = new two();
        g.print_geek();
        g.print_for();
        g.print_geek();
    }
}
```

Output

Geeks
for
Geeks

Multilevel Inheritance

- In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class.
- In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the grandparent's members.



// Java program to illustrate the concept of Multilevel inheritance

```
import java.io.*;
import java.lang.*;
import java.util.*;
class one {
public void print_geek()
{
System.out.println("Geeks");
}
}
```

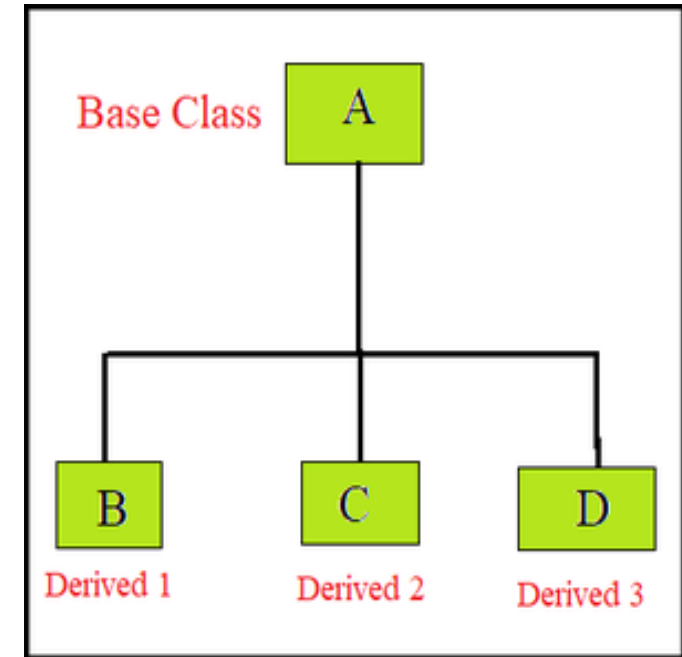
```
class two extends one {
public void print_for() {
System.out.println("for");
}
}
```

```
class three extends two {
public void print_geek()
{
System.out.println("Geeks");
}
}
// Derived class
public class Main {
public static void main(String[] args)
{
three g = new three();
g.print_geek();
g.print_for();
g.print_geek();
}
}
```

Output
Geeks
for
Geeks

Hierarchical Inheritance

- In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass.
- In the below image, class A serves as a base class for the derived class B, C and D.



// Java program to illustrate the concept of Hierarchical inheritance

```
class A {  
    public void print_A() {  
        System.out.println("Class A");  
    }  
}
```

```
class B extends A {  
    public void print_B() {  
        System.out.println("Class B");  
    }  
}
```

```
class C extends A {  
    public void print_C() {  
        System.out.println("Class C");  
    }  
}
```

```
class D extends A {  
    public void print_D() {  
        System.out.println("Class D");  
    }  
}
```

// Driver Class

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

```
        B obj_B = new B();
```

```
        obj_B.print_A();
```

```
        obj_B.print_B();
```

```
        C obj_C = new C();
```

```
        obj_C.print_A();
```

```
        obj_C.print_C();
```

```
        D obj_D = new D();
```

```
        obj_D.print_A();
```

```
        obj_D.print_D();
```

```
    }
```

```
}
```

Output

Class A

Class B

Class A

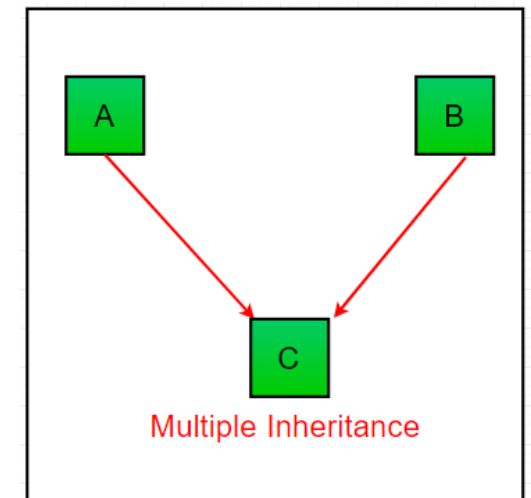
Class C

Class A

Class D

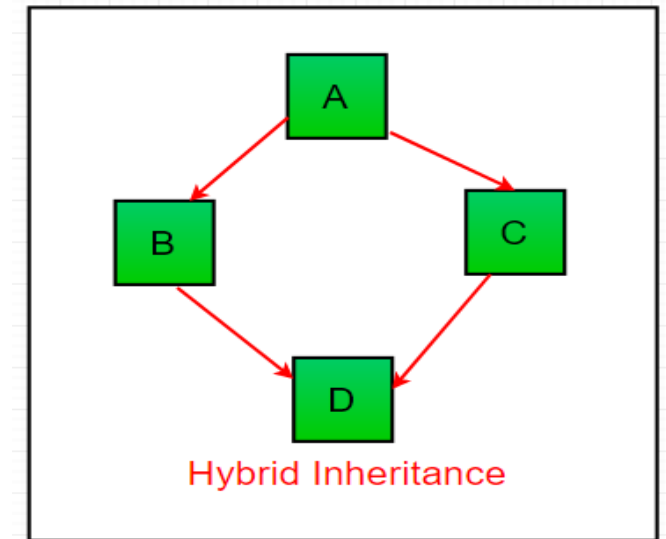
Multiple Inheritance (Through Interfaces)

- In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes.
- Please note that Java does **not** support multiple inheritances with classes.
- In Java, we can achieve multiple inheritances only through Interfaces.
- In the image below, Class C is derived from interface A and B.

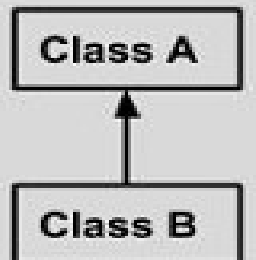
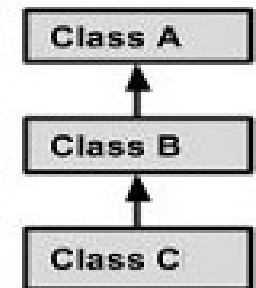
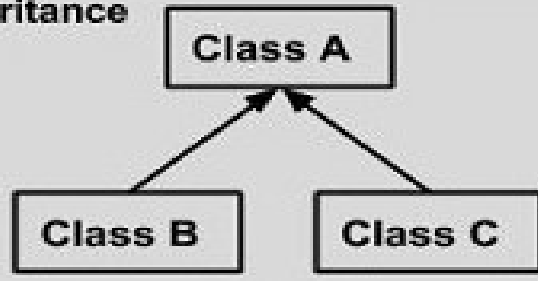
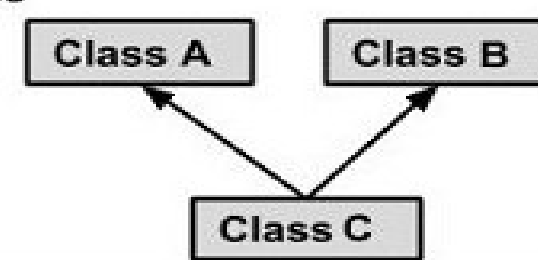


Hybrid Inheritance (Through Interfaces)

- It is a mix of two or more of the above types of inheritance.
- Since java doesn't support multiple inheritances with classes, hybrid inheritance is **also not possible** with classes.
- In Java, we can achieve hybrid inheritance only through Interfaces.



Java Inheritance Summary

<p>Single Inheritance</p>  <pre> graph BT B[Class B] --> A[Class A] </pre>	<pre> public class A { } public class B extends A { } </pre>
<p>Multi Level Inheritance</p>  <pre> graph BT C[Class C] --> B[Class B] B --> A[Class A] </pre>	<pre> public class A {} public class B extends A {.....} public class C extends B {.....} </pre>
<p>Hierarchical Inheritance</p>  <pre> graph BT B[Class B] --> A[Class A] C[Class C] --> A </pre>	<pre> public class A {} public class B extends A {.....} public class C extends A {.....} </pre>
<p>Multiple Inheritance</p>  <pre> graph BT C[Class C] --> A[Class A] C --> B[Class B] </pre>	<pre> public class A {} public class B {.....} public class C extends A,B { } // Java does not support mutiple Inheritance </pre>

Important facts about inheritance in Java

- **Default superclass:** Except Object class, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of the Object class.
- **Superclass can only be one:** A superclass can have any number of subclasses. But a subclass can have only **one** superclass. This is because Java does not support multiple inheritance with classes. Although with interfaces, multiple inheritances are supported by java.
- **Inheriting Constructors:** A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.
- **Private member inheritance:** A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods (like getters and setters) for accessing its private fields, these can also be used by the subclass.

What all can be done in a Subclass?

- In sub-classes we can inherit members as is, replace them, hide them, or supplement them with new members:
- The inherited fields can be used directly, just like any other fields.
- We can declare new fields in the subclass that are not in the superclass.
- The inherited methods can be used directly as they are.
- We can write a new instance method in the subclass that has the same signature as the one in the superclass, thus overriding it (as in the example above, toString() method is overridden).
- We can write a new static method in the subclass that has the same signature as the one in the superclass, thus hiding it.
- We can declare new methods in the subclass that are not in the superclass.
- We can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.

The super keyword

- The **super** keyword is similar to **this** keyword. Following are the scenarios where the super keyword is used.
- It is used to **differentiate the members** of superclass from the members of subclass, if they have same names.
- It is used to **invoke the superclass** constructor from subclass.
- Differentiating the Members
- If a class is inheriting the properties of another class. And if the members of the superclass have the names same as the sub class, to differentiate these variables we use super keyword as shown below.
 - `super.variable`
 - `super.method();`

super keyword_Example

```
class Super_class {  
    int num = 20;  
    // display method of superclass  
    public void display() {  
        System.out.println("This is the display method of superclass");  
    }  
}  
  
public class Sub_class extends Super_class {  
    int num = 10;  
    // display method of sub class  
    public void display() {  
        System.out.println("This is the display method of subclass");  
    }  
    public void my_method() {  
        // Instantiating subclass  
        Sub_class sub = new Sub_class();  
        // Invoking the display() method of sub class
```

```
        sub.display();  
        // Invoking the display() method of superclass  
        super.display();  
        // printing the value of variable num of subclass  
        System.out.println("value of the variable named  
num in sub class:"+ sub.num);  
        // printing the value of variable num of  
superclass  
        System.out.println("value of the variable named  
num in super class:"+ super.num);  
    }  
  
    public static void main(String args[]) {  
        Sub_class obj = new Sub_class();  
        obj.my_method();  
    }  
}
```

Output
This is the display method of subclass
This is the display method of superclass
value of the variable named num in sub class:10
value of the variable named num in super class:20

Final keyword

- In Java, the final keyword can be used while declaring a variable, class, or method to make the value unchangeable.
- The value of the entity is decided at initialization and will remain immutable throughout the program.
- Attempting to change the value of anything declared as final will throw a compiler error.

// declaring a final variable

```
class FinalVariable {  
    final int var = 50;  
    var = 60 //This line would give an error  
}
```

- The exact behavior of final depend on the type of entity:
- final Parameter cannot be changed anywhere in the function
- final Method cannot be overridden or hidden by any subclass
- final Class cannot be a parent class for any subclass

Problem Statement for Practice

Write a Java Program for demonstrating Inheritance in Java.

Write a program in Java showing hierarchical inheritance with base class as Employee and derived classes as FullTimeEmployee and InternEmployee with methods DisplaySalary in base class and CalculateSalary in derived classes.

Calculate salary method will calculate as per increment given to fulltime and intern Employees. Fulltime employee- 50% hike, Intern employee-25% hike. Display salary before and after hike.

Abstraction in Java

Abstraction in Java

- A process of hiding the implementation details and showing only functionality to the user.
- It shows only essential things to the user and hides the internal details
- e.g. sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.
- It focuses on what the object does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class
2. Interface

Abstract class in Java

- A class which is declared as abstract is known as an **abstract class**.
- It can have abstract and non-abstract methods.
- It needs to be extended and its method implemented.
- A normal class cannot have abstract methods.

Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

When to use Abstract Methods & Abstract Class?

- **Abstract methods** are usually declared where two or more subclasses are expected to do a similar thing in different ways through different implementations.
 - These subclasses extend the same Abstract class and provide different implementations for the abstract methods.
 - can only be used in an abstract class, and it does not have a body.
 - The body is provided by the subclass (inherited from).
-
- **Abstract classes** are used to define generic types of behaviors at the top of an object-oriented programming class hierarchy, and use its subclasses to provide implementation details of the abstract class.
 - is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

An Abstract Class Example

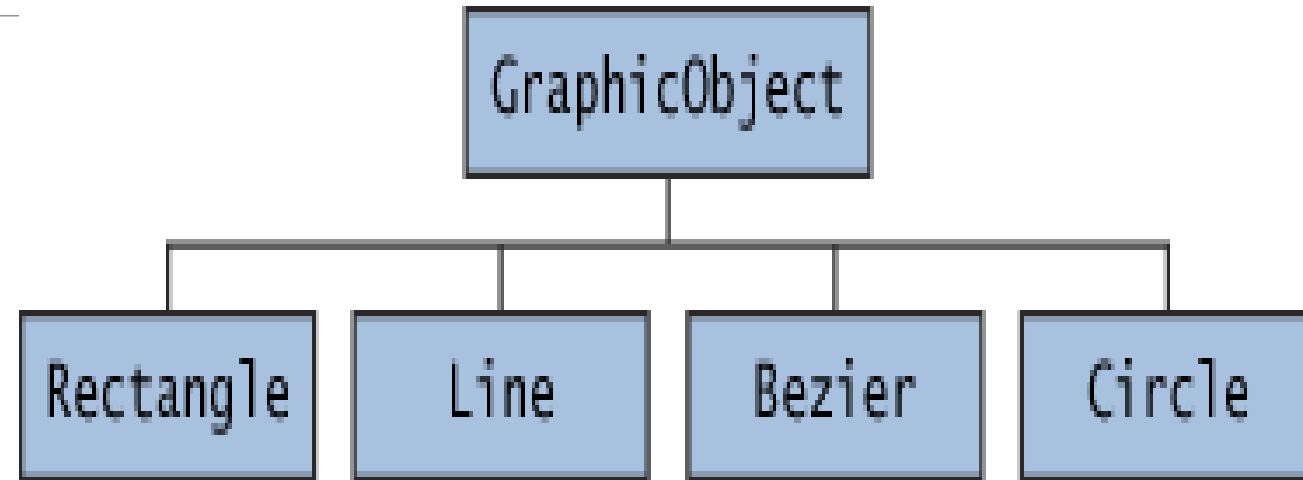
```
abstract class Animal {  
    public abstract void animalSound();  
    public void sleep() {  
        System.out.println("Zzz");  
    }  
}
```

`Animal myObj = new Animal();` `// will generate an error`

- To access the abstract class, it must be inherited from another class.

An Abstract Class Example

```
abstract class GraphicObject {  
    int x, y;  
  
    ...  
  
    void moveTo(int newX, int newY) {  
        ...  
    }  
  
    abstract void draw();  
    abstract void resize();  
}
```



nonabstract subclass of `GraphicObject`, such as `Circle` and `Rectangle`, must provide implementations for the `draw` and `resize` methods:

An Abstract Class Example

```
class Circle extends GraphicObject {  
    void draw() {  
        ...  
    }  
    void resize() {  
        ...  
    }  
}  
class Rectangle extends GraphicObject {  
    void draw() {  
        ...  
    }  
    void resize() {  
        ...  
    }  
}
```

An Abstract Class Example

```
// Abstract class
abstract class Animal {
    // Abstract method (does not have a body)
    public abstract void animalSound();
    // Regular method
    public void sleep() {
        System.out.println("Zzz");
    }
}

// Subclass (inherit from Animal)
class Pig extends Animal {
    public void animalSound() {
        // The body of animalSound() is provided here
        System.out.println("The pig says: wee wee");
    }
}
```

```
class Main {
    public static void main(String[] args) {
        Pig myPig = new Pig(); // Create a Pig object
        myPig.animalSound();
        myPig.sleep();
    }
}
```

Output

```
The pig says: wee wee
Zzz
```

An Abstract Class Example

```
abstract class Bike{  
    abstract void run();  
}  
class Honda4 extends Bike{  
    void run(){  
        System.out.println("running safely");  
    }  
    public static void main(String args[]){  
        Bike obj = new Honda4();  
        obj.run();  
    }  
}
```

Output



running safely

An Abstract Class Example

Shape is the abstract class, and its implementation is provided by the Rectangle and Circle classes.

```
abstract class Shape{
    abstract void draw();
}
//In real scenario, implementation is provided by others i.e. unknown by end user
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();//In a real scenario, object is provided through method, e.g., getShape() method
        s.draw();
    }
}
```

Output

drawing circle

An Abstract Class Example

```
abstract class Bank{  
    abstract int getRateOfInterest();  
}  
  
class SBI extends Bank{  
    int getRateOfInterest(){return 7;}  
}  
  
class PNB extends Bank{  
    int getRateOfInterest(){return 8;}  
}
```

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

Output

```
Rate of Interest is: 7 %  
Rate of Interest is: 8 %
```

An Abstract Class Example

```
abstract class MotorBike {  
    abstract void brake();  
}  
class SportsBike extends MotorBike {  
  
    // implementation of abstract method  
    public void brake() {  
        System.out.println("SportsBike Brake");  
    }  
}  
class MountainBike extends MotorBike {  
    // implementation of abstract method  
    public void brake() {  
        System.out.println("MountainBike Brake");  
    }  
}
```

```
class Main {  
    public static void main(String[] args)  
    {  
        MountainBike m1 = new  
MountainBike();  
        m1.brake();  
        SportsBike s1 = new SportsBike();  
        s1.brake();  
    }  
}
```

Output

```
MountainBike Brake  
SportsBike Brake
```

Abstract class having constructor, data member and methods

//Example of an abstract class that has abstract and non-abstract methods

```
abstract class Bike{  
    Bike(){System.out.println("bike is created");}  
    abstract void run();  
    void changeGear(){System.out.println("gear changed");}  
}
```

//Creating a Child class which inherits Abstract class

```
class Honda extends Bike{  
    void run(){System.out.println("running safely..");}  
}
```

//Creating a Test class which calls abstract and non-abstract methods

```
class TestAbstraction2{  
    public static void main(String args[]){  
        Bike obj = new Honda();  
        obj.run();  
        obj.changeGear();  
    }  
}
```

Output

```
bike is created  
running safely..  
gear changed
```

Accesses Constructor of Abstract Classes

To access the constructor of an abstract class from the subclass using the **super** keyword

```
abstract class Animal {  
    Animal() {  
        ....  
    }  
}  
  
class Dog extends Animal {  
    Dog() {  
        super();  
        ...  
    }  
}
```

Why And When To Use Abstract Classes and Methods?

- To achieve security
- To hide certain details and only show the important details of an object

Key Points to Remember

- We use the abstract keyword to create abstract classes and methods.
- An abstract method doesn't have any implementation (method body).
- A class containing abstract methods should also be abstract.
- We cannot create objects of an abstract class.
- To implement features of an abstract class, we inherit subclasses from it and create objects of the subclass.
- A subclass must override all abstract methods of an abstract class. However, if the subclass is declared abstract, it's not mandatory to override abstract methods.
- We can access the static attributes and methods of an abstract class using the reference of the abstract class.
- For example, `Animal.staticMethod();`

Interfaces in Java

Interfaces

- An interface is a completely "abstract class" that is used to group related methods with empty bodies
- To access the interface methods, the interface must be "**implemented**" by another class with the implements keyword (instead of extends)

```
interface Animal {  
    public void animalSound(); // interface method (does not have a body)  
    public void run(); // interface method (does not have a body)  
}
```

Interface_Example 1

```
interface Animal {  
    public void animalSound(); // interface method (does not have a body)  
    public void sleep(); // interface method (does not have a body)  
}
```

// Pig "implements" the Animal interface

```
class Pig implements Animal {  
    public void animalSound() {  
        // The body of animalSound() is provided here  
        System.out.println("The pig says: wee wee");  
    }  
    public void sleep() {  
        // The body of sleep() is provided here  
        System.out.println("Zzz");  
    }  
}
```

```
class Main {  
    public static void main(String[] args) {  
        Pig myPig = new Pig(); // Create a  
        Pig object  
        myPig.animalSound();  
        myPig.sleep();  
    }  
}
```

```
The pig says: wee wee  
Zzz
```

Interface_Example 2

```
interface Polygon {  
    void getArea(int length, int breadth);  
}
```

```
// implement the Polygon interface  
class Rectangle implements Polygon {
```

```
    // implementation of abstract method  
    public void getArea(int length, int breadth) {  
        System.out.println("The area of the rectangle is " + (length * breadth));  
    }  
}
```

```
class Main {  
    public static void main(String[] args) {  
        Rectangle r1 = new Rectangle();  
        r1.getArea(5, 6);  
    }  
}
```

```
The area of the rectangle is 30
```

Interface_Example 3

```
interface Language {  
    void getName(String name);  
}
```

```
// class implements interface  
class ProgrammingLanguage implements  
    Language {
```

```
    // implementation of abstract method  
    public void getName(String name) {  
        System.out.println("Programming  
Language: " + name);  
    }  
}
```

```
class Main {  
    public static void main(String[] args) {  
        ProgrammingLanguage language = new  
        ProgrammingLanguage();  
        language.getName("Java");  
    }  
}
```

```
Programming Language: Java
```

Interfaces

- Like abstract classes, interfaces cannot be used to create objects (in the example above, it is not possible to create an "Animal" object in the MyMainClass)
- Interface methods do not have a body - the body is provided by the "implement" class
- On implementation of an interface, you must override all of its methods
- Interface methods are by default abstract and public
- Interface attributes are by default public, static and final
- An interface cannot contain a constructor (as it cannot be used to create objects)

Why And When To Use Interfaces?

- To achieve security - hide certain details and only show the important details of an object (interface).
- Java does not support "**multiple inheritance**" (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can implement multiple interfaces.

Multiple Interfaces

```
interface FirstInterface {  
    public void myMethod(); // interface method  
}
```

```
interface SecondInterface {  
    public void myOtherMethod(); // interface method  
}
```

```
class DemoClass implements FirstInterface, SecondInterface {  
    public void myMethod() {  
        System.out.println("Some text..");  
    }  
    public void myOtherMethod() {  
        System.out.println("Some other text...");  
    }  
}
```

```
class Main {  
    public static void main(String[] args) {  
        DemoClass myObj = new DemoClass();  
        myObj.myMethod();  
        myObj.myOtherMethod();  
    }  
}
```

```
Some text...  
Some other text...
```

Implementing Multiple Interfaces

```
interface A {  
    // members of A  
}
```

```
interface B {  
    // members of B  
}
```

```
class C implements A, B {  
    // abstract members of A  
    // abstract members of B  
}
```


Extending an Interface

```
interface Line {  
    // members of Line interface  
}  
  
// extending interface  
interface Polygon extends Line {  
    // members of Polygon interface  
    // members of Line interface  
}
```

Extending Multiple Interface

```
interface A {
```

```
    ...
```

```
}
```

```
interface B {
```

```
    ...
```

```
}
```

```
interface C extends A, B {
```

```
    ...
```

```
}
```

Interfaces

- All the methods inside an interface are implicitly public and all fields are implicitly public static final

```
interface Language {
```

```
    // by default public static final
```

```
    String type = "programming language";
```

```
    // by default public
```

```
    void getName();
```

```
}
```

- In Java, protected members of a class “A” are accessible in other class “B” of the same package, even if B doesn’t inherit from A (they both have to be in the same package).

Comparison of Inheritance in C++ and Java

- In Java, all classes inherit from the Object class directly or indirectly.
- In Java, members of the grandparent class are not directly accessible.
- The meaning of protected member access specifier is somewhat different in Java.
- Java uses ‘extends’ keywords for inheritance.
- In Java, methods are virtual by default. In C++, we explicitly use virtual keywords.
- Java uses a separate keyword *interface* for interfaces and *abstract* keywords for abstract classes and abstract functions.
- Unlike C++, Java doesn’t support multiple inheritances.
- Like C++, the default constructor of the parent class is automatically called in Java, but if we want to call parameterized constructor then we must use super to call the parent constructor

Problem Statement_Assignment 2_Implementing Interface in Java

Write a java program to create two interfaces Motorbike and Cycle.

- Motorbike interface consists of the attribute speed.
- The method is totalDistance().
- Cycle interface consists of the attributes distance and the method speed().
- These interfaces are implemented by the class TwoWheeler.
- Calculate total distance travelled and Average Speed maintained by Two Wheeler.

FAQs

- What is inheritance in C++ and give examples of the different types of inheritance?
- What's the difference between public, private, and protected?
- Why can't derived class access private things from base class?

References

- https://www.tutorialspoint.com/java/java_inheritance.htm
- <https://beginnersbook.com/2013/03/inheritance-in-java/>
- <https://www.geeksforgeeks.org/java-and-multiple-inheritance/>
- <https://www.mygreatlearning.com/blog/inheritance-in-java/>
- https://www.w3schools.com/java/java_inner_classes.asp

Thank You!!