Inheritance in Java

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. In Java, Inheritance means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and fields of that class. In addition, you can add new fields and methods to your current class as well.

Why Do We Need Java Inheritance?

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

Important Terminologies Used in Java Inheritance

- **Class:** Class is a set of objects which shares common characteristics/ behavior and common properties/ attributes. Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
- **Super Class/Parent Class:** The class whose features are inherited is known as a superclass(or a base class or a parent class).
- **Sub Class/Child 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.

How to Use Inheritance in Java?

The **extends keyword** is used for inheritance in Java. Using the extends keyword indicates you are derived from an existing class. In other words, “extends” refers to increased functionality.

Syntax :

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

Inheritance in Java Example

Example: In the below example of inheritance, class Bicycle is a base class, class MountainBike is a derived class that extends the Bicycle class and class Test is a driver class to run the program.

Java

```java
// Java program to illustrate the
// concept of inheritance

// base class
class Bicycle {
    // the Bicycle class has two fields
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int gear, int speed)
    {
        this.gear = gear;
        this.speed = speed;
    }

    // the Bicycle class has three methods
    public void applyBrake(int decrement)
    {
        speed -= decrement;
    }

    public void speedUp(int increment)
    {
        speed += increment;
    }

    // toString() method to print info of Bicycle
    public String toString()
    {
        return ("No of gears are " + gear + "\n"
                + "speed of bicycle is " + speed);
    }
}

// derived class
class MountainBike extends Bicycle {

    // the MountainBike subclass adds one more field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int gear, int speed,
                        int startHeight)
    {
        // invoking base-class(Bicycle) constructor
        super(gear, speed);
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one more method
    public void setHeight(int newValue)
    {
        seatHeight = newValue;
```

```
    }

    // overriding toString() method
    // of Bicycle to print more info
    @Override public String toString()
    {
        return (super.toString() + "\nseat height is "
                + seatHeight);
    }
}

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

        MountainBike mb = new MountainBike(3, 100, 25);
        System.out.println(mb.toString());
    }
}
```

Output

```
No of gears are 3
speed of bicycle is 100
seat height is 25
```

In the above program, when an object of MountainBike class is created, a copy of all methods and fields of the superclass acquires memory in this object. That is why by using the object of the subclass we can also access the members of a superclass.

Please note that during inheritance only the object of the subclass is created, not the superclass. For more, refer to Java Object Creation of Inherited Class.

Example 2: In the below example of inheritance, class Employee is a base class, class Engineer is a derived class that extends the Employee class and class Test is a driver class to run the program.

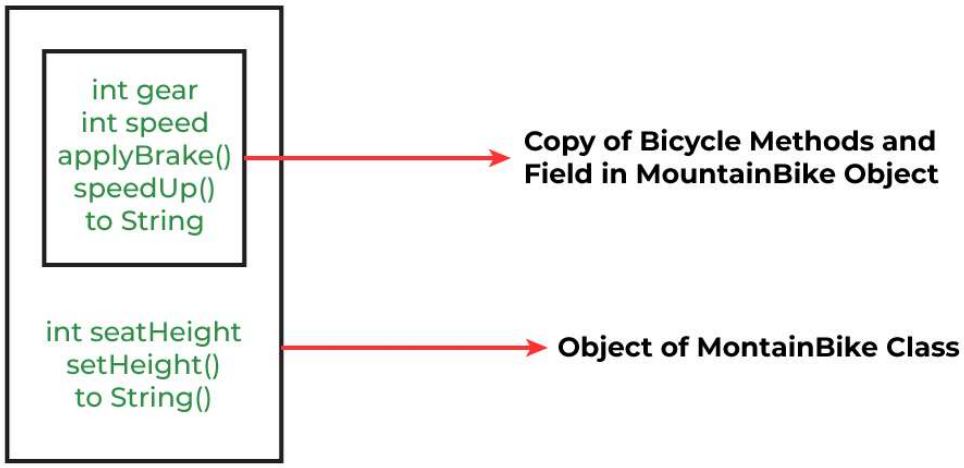
Java

```
// Java Program to illustrate Inheritance (concise)

import java.io.*;

// Base or Super Class
class Employee {
    int salary = 60000;
}

// Inherited or Sub Class
class Engineer extends Employee {
    int benefits = 10000;
}

// Driver Class
class Gfg {
    public static void main(String args[])
    {
        Engineer E1 = new Engineer();
        System.out.println("Salary : " + E1.salary
                           + "\nBenefits : " + E1.benefits);
    }
}
```

Output

```
Salary : 60000
Benefits : 10000
```

Illustrative image of the program:



In practice, inheritance, and polymorphism are used together in Java to achieve fast performance and readability of code.

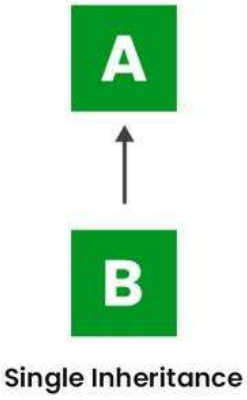
Java Inheritance Types

Below are the different types of inheritance which are supported by Java.

1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance
5. Hybrid Inheritance

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

Java

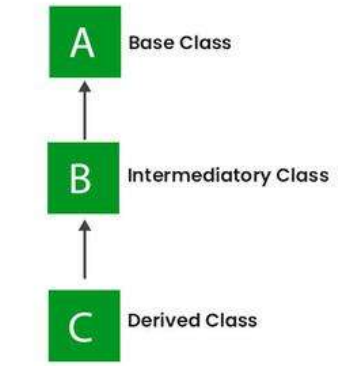
```java
// Java program to illustrate the
// concept of single inheritance
import java.io.*;
import java.lang.*;
import java.util.*;

// Parent class
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 {
      // Main function
    public static void main(String[] args)
    {
        two g = new two();
        g.print_geek();
        g.print_for();
        g.print_geek();
    }
}
```

Output

```
Geeks
for
Geeks
```

2. Multilevel Inheritance

In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes. 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.



Multilevel Inheritance

Java

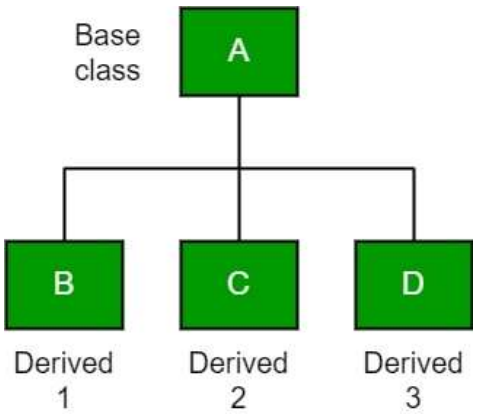
```java
// Java program to illustrate the
// concept of Multilevel inheritance
import java.io.*;
import java.lang.*;
import java.util.*;

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");
    }
}

// Drived 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
```

3. 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 classes B, C, and D.



Java

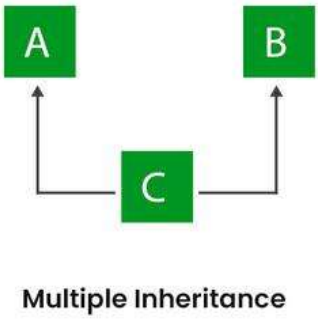
```java
// Java program to illustrate the
// concept of Hierarchical  inheritance

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
```

4. 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 interfaces A and B.



Multiple Inheritance

Java

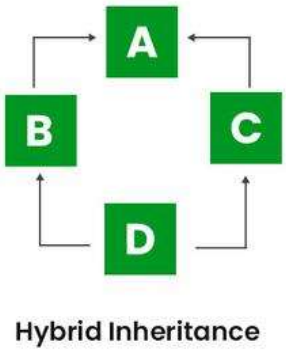
```java
// Java program to illustrate the
// concept of Multiple inheritance
import java.io.*;
import java.lang.*;
import java.util.*;

interface one {
    public void print_geek();
}

interface two {
    public void print_for();
}

interface three extends one, two {
    public void print_geek();
}
class child implements three {
    @Override public void print_geek()
    {
        System.out.println("Geeks");
    }

    public void print_for() { System.out.println("for"); }
}

// Drived class
public class Main {
    public static void main(String[] args)
    {
        child c = new child();
        c.print_geek();
        c.print_for();
        c.print_geek();
    }
}
```

Output

```
Geeks
for
```

5. 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.



Hybrid Inheritance

Java IS-A type of Relationship

IS-A is a way of saying: This object is a type of that object. Let us see how the extends keyword is used to achieve inheritance.

Java

```
public class SolarSystem {
}
public class Earth extends SolarSystem {
}
public class Mars extends SolarSystem {
}
public class Moon extends Earth {
}
```

Now, based on the above example, in Object-Oriented terms, the following are true:-

- SolarSystem is the superclass of Earth class.
- SolarSystem is the superclass of Mars class.
- Earth and Mars are subclasses of SolarSystem class.
- Moon is the subclass of both Earth and SolarSystem classes.

Java

```
class SolarSystem {
}
class Earth extends SolarSystem {
}
class Mars extends SolarSystem {
}
public class Moon extends Earth {
    public static void main(String args[])
    {
        SolarSystem s = new SolarSystem();
        Earth e = new Earth();
```

```
        Mars m = new Mars();

        System.out.println(s instanceof SolarSystem);
        System.out.println(e instanceof Earth);
        System.out.println(m instanceof SolarSystem);
    }
}
```

Output

```
true
true
true
```

What 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.

Advantages Of Inheritance in Java:

1. Code Reusability: Inheritance allows for code reuse and reduces the amount of code that needs to be written. The subclass can reuse the properties and methods of the superclass, reducing duplication of code.
2. Abstraction: Inheritance allows for the creation of abstract classes that define a common interface for a group of related classes. This promotes abstraction and encapsulation, making the code easier to maintain and extend.
3. Class Hierarchy: Inheritance allows for the creation of a class hierarchy, which can be used to model real-world objects and their relationships.
4. Polymorphism: Inheritance allows for polymorphism, which is the ability of an object to take on multiple forms. Subclasses can override the methods of the superclass, which allows them to change their behavior in different ways.

Disadvantages of Inheritance in Java:

1. Complexity: Inheritance can make the code more complex and harder to understand. This is especially true if the inheritance hierarchy is deep or if multiple inheritances is used.
2. Tight Coupling: Inheritance creates a tight coupling between the superclass and subclass, making it difficult to make changes to the superclass without affecting the subclass.

Conclusion

Let us check some important points from the article are mentioned below:

- **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 inheritances 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.

FAQs in Inheritance

1. What is Inheritance Java?

Inheritance is a concept of OOPs where one class inherits from another class that can reuse the methods and fields of the parent class.

2. What are the 4 types of inheritance in Java?

There are Single, Multiple, Multilevel, and Hybrid.

3. What is the use of extend keyword?

Extend keyword is used for inheriting one class into another.

4. What is an example of inheritance in Java?

A real-world example of Inheritance in Java is mentioned below:

> *Consider a group of vehicles. You need to create classes for Bus, Car, and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be the same for all three classes.*

References Used:

1. *"Head First Java" by Kathy Sierra and Bert Bates*
2. *"Java: A Beginner's Guide" by Herbert Schildt*
3. *"Java: The Complete Reference" by Herbert Schildt*
4. *"Effective Java" by Joshua Bloch*
5. *"Java: The Good Parts" by Jim Waldo.*

Last Updated : 17 Apr, 2023

Similar Reads

1. Inheritance and Constructors in Java
2. Java and Multiple Inheritance

Inheritance and Constructors in Java

Constructors in Java are used to initialize the values of the attributes of the object serving the goal to bring Java closer to the real world. We already have a default constructor that is called automatically if no constructor is found in the code. But if we make any constructor say parameterized constructor in order to initialize some attributes then it must write down the default constructor because it now will be no more automatically called.

> ***Note:*** *In Java, constructor of the base class with no argument gets automatically called in the derived class constructor.*

Example:

Java

```
// Java Program to Illustrate
// Invocation of Constructor
// Calling Without Usage of
// super Keyword

// Class 1
// Super class
class Base {

    // Constructor of super class
    Base()
    {
        // Print statement
        System.out.println(
            "Base Class Constructor Called ");
    }
}

// Class 2
// Sub class
class Derived extends Base {
```

```
    // Constructor of sub class
    Derived()
    {

        // Print statement
        System.out.println(
            "Derived Class Constructor Called ");
    }
}

// Class 3
// Main class
class GFG {

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

        // Creating an object of sub class
        // inside main() method
        Derived d = new Derived();

        // Note: Here first super class constructor will be
        // called there after derived(sub class) constructor
        // will be called
    }
}
```

Output

```
Base Class Constructor Called
Derived Class Constructor Called
```

Output Explanation: Here first superclass constructor will be called thereafter derived(sub-class) constructor will be called because the constructor call is from top to

bottom. And yes if there was any class that our Parent class is extending then the body of that class will be executed thereafter landing up to derived classes.

But, if we want to call a parameterized constructor of the base class, then we can call it using super(). The point to note is base class constructor call must be the first line in the derived class constructor.

Implementation: super(_x) is the first line-derived class constructor.

Java

```
// Java Program to Illustrate Invocation
// of Constructor Calling With Usage
// of super Keyword

// Class 1
// Super class
class Base {
    int x;

    // Constructor of super class
    Base(int _x) { x = _x; }
}

// Class 2
// Sub class
class Derived extends Base {

    int y;

    // Constructor of sub class
    Derived(int _x, int _y)
    {

        // super keyword refers to super class
        super(_x);
        y = _y;
    }

    // Method of sub class
    void Display()
    {
        // Print statement
        System.out.println("x = " + x + ", y = " + y);
    }
}

// Class 3
// Main class
public class GFG {

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

```
        // Creating object of sub class
        // inside main() method
        Derived d = new Derived(10, 20);

        // Invoking method inside main() method
        d.Display();
    }
}
```

Output

```
x = 10, y = 20
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Last Updated : 19 Jul, 2022

86

Similar Reads

1. Java - Exception Handling With Constructors in Inheritance
2. What are Java Records and How to Use them Alongside Constructors and Methods?
3. Private Constructors and Singleton Classes in Java
4. Order of execution of Initialization blocks and Constructors in Java
5. Generic Constructors and Interfaces in Java
6. Java Interview Questions on Constructors
7. Why Constructors are not inherited in Java?
8. User Defined Exceptions using Constructors in Java
9. Properties of Constructors in Java
10. Java Constructors

Related Tutorials

Java and Multiple Inheritance

Multiple Inheritance is a feature of an object-oriented concept, where a class can inherit properties of more than one parent class. The problem occurs when there exist methods with the same signature in both the superclasses and subclass. On calling the method, the compiler cannot determine which class method to be called and even on calling which class method gets the priority.

Note: *Java doesn't support Multiple Inheritance*

Example 1:

Java

```
// Java Program to Illustrate Unsupportance of
// Multiple Inheritance

// Importing input output classes
import java.io.*;

//  Class 1
```

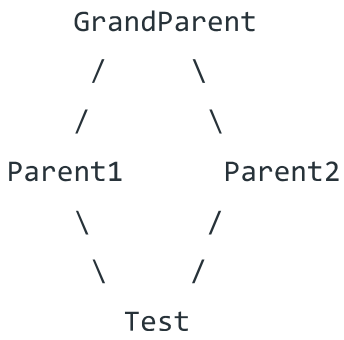
```
// First Parent class
class Parent1 {

  // Method inside first parent class
  void fun() {

    // Print statement if this method is called
    System.out.println("Parent1");
  }
}

// Class 2
// Second Parent Class
class Parent2 {

  // Method inside first parent class
  void fun() {

    // Print statement if this method is called
    System.out.println("Parent2");
  }
}

// Class 3
// Trying to be child of both the classes
class Test extends Parent1, Parent2 {

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

    // Creating object of class in main() method
    Test t = new Test();

    // Trying to call above functions of class where
    // Error is thrown as this class is inheriting
    // multiple classes
    t.fun();
  }
}
```

Output: Compilation error is thrown

```
mayanksolanki@Mayanks-MacBook-Air test % javac GFG.java
GFG.java:228: error: '{' expected
class Test extends Parent1, Parent2 {
                           ^
1 error
mayanksolanki@Mayanks-MacBook-Air test %
```

Conclusion: *As depicted from code above, on calling the method fun() using Test object will cause complications such as whether to call Parent1's fun() or Parent2's fun() method.*

Example 2:



The code is as follows

Java

```
// Java Program to Illustrate Unsupportance of
// Multiple Inheritance
// Diamond Problem Similar Scenario

// Importing input output classes
import java.io.*;

//  Class 1
// A Grand parent class in diamond
class GrandParent {

  void fun() {
```

```
    // Print statement to be executed when this method is called
    System.out.println("Grandparent");
  }
}

// Class 2
// First Parent class
class Parent1 extends GrandParent {
  void fun() {

    // Print statement to be executed when this method is called
    System.out.println("Parent1");
  }
}

// Class 3
// Second Parent Class
class Parent2 extends GrandParent {
  void fun() {

    // Print statement to be executed when this method is called
    System.out.println("Parent2");
  }
}

// Class 4
// Inheriting from multiple classes
class Test extends Parent1, Parent2 {

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

    // Creating object of this class i main() method
    Test t = new Test();

    // Now calling fun() method from its parent classes
    // which will throw compilation error
    t.fun();
  }
}
```

Output:

```
mayanksolanki@Mayanks-MacBook-Air test % javac GFG.java
GFG.java:41: error: '{' expected
class Test extends Parent1, Parent2 {
                            ^
1 error
mayanksolanki@Mayanks-MacBook-Air test %
```

Again it throws compiler error when run fun() method as multiple inheritances cause a diamond problem when allowed in other languages like C++. From the code, we see that: On calling the method fun() using Test object will cause complications such as whether to call Parent1's fun() or Parent2's fun() method. Therefore, in order to avoid such complications, Java does not support multiple inheritances of classes.

Multiple inheritance is not supported by Java using classes, handling the complexity that causes due to multiple inheritances is very complex. It creates problems during various operations like casting, constructor chaining, etc, and the above all reason is that there are very few scenarios on which we actually need multiple inheritances, so better to omit it for keeping things simple and straightforward.

How are the above problems handled for Default Methods and Interfaces?
Java 8 supports default methods where interfaces can provide a default implementation of methods. And a class can implement two or more interfaces. In case both the implemented interfaces contain default methods with the same method signature, the implementing class should explicitly specify which default method is to be used in some method excluding the main() of implementing class using super keyword, or it should

override the default method in the implementing class, or it should specify which default method is to be used in the default overridden method of the implementing class.

Example 3:

Java

```
// Java program to demonstrate Multiple Inheritance
// through default methods

// Interface 1
interface PI1 {

    // Default method
    default void show()
    {

        // Print statement if method is called
        // from interface 1
        System.out.println("Default PI1");
    }
}

// Interface 2
interface PI2 {

    // Default method
    default void show()
    {

        // Print statement if method is called
        // from interface 2
        System.out.println("Default PI2");
    }
}

// Main class
// Implementation class code
class TestClass implements PI1, PI2 {

    // Overriding default show method
      @Override
    public void show()
    {

        // Using super keyword to call the show
        // method of PI1 interface
        PI1.super.show();//Should not be used directly in the main method;

        // Using super keyword to call the show
        // method of PI2 interface
        PI2.super.show();//Should not be used directly in the main method;
    }

      //Method for only executing the show() of PI1
```

```
    public void showOfPI1() {
      PI1.super.show();//Should not be used directly in the main method;
    }

    //Method for only executing the show() of PI2
    public void showOfPI2() {
      PI2.super.show(); //Should not be used directly in the main method;
    }

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

        // Creating object of this class in main() method
        TestClass d = new TestClass();
        d.show();
          System.out.println("Now Executing showOfPI1() showOfPI2()");
          d.showOfPI1();
          d.showOfPI2();
    }
}
```

Output

```
Default PI1
Default PI2
Now Executing showOfPI1() showOfPI2()
Default PI1
Default PI2
```

> ***Note:*** *If we remove the implementation of default method from "TestClass", we get a compiler error. If there is a diamond through interfaces, then there is no issue if none of the middle interfaces provide implementation of root interface. If they provide implementation, then implementation can be accessed as above using super keyword.*

Example 4:

Java

```
// Java program to demonstrate How Diamond Problem
// Is Handled in case of Default Methods

// Interface 1
interface GPI {
```

```
    // Default method
    default void show()
    {

        // Print statement
        System.out.println("Default GPI");
    }
}

// Interface 2
// Extending the above interface
interface PI1 extends GPI {
}

// Interface 3
// Extending the above interface
interface PI2 extends GPI {
}

// Main class
// Implementation class code
class TestClass implements PI1, PI2 {

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

        // Creating object of this class
        // in main() method
        TestClass d = new TestClass();

        // Now calling the function defined in interface 1
        // from whom Interface 2and 3 are deriving
        d.show();
    }
}
```

Output

```
Default GPI
```

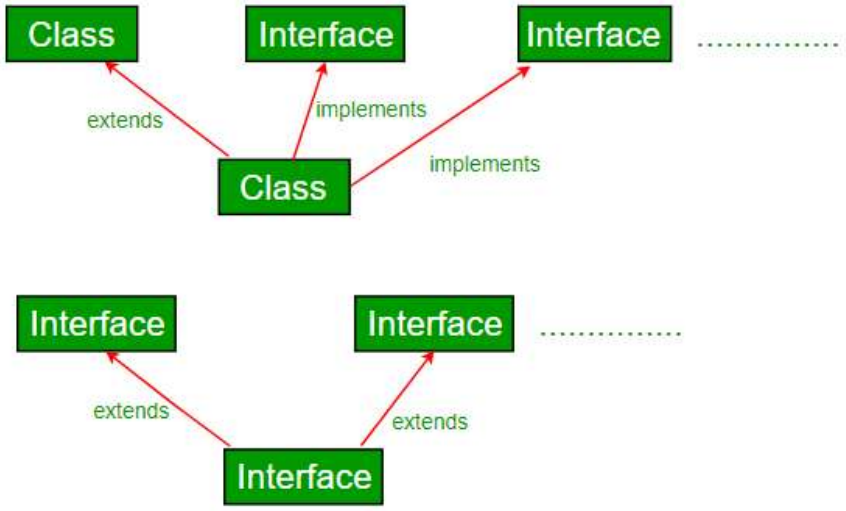
Last Updated : 16 Nov, 2022

Similar Reads

1. Resolving Conflicts During Multiple Inheritance in Java
2. How to Implement Multiple Inheritance by Using Interfaces in Java?
3. Multiple Inheritance in C++

Interfaces and Inheritance in Java

Prerequisites: Interfaces in Java, Java, and Multiple Inheritance A class can extend another class and/ can implement one and more than one interface.



Example:

Java

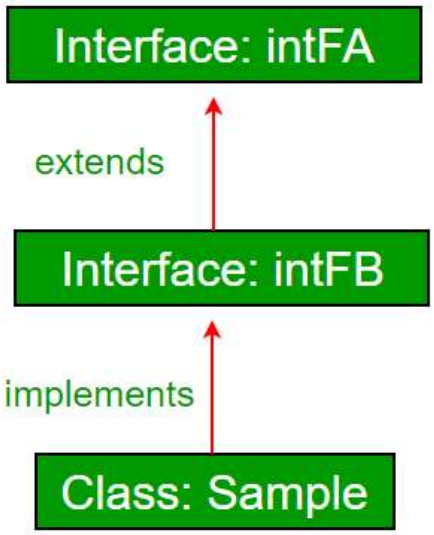
```
// Java program to demonstrate that a class can
// implement multiple interfaces
import java.io.*;
interface intfA
{
    void m1();
}

interface intfB
{
    void m2();
}

// class implements both interfaces
// and provides implementation to the method.
class sample implements intfA, intfB
{
```

```
    @Override
    public void m1()
    {
        System.out.println("Welcome: inside the method m1");
    }

    @Override
    public void m2()
    {
        System.out.println("Welcome: inside the method m2");
    }
}

class GFG
{
    public static void main (String[] args)
    {
        sample ob1 = new sample();

        // calling the method implemented
        // within the class.
        ob1.m1();
        ob1.m2();
    }
}
```

Output;

```
Welcome: inside the method m1
Welcome: inside the method m2
```

Interface inheritance : An Interface can extend other interface.



Inheritance is inheriting the properties of parent class into child class.
 Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object.
 The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class.
 You can also add new methods and fields in your current class.
 Inheritance represents the IS_A relationship which is also known as parent- child relationship.

Eg:
Dog IS_A Animal
Car IS_A Vehicle
Employee IS_A Person
Surgeon IS_A Doctor etc.

Java

```
class Animal
{
    public void eat()
    {
    }
}

class Dog extends Animal
{
    public static void main(String args[])
    {
        Dog d=new Dog();
```

```
        d.eat();
    }
}
```

Syntax of Java Inheritance

```
class <Subclass-name> extends <Superclass-name>
{
    //methods and fields
}
```

Note: The extends keyword indicates that you are making a new class that derives from an existing class. The meaning of “extends” is to increase the functionality.

Eg_1:

Java

```
import java.io.*;
class Person {
    int id;
    String name;
    void set_Person()
    {

        try {
            BufferedReader br = new BufferedReader(
                new InputStreamReader(System.in));
            System.out.println("Enter the Id:");
            id = Integer.parseInt(br.readLine());
            System.out.println("Enter the Name");
            name = br.readLine();
        }
        catch (Exception ex) {
            ex.printStackTrace();
        }
    }
    void disp_Person()
    {
        System.out.print(id + "\t" + name + "\t");
    }
}
class Employee extends Person {
    int sal;
    String desgn;
    void set_Emp()
    {
        try {
            set_Person();
            BufferedReader br = new BufferedReader(
                new InputStreamReader(System.in));
```

```
            System.out.println("Enter the Designation:");
            desgn = br.readLine();
            System.out.println("Enter the Salary:");
            sal = Integer.parseInt(br.readLine());
        }
        catch (Exception ex) {
            ex.printStackTrace();
        }
    }
    void disp_Emp()
    {
        disp_Person();
        System.out.println(desgn + "\t" + sal);
    }
    public static void main(String args[])
    {
        Employee e1 = new Employee();
        e1.set_Emp();
        e1.disp_Emp();
    }
}
```

Eg_2:

Java

```
class Person1 {
    int id;
    String name;
    void set_Person(int id, String name)
    {
        try {
            this.id = id;
            this.name = name;
        }
        catch (Exception ex) {
            ex.printStackTrace();
        }
    }
    void disp_Person()
    {
        System.out.print(id + "\t" + name + "\t");
    }
}
class Employee1 extends Person1 {
    int sal;
    String desgn;
    void set_Emp(int id, String name, String desgn, int sal)
    {
        try {
            set_Person(id, name);
            this.desgn = desgn;
            this.sal = sal;
        }
```

```
        catch (Exception ex) {
            ex.printStackTrace();
        }
    }
    void disp_Emp()
    {
        disp_Person();
        System.out.print(desgn + "\t" + sal);
    }
    public static void main(String args[])
    {

        Employee1 e1 = new Employee1();
        e1.set_Emp(1001, "Manjeet", "AP", 20000);
        e1.disp_Emp();
    }
}
```

Types of inheritance in java

 Java Support three types of inheritance in java: single level, multilevel and hierarchical inheritance in case of classes to avoid ambiguity.
 In java programming, multiple and hybrid inheritance is supported through interface only.

Single Inheritance Example

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

Java

```
class A {
    int a;
    void set_A(int x) {
      a = x;
    }
}
class B extends A {
```

```
    int b, product;
    void set_B(int x) {
      b = x;
    }
    void cal_Product()
    {
        product = a * b;
        System.out.println("Product =" + product);
    }
    public static void main(String[] args)
    {
        B b = new B();
        b.set_A(5);
        b.set_B(5);
        b.cal_Product();
    }
}
```

Multilevel Inheritance Example

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

Java

```
class A {
    int a;
    void set_A(int x) {
      a = x;
    }
}
class B extends A {
    int b;
    void set_B(int x) {
      b = x;
    }
}
class C extends B {
    int c, product;
    void cal_Product()
    {
        product = a * b;
        System.out.println("Product =" + product);
    }
    public static void main(String[] args)
    {
        C c = new C();
        c.set_A(5);
        c.set_B(5);
        c.cal_Product();
    }
```

```
}
```

Hierarchical Inheritance Example

When two or more classes inherit a single class, it is known as hierarchical inheritance.

Eg:

Java

```
class A {
    int a;
    void set_A(int x) {
      a = x;
    }
}
class B extends A {
    int b;
    void set_B(int x) {
      b = x;
    }
}
class C extends A {
    int c;
    void set_C(int x) {
      c = x;
    }
```

Java

```
// Java program to demonstrate inheritance in
// interfaces.
import java.io.*;
interface intfA {
    void geekName();
}

interface intfB extends intfA {
    void geekInstitute();
}

// class implements both interfaces and provides
// implementation to the method.
class sample implements intfB {
    @Override public void geekName()
    {
        System.out.println("Rohit");
    }
```

```
    @Override public void geekInstitute()
    {
        System.out.println("JIIT");
    }

    public static void main(String[] args)
    {
        sample ob1 = new sample();

        // calling the method implemented
        // within the class.
        ob1.geekName();
        ob1.geekInstitute();
    }
}
```

Output:

```
Rohit
JIIT
```

An interface can also extend multiple interfaces.

Java

```
// Java program to demonstrate multiple inheritance
// in interfaces

import java.io.*;

interface intfA {
    void geekName();
}

interface intfB {
    void geekInstitute();
}

interface intfC extends intfA, intfB {
    void geekBranch();
}

// class implements both interfaces and provides
// implementation to the method.
class sample implements intfC {
    public void geekName() { System.out.println("Rohit"); }

    public void geekInstitute()
    {
        System.out.println("JIIT");
    }
```

```
    public void geekBranch() { System.out.println("CSE"); }

    public static void main(String[] args)
    {
        sample ob1 = new sample();

        // calling the method implemented
        // within the class.
        ob1.geekName();
        ob1.geekInstitute();
        ob1.geekBranch();
    }
}
```

Output

```
Rohit
JIIT
CSE
```

Why Multiple Inheritance is not supported through a class in Java, but it can be possible through the interface? Multiple Inheritance is not supported by class because of ambiguity. In the case of interface, there is no ambiguity because the implementation of the method(s) is provided by the implementing class up to Java 7. From Java 8, interfaces also have implementations of methods. So if a class implements two or more interfaces having the same method signature with implementation, it is mandated to implement the method in class also. Refer to Java and Multiple Inheritance for details.

This article is contributed by **Nitsdheerendra**. If you like GeeksforGeeks and would like to contribute, you can also write an article using write.geeksforgeeks.org or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

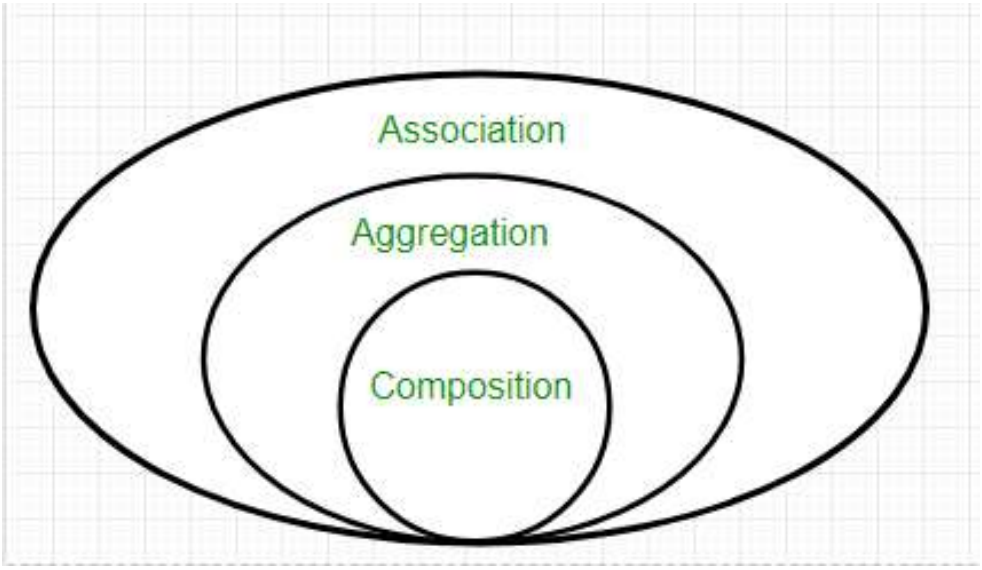
Last Updated : 17 Jan, 2023

Similar Reads

1. How to Implement Multiple Inheritance by Using Interfaces in Java?
2. Generic Constructors and Interfaces in Java

Association, Composition and Aggregation in Java

Association is a relation between two separate classes which establishes through their Objects. Association can be one-to-one, one-to-many, many-to-one, many-to-many. In Object-Oriented programming, an Object communicates to another object to use functionality and services provided by that object. **Composition** and **Aggregation** are the two forms of association.



Example:

Java

```
// Java Program to illustrate the
// Concept of Association

// Importing required classes
import java.io.*;
import java.util.*;

// Class 1
// Bank class
class Bank {

    // Attributes of bank
```

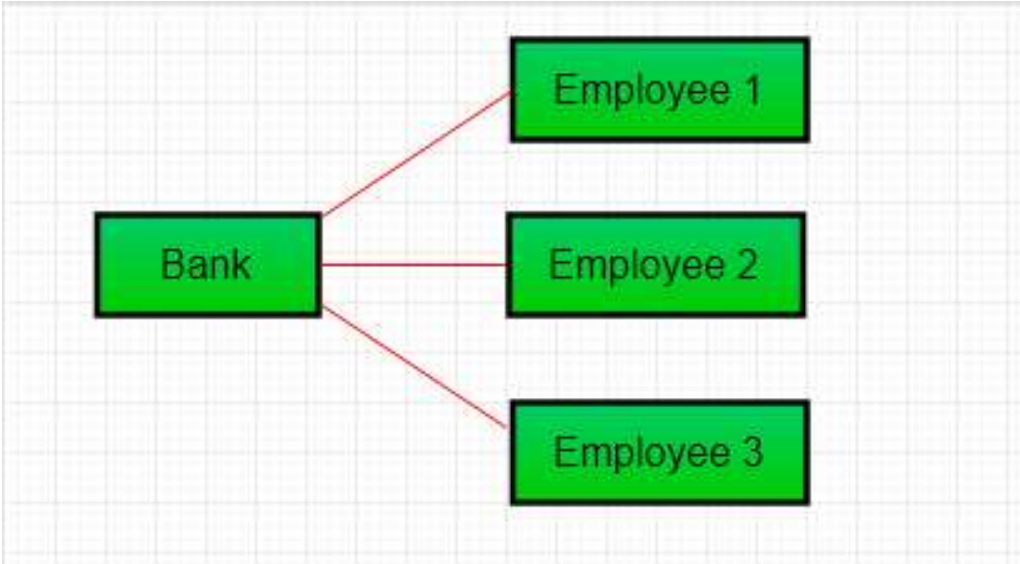
```
    private String name;

    private Set<Employee> employees;
    // Constructor of this class
    Bank(String name)
    {
        // this keyword refers to current instance itself
        this.name = name;
    }

    // Method of Bank class
    public String getBankName()
    {
        // Returning name of bank
        return this.name;
    }

    public void setEmployees(Set<Employee> employees)
    {
        this.employees = employees;
    }
    public Set<Employee>
    getEmployees(Set<Employee> employees)
    {
        return this.employees;
    }
}

// Class 2
// Employee class
class Employee {
    // Attributes of employee
    private String name;
    // Employee name
    Employee(String name)
    {
        // This keyword refers to current instance itself
        this.name = name;
    }

    // Method of Employee class
    public String getEmployeeName()
    {
        // returning the name of employee
        return this.name;
    }
}

// Class 3
// Association between both the
// classes in main method
class GFG {

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

```
        // Creating objects of bank and Employee class
        Bank bank = new Bank("ICICI");
        Employee emp = new Employee("Ridhi");

        Set<Employee> employees = new HashSet<>();
        employees.add(emp);

        bank.setEmployees(employees);

        System.out.println(emp.getEmployeeName()
                           + " belongs to bank "
                           + bank.getBankName());
    }
}
```

Output

```
Ridhi belongs to bank ICICI
```

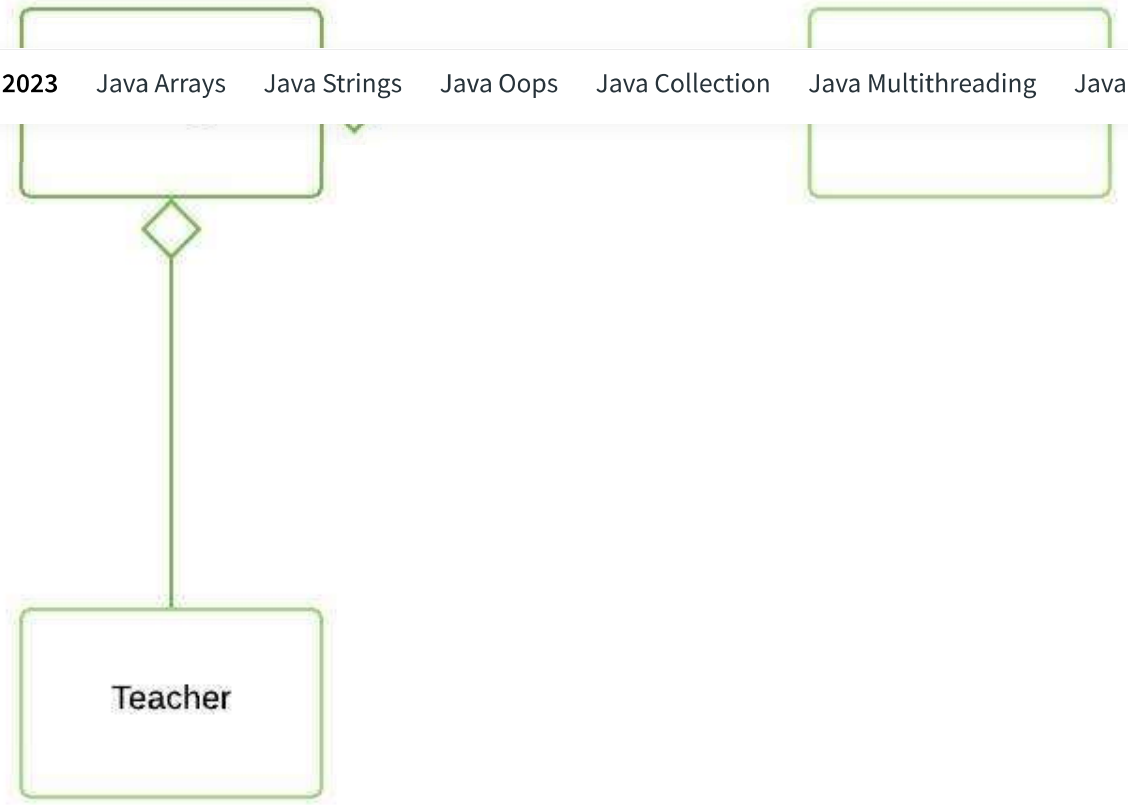
Output Explanation: In the above example, two separate classes Bank and Employee are associated through their Objects. Bank can have many employees, So it is a one-to-many relationship.



Aggregation

It is a special form of Association where:

- It represents Has-A's relationship.
- It is a **unidirectional association** i.e. a one-way relationship. For example, a department can have students but vice versa is not possible and thus unidirectional in nature.
- In Aggregation, **both entries can survive individually** which means ending one entity will not affect the other entity.



Aggregation

Example

Java

```
// Java program to illustrate Concept of Aggregation

// Importing required classes
import java.io.*;
import java.util.*;

// Class 1
// Student class
class Student {

    // Attributes of student
    String name;
    int id;
    String dept;

    // Constructor of student class
    Student(String name, int id, String dept)
    {

        // This keyword refers to current instance itself
        this.name = name;
        this.id = id;
        this.dept = dept;
```

```
    }
}

// Class 2
// Department class contains list of student objects
// It is associated with student class through its Objects
class Department {
    // Attributes of Department class
    String name;
    private List<Student> students;
    Department(String name, List<Student> students)
    {
        // this keyword refers to current instance itself
        this.name = name;
        this.students = students;
    }

    // Method of Department class
    public List<Student> getStudents()
    {
        // Returning list of user defined type
        // Student type
        return students;
    }
}

// Class 3
//  Institute class contains list of Department
// Objects. It is associated with Department
// class through its Objects
class Institute {

    // Attributes of Institute
    String instituteName;
    private List<Department> departments;

    // Constructor of institute class
    Institute(String instituteName,List<Department> departments)
    {
        // This keyword refers to current instance itself
        this.instituteName = instituteName;
        this.departments = departments;
    }

    // Method of Institute class
    // Counting total students of all departments
    // in a given institute
    public int getTotalStudentsInInstitute()
    {
        int noOfStudents = 0;
        List<Student> students;

        for (Department dept : departments) {
            students = dept.getStudents();

            for (Student s : students) {
                noOfStudents++;
```

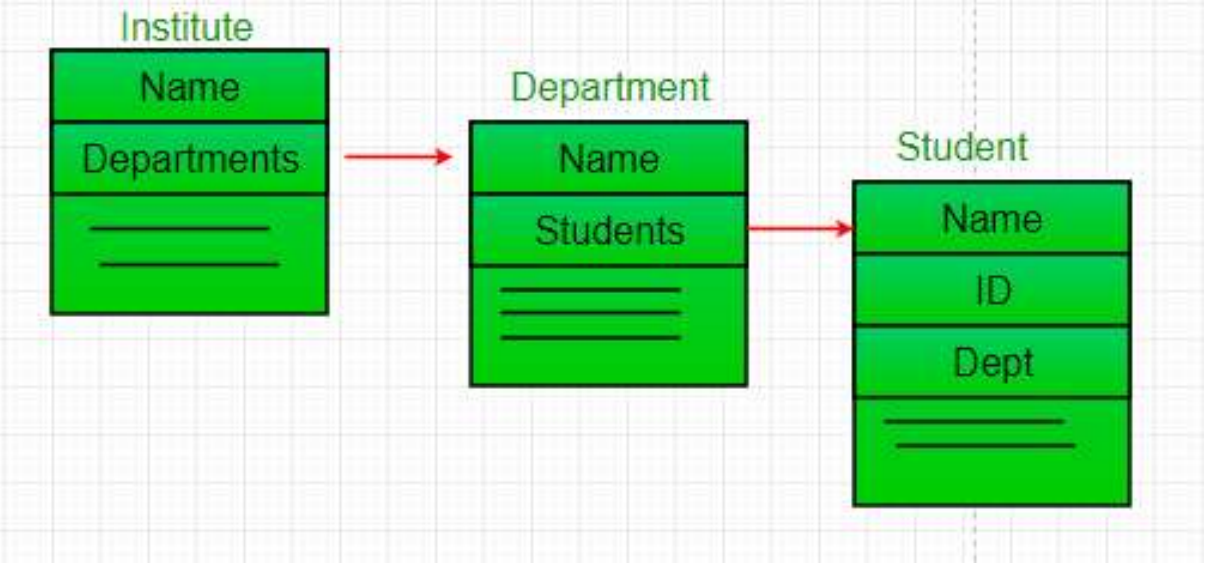
```
            }
        }

        return noOfStudents;
    }
}

// Class 4
// main class
class GFG {

    // main driver method
    public static void main(String[] args)
    {
        // Creating object of Student class inside main()
        Student s1 = new Student("Mia", 1, "CSE");
        Student s2 = new Student("Priya", 2, "CSE");
        Student s3 = new Student("John", 1, "EE");
        Student s4 = new Student("Rahul", 2, "EE");

        // Creating a List of CSE Students
        List<Student> cse_students = new ArrayList<Student>();

        // Adding CSE students
        cse_students.add(s1);
        cse_students.add(s2);

        // Creating a List of EE Students
        List<Student> ee_students
            = new ArrayList<Student>();

        // Adding EE students
        ee_students.add(s3);
        ee_students.add(s4);

        // Creating objects of EE and CSE class inside
        // main()
        Department CSE = new Department("CSE", cse_students);
        Department EE = new Department("EE", ee_students);

        List<Department> departments = new ArrayList<Department>();
        departments.add(CSE);
        departments.add(EE);

        // Lastly creating an instance of Institute
        Institute institute = new Institute("BITS", departments);

        // Display message for better readability
        System.out.print("Total students in institute: ");

        // Calling method to get total number of students
        // in institute and printing on console
        System.out.print(institute.getTotalStudentsInInstitute());
    }
}
```

Output

```
Total students in institute: 4
```

Output Explanation: In this example, there is an Institute which has no. of departments like CSE, EE. Every department has no. of students. So, we make an Institute class that has a reference to Object or no. of Objects (i.e. List of Objects) of the Department class. That means Institute class is associated with Department class through its Object(s). And Department class has also a reference to Object or Objects (i.e. List of Objects) of the Student class means it is associated with the Student class through its Object(s).

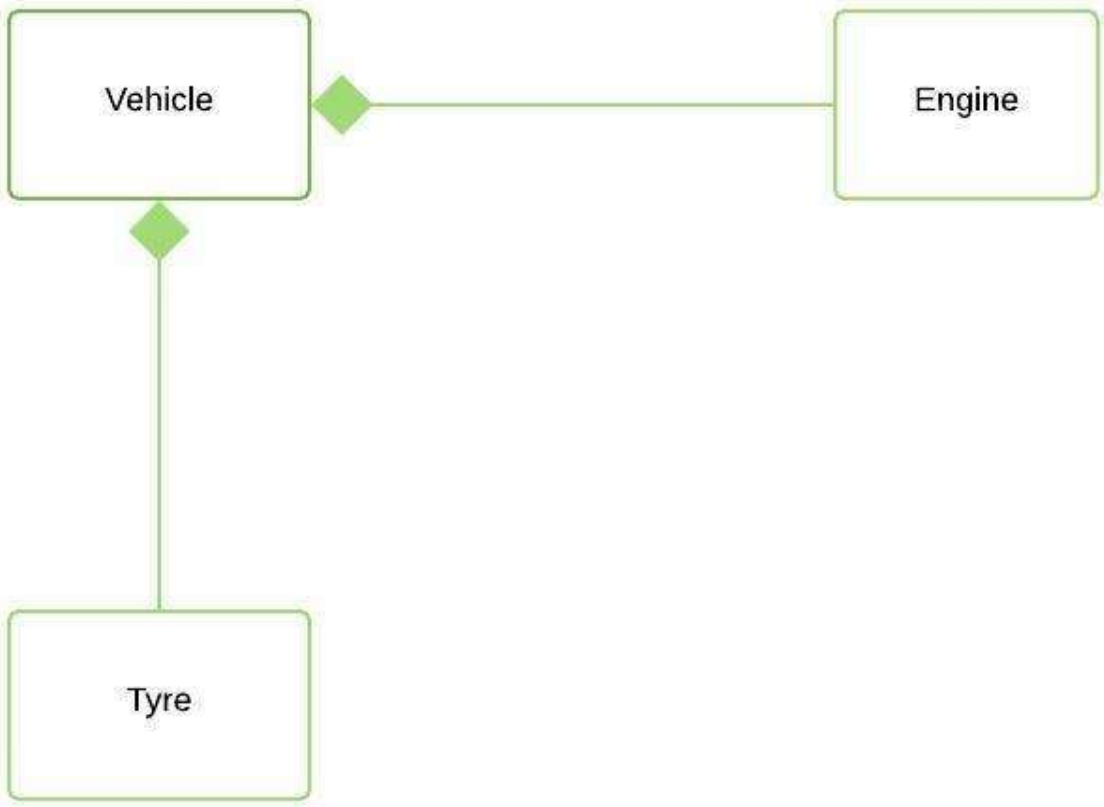
It represents a **Has-A** relationship. In the above example: Student **Has-A** name. Student **Has-A** ID. Student **Has-A** Dept. Department **Has-A** Students as depicted from the below media.



When do we use Aggregation ??

Code reuse is best achieved by aggregation.

Concept 3: **Composition**



Composition

Composition is a restricted form of Aggregation in which two entities are highly dependent on each other.

- It represents **part-of** relationship.
- In composition, both entities are dependent on each other.
- When there is a composition between two entities, the composed object **cannot exist** without the other entity.

Example Library

Java

```
// Java program to illustrate
// the concept of Composition

// Importing required classes
import java.io.*;
import java.util.*;

// Class 1
//  Book
class Book {

    // Attributes of book
    public String title;
```

```
    public String author;

    // Constructor of Book class
    Book(String title, String author)
    {

        // This keyword refers to current instance itself
        this.title = title;
        this.author = author;
    }
}

// Class 2
class Library {

    // Reference to refer to list of books
    private final List<Book> books;

    // Library class contains list of books
    Library(List<Book> books)
    {

        // Referring to same book as
        // this keyword refers to same instance itself
        this.books = books;
    }

    // Method
    // To get total number of books in library
    public List<Book> getTotalBooksInLibrary()
    {

        return books;
    }
}

// Class 3
// Main class
class GFG {

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

        // Creating objects of Book class inside main()
        // method Custom inputs
        Book b1
            = new Book("EffectiveJ Java", "Joshua Bloch");
        Book b2
            = new Book("Thinking in Java", "Bruce Eckel");
        Book b3 = new Book("Java: The Complete Reference",
                           "Herbert Schildt");

        // Creating the list which contains number of books
        List<Book> books = new ArrayList<Book>();

        // Adding books
```

```
        // using add() method
        books.add(b1);
        books.add(b2);
        books.add(b3);

        Library library = new Library(books);

        // Calling method to get total books in library
        // and storing it in list of user0defined type -
        // Books
        List<Book> bks = library.getTotalBooksInLibrary();

        // Iterating over books using for each loop
        for (Book bk : bks) {

            // Printing the title and author name of book on
            // console
            System.out.println("Title : " + bk.title
                               + " and "
                               + " Author : " + bk.author);
        }
    }
}
```

Output

```
Title : EffectiveJ Java and  Author : Joshua Bloch
Title : Thinking in Java and  Author : Bruce Eckel
Title : Java: The Complete Reference and  Author : Herbert Schildt
```

Output explanation: In the above example, a library can have no. of **books** on the same or different subjects. So, If Library gets destroyed then All books within that particular library will be destroyed. i.e. books can not exist without libraries. That's why it is composition. Book is **Part-of** Library.

Aggregation vs Composition

1. Dependency: Aggregation implies a relationship where the child **can exist independently** of the parent. For example, Bank and Employee, delete the Bank and the Employee still exist. whereas Composition implies a relationship where the child **cannot exist independent** of the parent. Example: Human and heart, heart don't exist separate to a Human

2. Type of Relationship: Aggregation relation is **"has-a"** and composition is **"part-of"** relation.

3. Type of association: Composition is a **strong** Association whereas Aggregation is a **weak** Association.

Example:

Java

```
// Java Program to Illustrate Difference between
// Aggregation and Composition

// Importing I/O classes
import java.io.*;

// Class 1
// Engine class which will
// be used by car. so 'Car'
// class will have a field
// of Engine type.
class Engine {

    // Method to starting an engine
    public void work()
    {

        // Print statement whenever this method is called
        System.out.println(
            "Engine of car has been started ");
    }
}

// Class 2
// Engine class
final class Car {

    // For a car to move,
    // it needs to have an engine.

    // Composition
    private final Engine engine;

    // Note: Uncommented part refers to Aggregation
    // private Engine engine;

    // Constructor of this class
    Car(Engine engine)
    {

        // This keywords refers to same instance
        this.engine = engine;
    }

    // Method
    // Car start moving by starting engine
    public void move()
    {

        // if(engine != null)
        {
```

```
            // Calling method for working of engine
            engine.work();

            // Print statement
            System.out.println("Car is moving ");
        }
    }
}

// Class 3
// Main class
class GFG {

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

        // Making an engine by creating
        // an instance of Engine class.
        Engine engine = new Engine();

        // Making a car with engine so we are
        // passing a engine instance as an argument
        // while creating instance of Car
        Car car = new Car(engine);

        // Making car to move by calling
        // move() method inside main()
        car.move();
    }
}
```

Output

```
Engine of car has been started
Car is moving
```

In case of aggregation, the Car also performs its functions through an Engine. but the Engine is not always an internal part of the Car. An engine can be swapped out or even can be removed from the car. That's why we make The Engine type field non-final.

This article is contributed by **Nitsdheerendra**. If you like GeeksforGeeks and would like to contribute, you can also write an article using write.geeksforgeeks.org or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Comparison of Inheritance in C++ and Java

The purpose of inheritance is the same in C++ and Java. Inheritance is used in both languages for reusing code and/or creating an 'is-a' relationship. The following examples will demonstrate the differences between Java and C++ that provide support for inheritance.

1) In Java, all classes inherit from the Object class directly or indirectly. Therefore, there is always a single inheritance tree of classes in Java, and the Object Class is the root of the tree. In Java, when creating a class it automatically inherits from the Object Class. In C++ however, there is a forest of classes; when we create a class that doesn't inherit from another, we create a new tree in a forest.

Following the Java, example shows that the **Test class** automatically inherits from the Object class.

Java

```
class Test {
    // members of test
}
class Main {
    public static void main(String[] args)
    {
        Test t = new Test();
        System.out.println("t is instanceof Object: "
                           + (t instanceof Object));
    }
}
```

Output

```
t is instanceof Object: true
```

2) In Java, members of the grandparent class are not directly accessible. (Refer to **this** article for more details).

3) The meaning of protected member access specifier is somewhat different in Java. 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).

For example, in the following program, protected members of A are accessible in B.

Java

```
class A {
    protected int x = 10, y = 20;
}

class B {
    public static void main(String args[])
    {
        A a = new A();
        System.out.println(a.x + " " + a.y);
    }
}
```

Output

```
10 20
```

4) Java uses 'extends' keywords for inheritance. Unlike C++, Java doesn't provide an inheritance specifier like public, protected, or private. Therefore, we cannot change the protection level of members of the base class in Java, if some data member is public or

protected in the base class then it remains public or protected in the derived class. Like C++, private members of a base class are not accessible in a derived class.

Unlike C++, in Java, we don't have to remember those rules of inheritance which are a combination of base class access specifier and inheritance specifier.

5) In Java, methods are virtual by default. In C++, we explicitly use virtual keywords (Refer to **this** article for more details).
6) Java uses a separate keyword *interface* for interfaces and *abstract* keywords for abstract classes and abstract functions.

Following is a Java abstract class example,

Java

```
// An abstract class example
abstract class myAbstractClass {

    // An abstract method
    abstract void myAbstractFun();

    // A normal method
    void fun() { System.out.println("Inside My fun"); }
}

public class myClass extends myAbstractClass {
    public void myAbstractFun()
    {
        System.out.println("Inside My fun");
    }
}
```

Following is a Java interface example,

Java

```
// An interface example
public interface myInterface {

    // myAbstractFun() is public
     // and abstract, even if we
    // don't use these keywords
    void myAbstractFun();
    // is same as public abstract void myAbstractFun()
}

// Note the implements keyword also.
public class myClass implements myInterface {
```

```
    public void myAbstractFun()
    {
        System.out.println("Inside My fun");
    }
}
```

7) Unlike C++, Java doesn't support multiple inheritances. A class cannot inherit from more than one class. However, A class can implement multiple interfaces.
8) In C++, the default constructor of the parent class is automatically called, but if we want to call a parameterized constructor of a parent class, we must use the Initializer list. 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. See the following Java example.

Java

```
class Base {
    private int b;
    Base(int x)
    {
        b = x;
        System.out.println("Base constructor called");
    }
}

class Derived extends Base {
    private int d;
    Derived(int x, int y)
    {
        // Calling parent class parameterized constructor
        // Call to parent constructor must be the first line
        // in a Derived class
        super(x);
        d = y;
        System.out.println("Derived constructor called");
    }
}

class Main {
    public static void main(String[] args)
    {
        Derived obj = new Derived(1, 2);
    }
}
```

Output

```
Base constructor called
```

```
Derived constructor called
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Last Updated : 03 Apr, 2023

75

Similar Reads

1. Comparison of Exception Handling in C++ and Java
2. Comparison of boolean data type in C++ and Java
3. Comparison of yield(), join() and sleep() in Java
4. Java 11 - Features and Comparison
5. Comparison of double and float primitive types in Java
6. Comparison of static keyword in C++ and Java
7. Inheritance and Constructors in Java
8. Java and Multiple Inheritance
9. Difference between Inheritance and Composition in Java
10. Difference between Inheritance and Interface in Java

Previous Next

Article Contributed By :

GeeksforGeeks

Vote for difficulty

Current difficulty : Easy

Easy | Normal | Medium | Hard | Expert

Using final with Inheritance in Java

Prerequisite – Overriding in java, Inheritance

final is a keyword in java used for restricting some functionalities. We can declare variables, methods, and classes with the final keyword.

Using final with inheritance

During inheritance, we must declare methods with the final keyword for which we are required to follow the same implementation throughout all the derived classes. Note that it is not necessary to declare final methods in the initial stage of inheritance(base class always). We can declare a final method in any subclass for which we want that if any other class extends this subclass, then it must follow the same implementation of the method as in that subclass.

Java

```java
// Java program to illustrate
// use of final with inheritance

// base class
abstract class Shape
{
    private double width;

    private double height;

    // Shape class parameterized constructor
    public Shape(double width, double height)
    {
        this.width = width;
        this.height = height;
    }

    // getWidth method is declared as final
    // so any class extending
    // Shape can't override it
```

```
    public final double getWidth()
    {
        return width;
    }

    // getHeight method is declared as final
    // so any class extending Shape
    // can not override it
    public final double getHeight()
    {
        return height;
    }


    // method getArea() declared abstract because
    // it upon its subclasses to provide
    // complete implementation
    abstract double getArea();
}

// derived class one
class Rectangle extends Shape
{
    // Rectangle class parameterized constructor
    public Rectangle(double width, double height)
    {
        // calling Shape class constructor
        super(width, height);
    }

    // getArea method is overridden and declared
    // as final    so any class extending
    // Rectangle can't override it
    @Override
    final double getArea()
    {
        return this.getHeight() * this.getWidth();
    }

}

//derived class two
class Square extends Shape
{
    // Square class parameterized constructor
    public Square(double side)
    {
        // calling Shape class constructor
        super(side, side);
    }

    // getArea method is overridden and declared as
    // final so any class extending
    // Square can't override it
    @Override
    final double getArea()
    {
```

```
        return this.getHeight() * this.getWidth();
    }

}

// Driver class
public class Test
{
    public static void main(String[] args)
    {
        // creating Rectangle object
        Shape s1 = new Rectangle(10, 20);

        // creating Square object
        Shape s2 = new Square(10);

        // getting width and height of s1
        System.out.println("width of s1 : "+ s1.getWidth());
        System.out.println("height of s1 : "+ s1.getHeight());

        // getting width and height of s2
        System.out.println("width of s2 : "+ s2.getWidth());
        System.out.println("height of s2 : "+ s2.getHeight());

        //getting area of s1
        System.out.println("area of s1 : "+ s1.getArea());

        //getting area of s2
        System.out.println("area of s2 : "+ s2.getArea());

    }
}
```

Output:

```
width of s1 : 10.0
height of s1 : 20.0
```

```
width of s2 : 10.0
height of s2 : 10.0
area of s1 : 200.0
area of s2 : 100.0
```

Using final to Prevent Inheritance

When a class is declared as final then it cannot be subclassed i.e. no other class can extend it. This is particularly useful, for example, when creating an immutable class like the predefined String class. The following fragment illustrates the **final** keyword with a class:

```
final class A
{
     // methods and fields
}
// The following class is illegal.
class B extends A
{
    // ERROR! Can't subclass A
}
```

Note :

- Declaring a class as final implicitly declares all of its methods as final, too.
- It is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations. For more on abstract classes, refer abstract classes in java

Using final to Prevent Overriding

When a method is declared as final then it cannot be overridden by subclasses. The Object class does this—a number of its methods are final. The following fragment illustrates the **final** keyword with a method:

```
class A
{
```

```
    final void m1()
    {
        System.out.println("This is a final method.");
    }
}

class B extends A
{
    void m1()
    {
        // ERROR! Can't override.
        System.out.println("Illegal!");
    }
}
```

Normally, Java resolves calls to methods dynamically, at run time. This is called **late or dynamic binding**. However, since final methods cannot be overridden, a call to one can be resolved at compile time. This is called **early or static binding**.

This article is contributed by **Gaurav Miglani**. If you like GeeksforGeeks and would like to contribute, you can also write an article using write.geeksforgeeks.org or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Last Updated : 16 Mar, 2022

37

Similar Reads

1. Unreachable statement using final and non-final variable in Java
2. Java Program to Calculate Interest For FDs, RDs using Inheritance
3. How to Implement Multiple Inheritance by Using Interfaces in Java?
4. Assigning values to static final variables in Java
5. Private and final methods in Java

Object Serialization with Inheritance in Java

Prerequisite: Serialization, Inheritance

Serialization is a mechanism of converting the state of an object into a byte stream. The byte array can be the class, version, and internal state of the object.

Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.

There are some cases of Serialization with respect to inheritance:

Case 1: If the superclass is serializable, then subclass is automatically serializable
If the superclass is Serializable, then by default, every subclass is serializable. Hence, even though subclass doesn't implement Serializable interface(and if its superclass implements Serializable), then we can serialize subclass object.

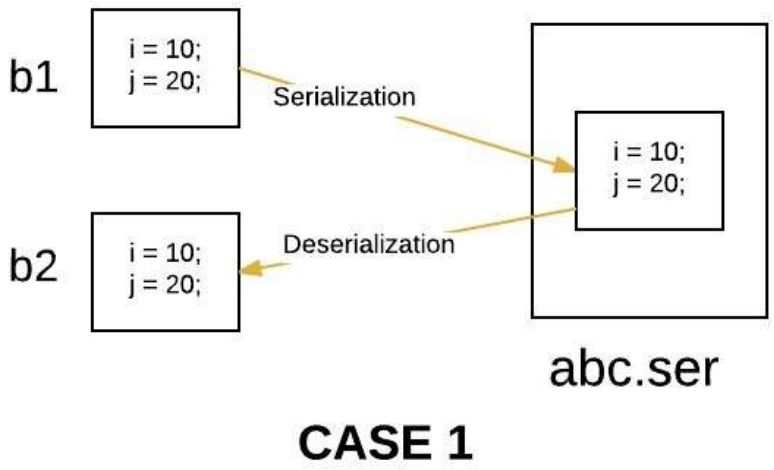

```
// Java program to demonstrate
// that if superclass is
// serializable then subclass
// is automatically serializable

import java.io.FileInputStream;
```

```
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

// superclass A
// implementing Serializable interface
class A implements Serializable
{
    int i;

    // parameterized constructor
    public A(int i)
    {
        this.i = i;
    }

}

// subclass B
// B class doesn't implement Serializable
// interface.
class B extends A
{
    int j;

    // parameterized constructor
    public B(int i, int j)
    {
        super(i);
        this.j = j;
    }
}

// Driver class
public class Test
{
    public static void main(String[] args)
            throws Exception
    {
        B b1 = new B(10,20);

        System.out.println("i = " + b1.i);
        System.out.println("j = " + b1.j);

        /* Serializing B's(subclass) object */

        //Saving of object in a file
        FileOutputStream fos = new FileOutputStream("abc.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);

        // Method for serialization of B's class object
        oos.writeObject(b1);

        // closing streams
        oos.close();
        fos.close();
```

```
        System.out.println("Object has been serialized");

        /* De-Serializing B's(subclass) object */

        // Reading the object from a file
        FileInputStream fis = new FileInputStream("abc.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);

        // Method for de-serialization of B's class object
        B b2 = (B)ois.readObject();

        // closing streams
        ois.close();
        fis.close();

        System.out.println("Object has been deserialized");

        System.out.println("i = " + b2.i);
        System.out.println("j = " + b2.j);
    }
}
```

Output:

```
i = 10
j = 20
Object has been serialized
Object has been deserialized
i = 10
j = 20
```



CASE 1

Case 2: If a superclass is not serializable, then subclass can still be serialized

Even though the superclass doesn't implement a Serializable interface, we can serialize subclass objects if the subclass itself implements a Serializable interface. So we can say

that to serialize subclass objects, superclass need not be serializable. But what happens with the instances of superclass during serialization in this case. The following procedure explains this.

Case 2(a): What happens when a class is serializable, but its superclass is not?
Serialization: At the time of serialization, if any instance variable inherits from the non-serializable superclass, then JVM ignores the original value of that instance variable and saves the default value to the file.

De- Serialization: At the time of de-serialization, if any non-serializable superclass is present, then JVM will execute instance control flow in the superclass. To execute instance control flow in a class, JVM will always invoke the default(no-arg) constructor of that class. So every non-serializable superclass must necessarily contain a default constructor. Otherwise, we will get a runtime exception.

Java

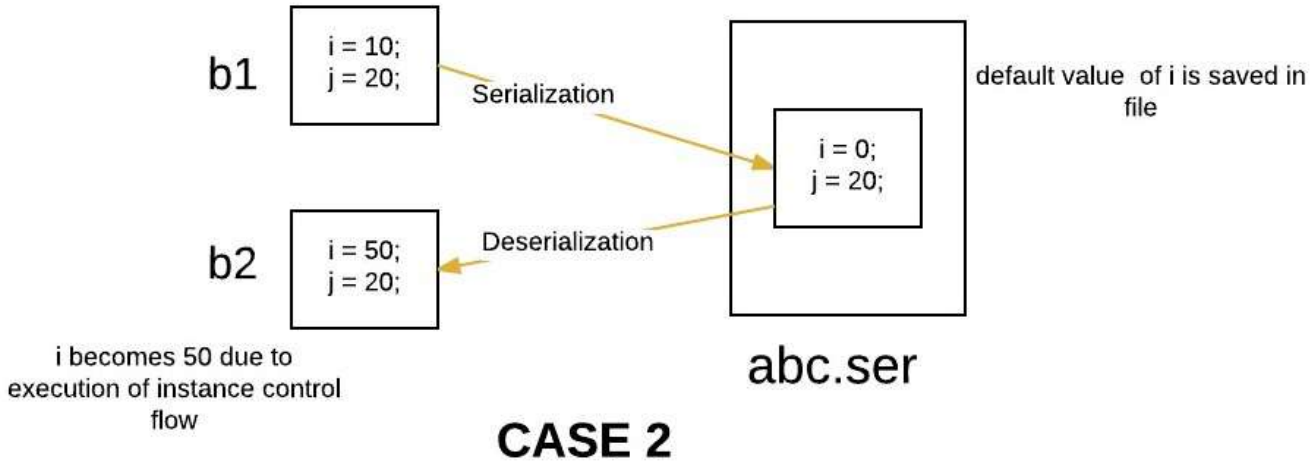
```
// Java program to demonstrate
// the case if superclass need
// not to be serializable
// while serializing subclass

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

// superclass A
// A class doesn't implement Serializable
// interface.
class A {
    int i;

    // parameterized constructor
    public A(int i){
      this.i = i;
    }

    // default constructor
    // this constructor must be present
    // otherwise we will get runtime exception
    public A()
    {
        i = 50;
        System.out.println("A's class constructor called");
    }
}

// subclass B
// implementing Serializable interface
```

```
class B extends A implements Serializable {
    int j;

    // parameterized constructor
    public B(int i, int j)
    {
        super(i);
        this.j = j;
    }
}

// Driver class
public class Test {
    public static void main(String[] args) throws Exception
    {
        B b1 = new B(10, 20);

        System.out.println("i = " + b1.i);
        System.out.println("j = " + b1.j);

        // Serializing B's(subclass) object

        // Saving of object in a file
        FileOutputStream fos
            = new FileOutputStream("abc.ser");
        ObjectOutputStream oos
            = new ObjectOutputStream(fos);

        // Method for serialization of B's class object
        oos.writeObject(b1);

        // closing streams
        oos.close();
        fos.close();

        System.out.println("Object has been serialized");

        // De-Serializing B's(subclass) object

        // Reading the object from a file
        FileInputStream fis
            = new FileInputStream("abc.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);

        // Method for de-serialization of B's class object
        B b2 = (B)ois.readObject();

        // closing streams
        ois.close();
        fis.close();

        System.out.println("Object has been deserialized");

        System.out.println("i = " + b2.i);
        System.out.println("j = " + b2.j);
    }
```

```
}
```

Output:

```
i = 10
j = 20
Object has been serialized
A's class constructor called
Object has been deserialized
i = 50
j = 20
```



Case 3: If the superclass is serializable, but we don't want the subclass to be serialized
There is no direct way to prevent sub-class from serialization in java. One possible way by which a programmer can achieve this is by implementing the *writeObject()* and *readObject()* methods in the subclass and needs to throw *NotSerializableException* from these methods. These methods are executed during serialization and de-serialization, respectively. By overriding these methods, we are just implementing our custom serialization.

Java

```
// Java program to demonstrate
// how to prevent
// subclass from serialization

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.NotSerializableException;
import java.io.ObjectInputStream;
```

```
import java.io.ObjectOutputStream;
import java.io.Serializable;

// superclass A
// implementing Serializable interface
class A implements Serializable
{
    int i;

    // parameterized constructor
    public A(int i)
    {
        this.i = i;
    }

}

// subclass B
// B class doesn't implement Serializable
// interface.
class B extends A
{
    int j;

    // parameterized constructor
    public B(int i,int j)
    {
        super(i);
        this.j = j;
    }

    // By implementing writeObject method,
    // we can prevent
    // subclass from serialization
    private void writeObject(ObjectOutputStream out) throws IOException
    {
        throw new NotSerializableException();
    }

    // By implementing readObject method,
    // we can prevent
    // subclass from de-serialization
    private void readObject(ObjectInputStream in) throws IOException
    {
        throw new NotSerializableException();
    }

}

// Driver class
public class Test
{
    public static void main(String[] args)
            throws Exception
    {
        B b1 = new B(10, 20);
```

```
        System.out.println("i = " + b1.i);
        System.out.println("j = " + b1.j);

        // Serializing B's(subclass) object

        //Saving of object in a file
        FileOutputStream fos = new FileOutputStream("abc.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);

        // Method for serialization of B's class object
        oos.writeObject(b1);

        // closing streams
        oos.close();
        fos.close();

        System.out.println("Object has been serialized");

        // De-Serializing B's(subclass) object

        // Reading the object from a file
        FileInputStream fis = new FileInputStream("abc.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);

        // Method for de-serialization of B's class object
        B b2 = (B)ois.readObject();

        // closing streams
        ois.close();
        fis.close();

        System.out.println("Object has been deserialized");

        System.out.println("i = " + b2.i);
        System.out.println("j = " + b2.j);
    }
}
```

Output:

```
i = 10
j = 20
Exception in thread "main" java.io.NotSerializableException
    at B.writeObject(Test.java:44)
```

This article is contributed by **Gaurav Miglani**. If you like GeeksforGeeks and would like to contribute, you can also write an article using write.geeksforgeeks.org or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Java – Exception Handling With Constructors in Inheritance

jainlovely450

Java provides a mechanism to handle exceptions. To learn about exception handling, you can refer to exceptions in java. In this article, we discuss exception handling with constructors when inheritance is involved. In Java, if the constructor of the parent class throws any checked exception, then the child class constructor can throw the same exception or its parent classes. There is no problem if the parent class or child class constructor throws any unchecked exceptions. The child class constructor can throw any unchecked exception without looking for a parent class constructor.

Understanding behavior of constructor calls

Whenever a method that throws some exception is called by another method, then the calling method is responsible for handling that exception (The calling method is the method that contains the actual call; the called method is the method being called). In case of constructors, the parent class constructor is called by the child class constructor. It means the child class constructor is responsible for handling the exception thrown by the parent class constructor.

Now, for handling an exception there are two ways, one is to catch the exception and another is to throw it. But in the case of the constructor, we can't handle it using the try-catch mechanism. The reason is that we enclose our code which can raise an exception in the try block and then catch it. The exception is raised due to a call to parent class constructor, like super(). It means if we want to handle the exception using try-catch is depicted in the below illustration.

Illustration 1

```
Child() {

    // Try- catch block
    try
    {
        super();
    }

    catch (FileNotFoundException exc)
    {
      // Handling exception(code)
    }
}
```

Actually, it is not correct as a call to super must be first statement in the child class constructor (refer super in java as it can be perceived from below illustration as follows:

Illustration 2

```
Child() {
   super(); // either called explicitly or added by the compiler in case of
default constructor
   try {
       // your code
      }
      catch(FileNotFoundException exc) {
       // handling code;
      }
  }
```

and hence the exception can't be caught (as its not inside the try block) and we can't handle it using try-catch mechanism. That's why we need to throw the exception. The below code will compile fine which will appear as follows:

```
// parent class constructor throws FileNotFoundException
Child() throws FileNotFoundException  {
  super(); // either called explicitly or added by the compiler in case of
default constructor
  try {
      // your code
     }
     catch(FileNotFoundException exc) {
      // handling code;
     }
 }
```

Different Use-cases:

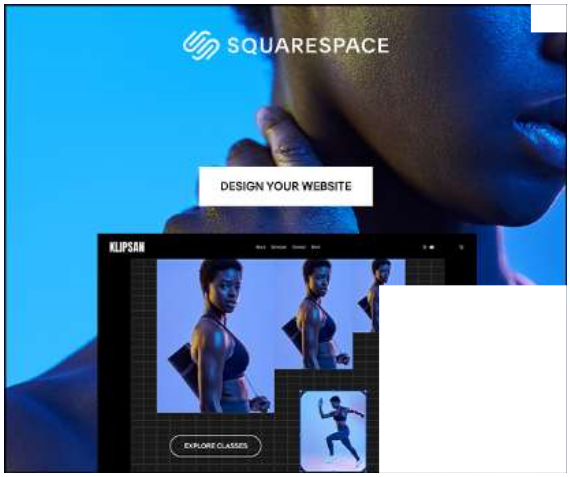
1. Parent class constructor does not throw any checked exception
2. Parent class constructor throws a checked exception

Now let us discuss each case in detail alongside justifying via clean java programs.

Case 1: Parent class constructor does not throw any checked exception

If the parent class constructor does not throw any exception then the child class can throw any exception or throw nothing.

Example 1



Java

```
// Java Program to Illustrate Exception handling with
// Constructors in inheritance where Parent class
```

```java
// constructor does not throw any checked exception

// Class 1
// Parent class
class Parent {

    // Constructor of Parent class
    // Not throwing any checked exception
    Parent()
    {

        // Print statement whenever parent class
        // constructor is called
        System.out.println("parent class constructor");
    }
}

// Class 2
// Child class
public class Child extends Parent {

    // Constructor of child class
    Child()
    {

        // Print statement whenever child class
        // constructor is called
        System.out.println("child class constructor");
    }

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

        // Creating object of child class inside main()
        Child child = new Child();
    }
}
```

Output

```
parent class constructor
child class constructor
```

Example 2

Java

```java
// Java Program to Illustrate Exception handling with
// Constructors in inheritance where Parent class
// constructor does not throw any checked exception

// Class 1
```

```
// Parent class
class Parent {

    // Constructor of parent class
    // Not throwing any checked exception
    Parent()
    {

        // Print statement when constructor of
        // parent class is called
        System.out.println("parent class constructor");
    }
}

// Class 2
// Child class
public class Child extends Parent {
    Child() throws Exception
    {

        // Print statement when constructor of
        // child class is called
        System.out.println(
            "child class constructor throwing Exception");
    }

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

        // Creating object of child class
        Child child = new Child();
    }
}
```

Output

```
parent class constructor
child class constructor throwing Exception
```

Case 2: Parent class constructor throws a checked exception

If the parent class constructor throws a checked exception, then the child class constructor can throw the same exception or its super-class exception. Now at this point, the child class constructors have to throw the exception.

Example

Java

```
// Java Program to Illustrate Exception handling with
// Constructors in inheritance where Child class constructor
```

```
// Not throwing exception of same type or its parent classes

// Importing I/O classes
import java.io.*;

// Class 1
// Parent class
class Parent {

    // Constructor of parent class
    // Throwing checked exception
    Parent() throws FileNotFoundException
    {

        // Print statement when
        // parent class constructor is called
        System.out.println(
            "parent class constructor throwing exception");
    }
}

// Class 2
// Child class
class Child extends Parent {

    // Constructor of child class
    Child()
    {
        // Print statement when
        // child class constructor is called
        System.out.println("child class constructor");
    }

    // Main driver method
    public static void main(String[] args) throws Exception
    {
        // Creating object of child class inside main()
        Child child = new Child();
    }
}
```

Output

```
error: unreported exception FileNotFoundException; must be caught or
declared to be thrown
    Child() {
            ^
```

In order to resolve the error we need to declare the exceptions to be thrown. These exception can be of same or parent class.

Example 1

Java

```java
// Java Program to Illustrate Exception handling with Constructors
// in Inheritance where we Resolve the Error we Need to
// Declare the Exceptions to be Thrown

// Importing I/O classes
import java.io.*;

// Parent class
class Parent {

    // throwing checked exception
    Parent() throws FileNotFoundException {

        System.out.println("parent class constructor throwing checked exception");
    }
}

public class Child extends Parent {

    Child() throws FileNotFoundException {

        System.out.println("child class constructor throwing same exception");
    }

    public static void main(String[] args) throws Exception {

        Child child = new Child();
    }
}
```

Output

```
parent class constructor throwing checked exception
child class constructor throwing same exception
```

Example 2

Java

```java
// Java Program to Illustrate Exception handling with
// Constructors in Inheritance where we Resolve the Error we
// Need to Declare the Exceptions to be Thrown

// Importing I/O classes

// Importing package
```

```
package package1;
// Importing required I/O classes
import java.io.*;

// Class 1
// Parent class
class Parent {

    // throwing checked exception
    Parent() throws FileNotFoundException
    {
        System.out.println(
            "parent class constructor throwing checked exception");
    }
}

// Class 2
// Child class
public class Child extends Parent {

    // It can also throw same exception or its parent
    // classes exceptions
    Child() throws IOException
    {

        System.out.println(
            "child class constructor throwing super-class exception");
    }

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

        // Creating object of child class
        // inside main() method
        Child child = new Child();
    }
}
```

Output

```
parent class constructor throwing checked exception
child class constructor throwing super-class exception
```


Similar Reads

1. Inheritance and Constructors in Java

Difference between Inheritance and Composition in Java

Pragya_Chaurasia

Inheritance:

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. In doing this, we can reuse the fields and methods of the existing class without having to write them ourself.

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.

Types of Inheritance are:

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

Example of Inheritance:

Java

```
class A {
    int a, b;
    public void add(int x, int y)
    {
        a = x;
        b = y;
        System.out.println(
            "addition of a + b is:"
            + (a + b));
    }
}

class B extends A {
    public void sum(int x, int y)
    {
        add(x, y);
    }

    // Driver Code
    public static void main(String[] args)
    {
        B b1 = new B();
        b1.sum(5, 6);
    }
}
```

Output:

```
addition of a+b is:11
```

Here, class B is the derived class which inherit the property(**add method**) of the base class A.

Composition:

The composition also provides code reusability but the difference here is we do not extend the class for this.

Example of Composition:

Let us take an example of the **Library**.

Java

```
// Java program to illustrate
// the concept of Composition

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

```java
// class book
class Book {

    public String title;
    public String author;

    Book(String title, String author)
    {

        this.title = title;
        this.author = author;
    }
}

// Library class contains
// list of books.
class Library {

    // reference to refer to the list of books.
    private final List<Book> books;

    Library(List<Book> books)
    {
        this.books = books;
    }

    public List<Book> getTotalBooksInLibrary()
    {

        return books;
    }
}

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

        // Creating the Objects of Book class.
        Book b1 = new Book(
            "EffectiveJ Java",
            "Joshua Bloch");
        Book b2 = new Book(
            "Thinking in Java",
            "Bruce Eckel");
        Book b3 = new Book(
            "Java: The Complete Reference",
            "Herbert Schildt");

        // Creating the list which contains the
        // no. of books.
        List<Book> books = new ArrayList<Book>();
        books.add(b1);
        books.add(b2);
        books.add(b3);
```

```
        Library library = new Library(books);

        List<Book> bks = library.getTotalBooksInLibrary();
        for (Book bk : bks) {

            System.out.println("Title : "
                               + bk.title + " and "
                               + " Author : "
        }
    }
}
```


Output:

```
Title : EffectiveJ Java and  Author : Joshua Bloch
Title : Thinking in Java and  Author : Bruce Eckel
Title : Java: The Complete Reference and  Author : Herbert Schildt
```

Difference between Inheritance and Composition:

S.NO	Inheritance	Composition
1.	In inheritance, we define the class which we are inheriting(super class) and most importantly it cannot be changed at runtime	Whereas in composition we only define a type which we want to use and which can hold its different implementation also it can change at runtime. Hence, Composition is much more flexible than Inheritance.
2.	Here we can only extend one class, in other words more than one class can't be extended as java do not support multiple inheritance.	Whereas composition allows to use functionality from different class.
3.	In inheritance we need parent class in order to test child class.	Composition allows to test the implementation of the classes we are using independent of parent or child class.
4.	Inheritance cannot extend final class.	Whereas composition allows code reuse even from final classes.

S.NO	Inheritance	Composition
5.	It is an **is-a** relationship.	While it is a **has-a** relationship.

Last Updated : 22 Jul, 2021

15

Similar Reads

1. Favoring Composition Over Inheritance In Java With Examples
2. Difference Between Aggregation and Composition in Java
3. Association, Composition and Aggregation in Java
4. Difference between Inheritance and Interface in Java
5. Composition in Java
6. Difference between Containership and Inheritance in C++
7. Inheritance and Constructors in Java
8. Java and Multiple Inheritance
9. Illustrate Class Loading and Static Blocks in Java Inheritance
10. Interfaces and Inheritance in Java

Next

Association, Composition and Aggregation in Java

Article Contributed By :

Pragya_Chaurasia
Pragya_Chaurasia

Difference between Inheritance and Interface in Java

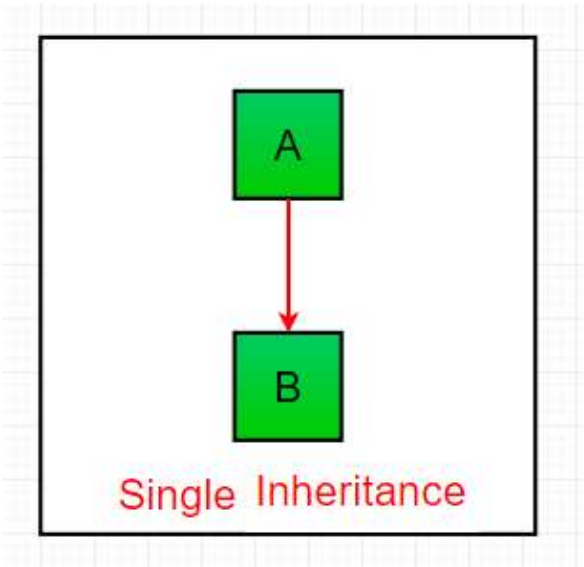
supriya_saxena

Java is one of the most popular and widely used programming languages. Java has been one of the most popular programming languages for many years. Java is Object Oriented. However, it is not considered as a pure object-oriented as it provides support for primitive data types (like int, char, etc). In this article, we will understand the difference between the two most important concepts in java, inheritance and interface.

Interface: Interfaces are the blueprints of the classes. They specify what a class must do and not how. Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (i.e.) they only contain method signature and not the body of the method. Interfaces are used to implement a complete abstraction.

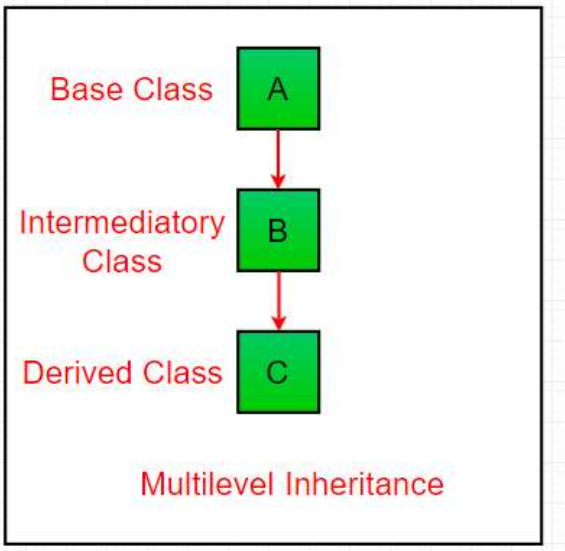
Inheritance: It is a mechanism in java by which one class is allowed to inherit the features of the another class. There are multiple inheritances possible in java. They are:

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.

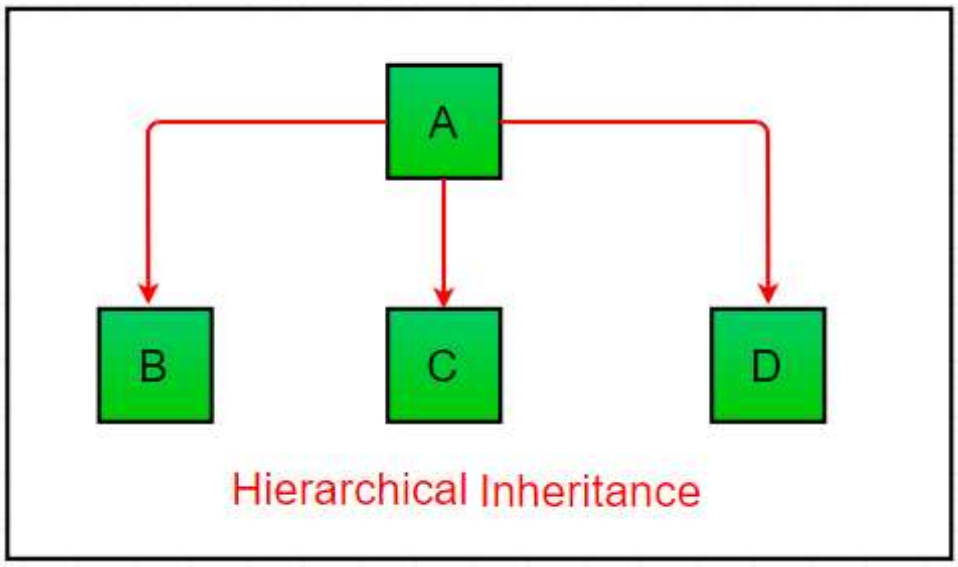


2. **Multilevel Inheritance:** In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In the below image, class A serves as a bas ass for the derived class B, which in turn

serves as a base class for the derived class C. In Java, a class cannot directly access the grandparent's members.



3. **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.



The following table describes the difference between the inheritance and interface:

Category	Inheritance	Interface
Description	Inheritance is the mechanism in java by which one class is allowed to inherit the features of another class.	Interface is the blueprint of the class. It specifies what a class must do and not how. Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).
Use	It is used to get the features of another class.	It is used to provide total abstraction.
Syntax	class subclass_name extends superclass_name { }	interface <interface_name>{ }
Number of Inheritance	It is used to provide 4 types of inheritance. (multi-level, simple, hybrid and hierarchical inheritance)	It is used to provide 1 types of inheritance (multiple).
Keywords	It uses extends keyword.	It uses implements keyword.
Inheritance	We can inherit lesser classes than Interface if we use Inheritance.	We can inherit enormously more classes than Inheritance, if we use Interface.
Method Definition	Methods can be defined inside the class in case of Inheritance.	Methods cannot be defined inside the class in case of Interface (except by using static and default keywords).
Overloading	It overloads the system if we try to extend a lot of classes.	System is not overloaded, no matter how many classes we implement.
Functionality Provided	It does not provide the functionality of loose	It provides the functionality of loose coupling.

Category	Inheritance	Interface
	coupling	
Multiple Inheritance	We cannot do multiple inheritance (causes compile time error).	We can do multiple inheritance using interfaces.

Last Updated : 21 Jun, 2020

Similar Reads

1. Inheritance of Interface in Java with Examples
2. Why to Use Comparator Interface Rather than Comparable Interface in Java?

4. Difference Between ReadWriteLock Interface and ReentrantReadWriteLock Class in Java
5. Difference between Abstract Class and Interface in Java
6. Difference Between poll() and remove() method of Queue Interface in Java
7. Difference between Priority Inversion and Priority Inheritance
8. Difference between Containership and Inheritance in C++
9. Difference between Single and Multiple Inheritance in C++
10. Difference between Inheritance and Polymorphism

Related Tutorials

1. Java 8 Tutorial
2. Graph Theory Tutorial
3. Discrete Mathematics Tutorial
4. Java IO Tutorial

Polymorphism in Java

PREMMAURYA

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

Real-life Illustration: Polymorphism

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 possesses different behavior in different situations. This is called polymorphism.
Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word “poly” means many and “morphs” means forms, So it means many forms.

Types of polymorphism

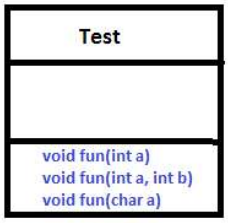
In Java polymorphism is mainly divided into two types:

- Compile-time Polymorphism
- Runtime Polymorphism

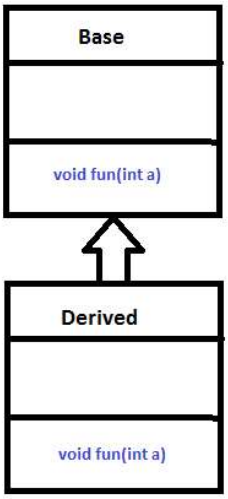
Type 1: Compile-time polymorphism

It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading.

> ***Note:*** *But Java doesn't support the Operator Overloading.*



Overloading



Overriding

Method Overloading: When there are multiple functions with the same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by changes in the number of arguments or/and a change in the type of arguments.

Example 1

Java

```
// Java Program for Method overloading
// By using Different Types of Arguments

// Class 1
// Helper class
class Helper {

    // Method with 2 integer parameters
    static int Multiply(int a, int b)
    {

        // Returns product of integer numbers
        return a * b;
    }

    // Method 2
    // With same name but with 2 double parameters
    static double Multiply(double a, double b)
```

```
    {

        // Returns product of double numbers
        return a * b;
    }
}

// Class 2
// Main class
class GFG {

    // Main driver method
```



```
    {

        // Calling method by passing
        // input as in arguments
        System.out.println(Helper.Multiply(2, 4));
        System.out.println(Helper.Multiply(5.5, 6.3));
    }
}
```

Output:

```
8
34.65
```

Example 2

Java

```
// Java program for Method Overloading
// by Using Different Numbers of Arguments

// Class 1
// Helper class
class Helper {

    // Method 1
```

```
    // Multiplication of 2 numbers
    static int Multiply(int a, int b)
    {

        // Return product
        return a * b;
    }

    // Method 2
    // // Multiplication of 3 numbers
    static int Multiply(int a, int b, int c)
    {

        // Return product
        return a * b * c;
    }
}

// Class 2
// Main class
class GFG {

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

        // Calling method by passing
        // input as in arguments
        System.out.println(Helper.Multiply(2, 4));
        System.out.println(Helper.Multiply(2, 7, 3));
    }
}
```

Output:

```
8
42
```

Subtypes of Compile-time Polymorphism:

1. **Function Overloading**: It is a feature in C++ where multiple functions can have the same name but with different parameter lists. The compiler will decide which function to call based on the number and types of arguments passed to the function.
2. **Operator Overloading:** It is a feature in C++ where the operators such as +, -, * etc. can be given additional meanings when applied to user-defined data types.
3. **template:** it is a powerful feature in C++ that allows us to write generic functions and classes. A template is a blueprint for creating a family of functions or classes.

Type 2: Runtime polymorphism

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

Example

Java

```
// Java Program for Method Overriding

// Class 1
// Helper class
class Parent {

    // Method of parent class
    void Print()
    {

        // Print statement
        System.out.println("parent class");
    }
}

// Class 2
// Helper class
class subclass1 extends Parent {

    // Method
    void Print() { System.out.println("subclass1"); }
}

// Class 3
// Helper class
class subclass2 extends Parent {

    // Method
    void Print()
    {

        // Print statement
        System.out.println("subclass2");
    }
}

// Class 4
// Main class
class GFG {

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

```
        // Creating object of class 1
        Parent a;

        // Now we will be calling print methods
        // inside main() method

        a = new subclass1();
        a.Print();

        a = new subclass2();
        a.Print();
    }
}
```

Output:

```
subclass1
subclass2
```

Output explanation:

Here in this program, When an object of child class is created, then the method inside the child class is called. This is because The method in the parent class is overridden by the child class. Since The method is overridden, This method has more priority than the parent method inside the child class. So, the body inside the child class is executed.

Subtype of Run-time Polymorphism:

1. **Virtual functions:** It allows an object of a derived class to behave as if it were an object of the base class. The derived class can override the virtual function of the base class to provide its own implementation. The function call is resolved at runtime, depending on the actual type of the object.

Diagram –

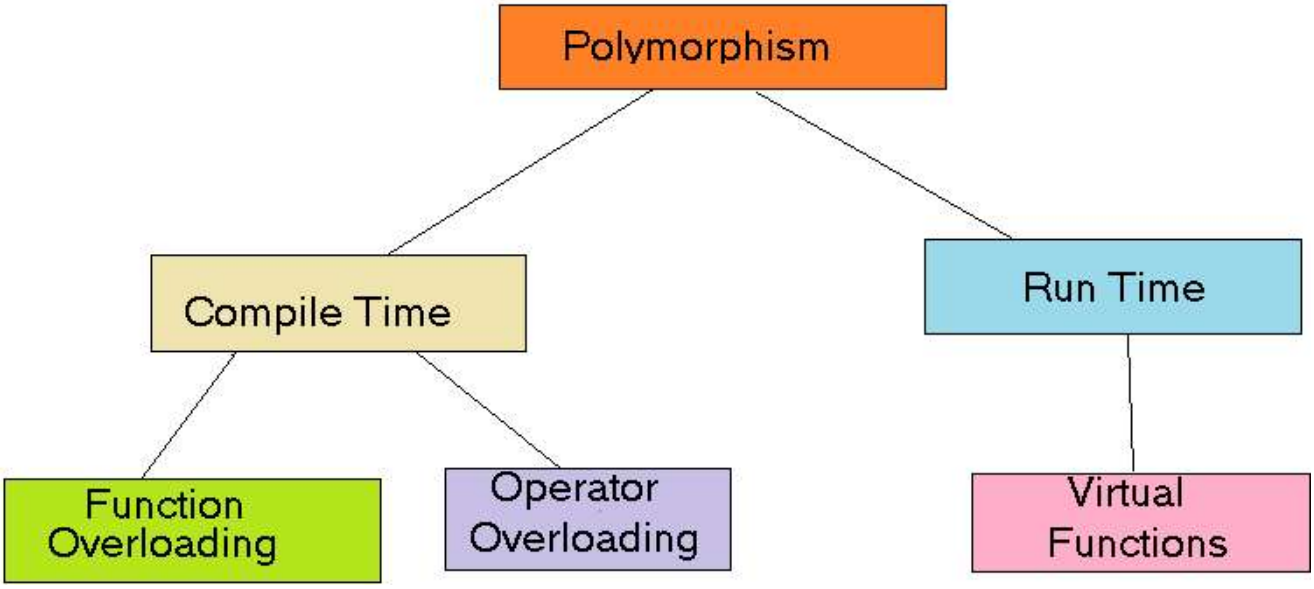


Fig – Types of polymorphism

Polymorphism in Java is a concept that allows objects of different classes to be treated as objects of a common class. It enables objects to behave differently based on their specific class type.

Advantages of Polymorphism in Java:

1. Increases code reusability by allowing objects of different classes to be treated as objects of a common class.
2. Improves readability and maintainability of code by reducing the amount of code that needs to be written and maintained.
3. Supports dynamic binding, enabling the correct method to be called at runtime, based on the actual class of the object.
4. Enables objects to be treated as a single type, making it easier to write generic code that can handle objects of different types.

Disadvantages of Polymorphism in Java:

1. Can make it more difficult to understand the behavior of an object, especially if the code is complex.
2. May lead to performance issues, as polymorphic behavior may require additional computations at runtime.

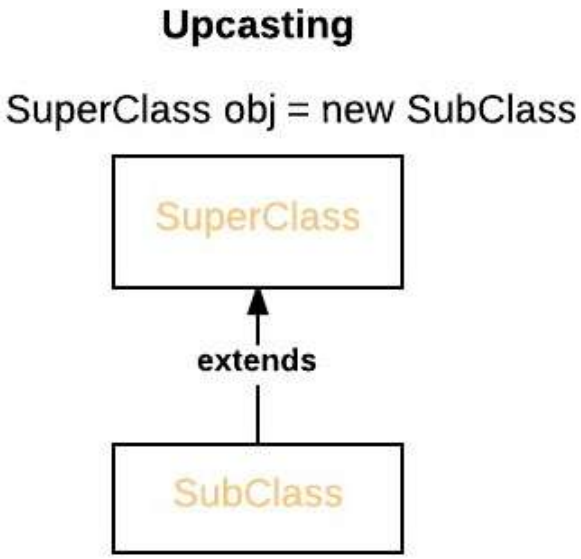
Similar Reads

Dynamic Method Dispatch or Runtime Polymorphism in Java

Prerequisite: Overriding in java, Inheritance

Method overriding is one of the ways in which Java supports Runtime Polymorphism. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

- When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.
- At run-time, it depends on the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed
- A superclass reference variable can refer to a subclass object. This is also known as upcasting. Java uses this fact to resolve calls to overridden methods at run time.



Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different

versions of the method are executed. Here is an example that illustrates dynamic method dispatch:


```
// A Java program to illustrate Dynamic Method
// Dispatch using hierarchical inheritance
class A
{
    void m1()
    {
        System.out.println("Inside A's m1 method");
    }
}

class B extends A
{
    // overriding m1()
    void m1()
    {
        System.out.println("Inside B's m1 method");
    }
}

class C extends A
{
    // overriding m1()
    void m1()
    {
        System.out.println("Inside C's m1 method");
    }
}

// Driver class
class Dispatch
{
    public static void main(String args[])
    {
        // object of type A
        A a = new A();

        // object of type B
        B b = new B();
```

```
        // object of type C
        C c = new C();

        // obtain a reference of type A
        A ref;

        // ref refers to an A object
        ref = a;

        // calling A's version of m1()
        ref.m1();

        // now ref refers to a B object
        ref = b;

        // calling B's version of m1()
        ref.m1();

        // now ref refers to a C object
        ref = c;

        // calling C's version of m1()
        ref.m1();
    }
}
```

Output:

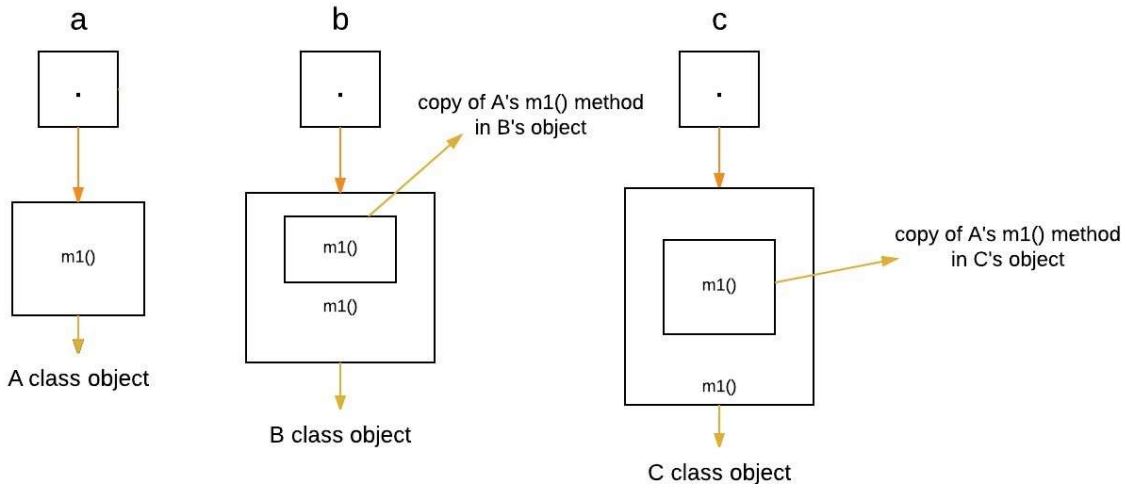
```
Inside A's m1 method
Inside B's m1 method
Inside C's m1 method
```

Explanation :

The above program creates one superclass called A and it's two subclasses B and C. These subclasses overrides m1( ) method.

1. Inside the main() method in Dispatch class, initially objects of type A, B, and C are declared.

   ```
   A a = new A(); // object of type A
   B b = new B(); // object of type B
   C c = new C(); // object of type C
   ```

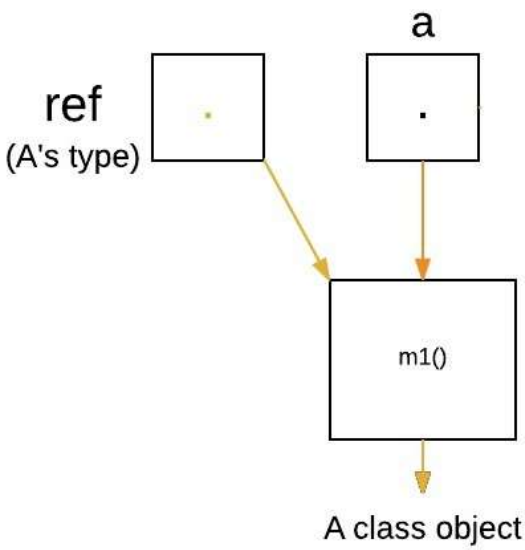


2. Now a reference of type A, called ref, is also declared, initially it will point to null.

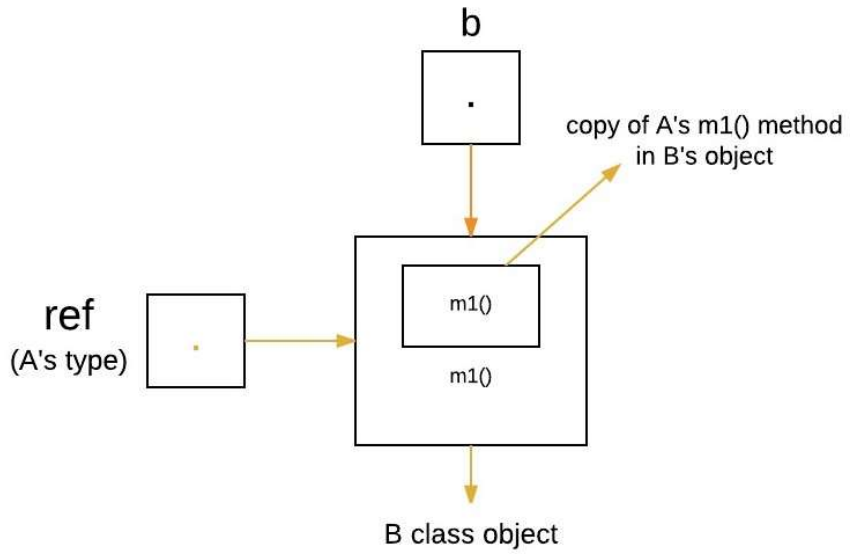
```
A ref; // obtain a reference of type A
```

3. Now we are assigning a reference to each **type of object** (either A's or B's or C's) to *ref*, one-by-one, and uses that reference to invoke m1( ). As the output shows, the version of m1( ) executed is determined **by the type of object being referred to at the time of the call.**

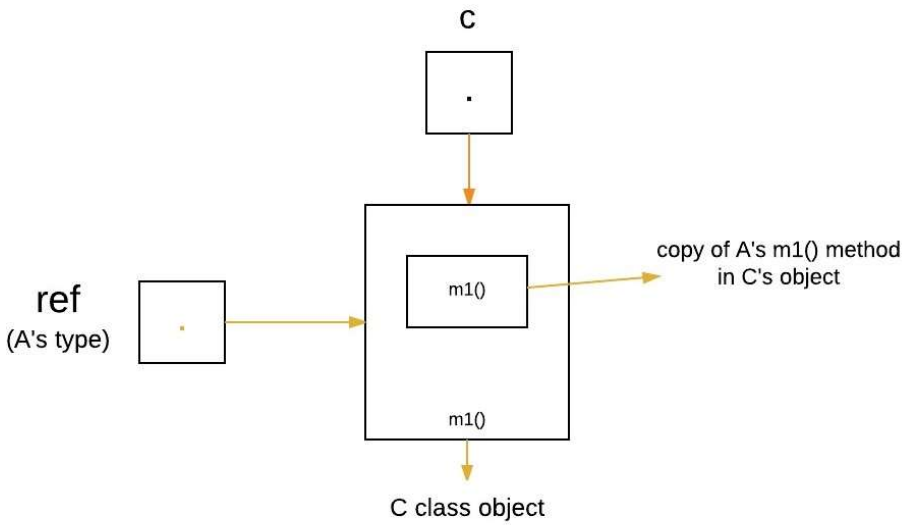
```
ref = a; // r refers to an A object
ref.m1(); // calling A's version of m1()
```




```
ref = b; // now r refers to a B object
ref.m1(); // calling B's version of m1()
```




```
ref = c; // now r refers to a C object
ref.m1(); // calling C's version of m1()
```



Runtime Polymorphism with Data Members

In Java, we can override methods only, not the variables(data members), so **runtime polymorphism cannot be achieved by data members.** For example :

```
// Java program to illustrate the fact that
// runtime polymorphism cannot be achieved
// by data members

// class A
class A
{
    int x = 10;
}

// class B
class B extends A
{
    int x = 20;
}

// Driver class
public class Test
{
    public static void main(String args[])
    {
        A a = new B(); // object of type B

        // Data member of class A will be accessed
        System.out.println(a.x);
    }
}
```

Output:

```
10
```

Explanation : In above program, both the class A(super class) and B(sub class) have a common variable 'x'. Now we make object of class B, referred by 'a' which is of type of class A. Since variables are not overridden, so the statement "a.x" will **always** refer to data member of super class.

Advantages of Dynamic Method Dispatch

1. Dynamic method dispatch allow Java to support overriding of methods which is central for run-time polymorphism.
2. It allows a class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.

3. It also allow subclasses to add its specific methods subclasses to define the specific implementation of some.

Static vs Dynamic binding

- Static binding is done during compile-time while dynamic binding is done during run-time.
- private, final and static methods and variables uses static binding and bonded by compiler while overridden methods are bonded during runtime based upon type of runtime object

This article is contributed by **Gaurav Miglani**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Last Updated : 07 Sep, 2018

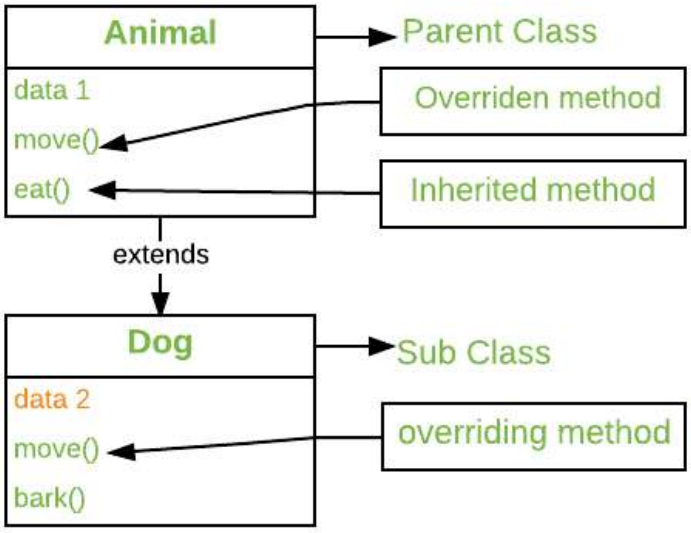
250

Similar Reads

1. Virtual Functions and Runtime Polymorphism in C++
2. Java.lang.Runtime class in Java
3. Difference between Compile-time and Run-time Polymorphism in Java
4. Interfaces and Polymorphism in Java
5. Variables in Java Do Not Follow Polymorphism and Overriding
6. Compile Time Polymorphism in Java
7. Polymorphism in Java
8. Calling an External Program in Java using Process and Runtime
9. Java Program to Handle Runtime Exceptions

Overriding in Java

In any object-oriented programming language, Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature, and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to *override* the method



in the super-class. Method overriding is one of the way by which java achieve Run Time Polymorphism.The version of a method that is executed will be determined by the object that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed. In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed.

Java

```java
// A Simple Java program to demonstrate
// method overriding in java

// Base Class
class Parent {
    void show()
    {
        System.out.println("Parent's show()");
    }
}

// Inherited class
class Child extends Parent {
    // This method overrides show() of Parent
    @Override
    void show()
    {
        System.out.println("Child's show()");
    }
}

// Driver class
class Main {
    public static void main(String[] args)
    {
        // If a Parent type reference refers
        // to a Parent object, then Parent's
        // show is called
        Parent obj1 = new Parent();
        obj1.show();

        // If a Parent type reference refers
        // to a Child object Child's show()
        // is called. This is called RUN TIME
        // POLYMORPHISM.
        Parent obj2 = new Child();
        obj2.show();
    }
}
```

Output:

```
Parent's show()
Child's show()
```

Rules for method overriding:

1. **Overriding and Access-Modifiers :** The access modifier for an overriding method can allow more, but not less, access than the overridden method. For example, a protected

subclass. Doing so, will generate compile-time error.

Java

```java
// A Simple Java program to demonstrate
// Overriding and Access-Modifiers

class Parent {
    // private methods are not overridden
    private void m1()
    {
        System.out.println("From parent m1()");
    }

    protected void m2()
    {
        System.out.println("From parent m2()");
    }
}

class Child extends Parent {
    // new m1() method
    // unique to Child class
    private void m1()
    {
        System.out.println("From child m1()");
    }

    // overriding method
    // with more accessibility
    @Override
    public void m2()
    {
        System.out.println("From child m2()");
    }
}

// Driver class
class Main {
    public static void main(String[] args)
    {
```

```
        Parent obj1 = new Parent();
        obj1.m2();
        Parent obj2 = new Child();
        obj2.m2();
    }
}
```

Output:

```
From parent m2()
From child m2()
```

1. **Final methods can not be overridden :** If we don't want a method to be overridden, we declare it as final. Please see Using final with Inheritance .

Java

```
// A Java program to demonstrate that
// final methods cannot be overridden

class Parent {
    // Can't be overridden
    final void show() {}
}

class Child extends Parent {
    // This would produce error
    void show() {}
}
```

1. **Output:**

```
 13: error: show() in Child cannot override show() in Parent
     void show() {  }
          ^
   overridden method is final
```

1. **Static methods can not be overridden(Method Overriding vs Method Hiding) :** When you define a static method with same signature as a static method in base class, it is known as method hiding. The following table summarizes what happens when you define a method with the same signature as a method in a super-class.

	Superclass Instance Method	Superclass Static Method
Subclass Instance Method	Overrides	Generates a compile-time error
Subclass Static Method	Generates a compile-time error	Hides

Java

```java
// Java program to show that
// if the static method is redefined by
// a derived class, then it is not
// overriding, it is hiding

class Parent {
    // Static method in base class
    // which will be hidden in subclass
    static void m1()
    {
        System.out.println("From parent "
                           + "static m1()");
    }

    // Non-static method which will
    // be overridden in derived class
    void m2()
    {
        System.out.println("From parent "
                           + "non-static(instance) m2()");
    }
}

class Child extends Parent {
    // This method hides m1() in Parent
    static void m1()
    {
        System.out.println("From child static m1()");
```

```
    }

    // This method overrides m2() in Parent
    @Override
    public void m2()
    {
        System.out.println("From child "
                           + "non-static(instance) m2()");
    }
}

// Driver class
class Main {
    public static void main(String[] args)
    {
        Parent obj1 = new Child();

        // As per overriding rules this
        // should call to class Child static
        // overridden method. Since static
        // method can not be overridden, it
        // calls Parent's m1()
        obj1.m1();

        // Here overriding works
        // and Child's m2() is called
        obj1.m2();
    }
}
```

Output:

```
From parent static m1()
From child non-static(instance) m2()
```

1. **Private methods can not be overridden :** Private methods cannot be overridden as they are bonded during compile time. Therefore we can't even override private methods in a subclass.(See this for details).

Java

```
class SuperClass {
    private void privateMethod() {
        System.out.println("This is a private method in SuperClass");
    }

    public void publicMethod() {
        System.out.println("This is a public method in SuperClass");
        privateMethod();
    }
}
```

```
class SubClass extends SuperClass {
    // This is a new method with the same name as the private method in SuperClass
    private void privateMethod() {
        System.out.println("This is a private method in SubClass");
    }

    // This method overrides the public method in SuperClass
    public void publicMethod() {
        System.out.println("This is a public method in SubClass");
        privateMethod();  // calls the private method in SubClass, not SuperClass
    }
}

public class Test {
    public static void main(String[] args) {
        SuperClass obj1 = new SuperClass();
        obj1.publicMethod();  // calls the public method in SuperClass

        SubClass obj2 = new SubClass();
        obj2.publicMethod();  // calls the overridden public method in SubClass
    }
}
```

1. **The overriding method must have same return type (or subtype) :** From Java 5.0 onwards it is possible to have different return type for a overriding method in child class, but child's return type should be sub-type of parent's return type. This phenomena is known as **covariant return type**.

Java

```
class SuperClass {
    public Object method() {
        System.out.println("This is the method in SuperClass");
        return new Object();
    }
}

class SubClass extends SuperClass {
    public String method() {
        System.out.println("This is the method in SubClass");
        return "Hello, World!";
    }
}

public class Test {
    public static void main(String[] args) {
        SuperClass obj1 = new SuperClass();
        obj1.method();

        SubClass obj2 = new SubClass();
        obj2.method();
    }
```

```
}
```

1. **Invoking overridden method from sub-class :** We can call parent class method in overriding method using super keyword.

Java

```java
// A Java program to demonstrate that overridden
// method can be called from sub-class

// Base Class
class Parent {
    void show()
    {
        System.out.println("Parent's show()");
    }
}

// Inherited class
class Child extends Parent {
    // This method overrides show() of Parent
    @Override
    void show()
    {
        super.show();
        System.out.println("Child's show()");
    }
}

// Driver class
class Main {
    public static void main(String[] args)
    {
        Parent obj = new Child();
        obj.show();
    }
}
```

Output:

```
Parent's show()
Child's show()
```

1. **Overriding and constructor :** We can not override constructor as parent and child class can never have constructor with same name(Constructor name must always be same as Class name).
2. **Overriding and Exception-Handling :** Below are two rules to note when overriding methods related to exception-handling.

- **Rule#1 :** If the super-class overridden method does not throw an exception, subclass overriding method can only throws the unchecked exception, throwing checked exception will lead to compile-time error.

Java

```
/* Java program to demonstrate overriding when
  superclass method does not declare an exception
*/

class Parent {
    void m1()
    {
        System.out.println("From parent m1()");
    }

    void m2()
    {
        System.out.println("From parent  m2()");
    }
}

class Child extends Parent {
    @Override
    // no issue while throwing unchecked exception
    void m1() throws ArithmeticException
    {
        System.out.println("From child m1()");
    }

    @Override
    // compile-time error
    // issue while throwing checked exception
    void m2() throws Exception
    {
        System.out.println("From child m2");
    }
}
```

- **Output:**

```
 error: m2() in Child cannot override m2() in Parent
     void m2() throws Exception{ System.out.println("From child m2");}
          ^
   overridden method does not throw Exception
```

- **Rule#2 :** If the super-class overridden method does throws an exception, subclass overriding method can only throw same, subclass exception. Throwing parent

exception in Exception hierarchy will lead to compile time error.Also there is no issue if subclass overridden method is not throwing any exception.

Java

```
// Java program to demonstrate overriding when
// superclass method does declare an exception

class Parent {
    void m1() throws RuntimeException
    {
        System.out.println("From parent m1()");
    }
}

class Child1 extends Parent {
    @Override
    // no issue while throwing same exception
    void m1() throws RuntimeException
    {
        System.out.println("From child1 m1()");
    }
}
class Child2 extends Parent {
    @Override
    // no issue while throwing subclass exception
    void m1() throws ArithmeticException
    {
        System.out.println("From child2 m1()");
    }
}
class Child3 extends Parent {
    @Override
    // no issue while not throwing any exception
    void m1()
    {
        System.out.println("From child3 m1()");
    }
}
class Child4 extends Parent {
    @Override
    // compile-time error
    // issue while throwing parent exception
    void m1() throws Exception
    {
        System.out.println("From child4 m1()");
    }
}
```

- **Output:**

```
 error: m1() in Child4 cannot override m1() in Parent
     void m1() throws Exception
          ^
   overridden method does not throw Exception
```

1. **Overriding and abstract method:** Abstract methods in an interface or abstract class are meant to be overridden in derived concrete classes otherwise a compile-time error will be thrown.
2. **Overriding and synchronized/strictfp method :** The presence of synchronized/strictfp modifier with method have no effect on the rules of overriding, i.e. it's possible that a synchronized/strictfp method can override a non synchronized/strictfp one and vice-versa.

Note :

1. In C++, we need virtual keyword to achieve overriding or Run Time Polymorphism. In Java, methods are virtual by default.
2. We can have multilevel method-overriding.

Java

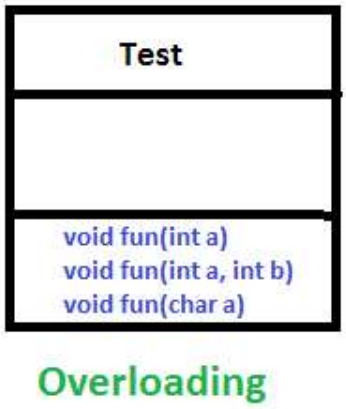
```
// A Java program to demonstrate
// multi-level overriding

// Base Class
class Parent {
    void show()
    {
        System.out.println("Parent's show()");
    }
}

// Inherited class
class Child extends Parent {
    // This method overrides show() of Parent
    void show() { System.out.println("Child's show()"); }
}

// Inherited class
class GrandChild extends Child {
    // This method overrides show() of Parent
    void show()
    {
        System.out.println("GrandChild's show()");
    }
}

// Driver class
```

```
class Main {
    public static void main(String[] args)
    {
        Parent obj1 = new GrandChild();
        obj1.show();
    }
}
```

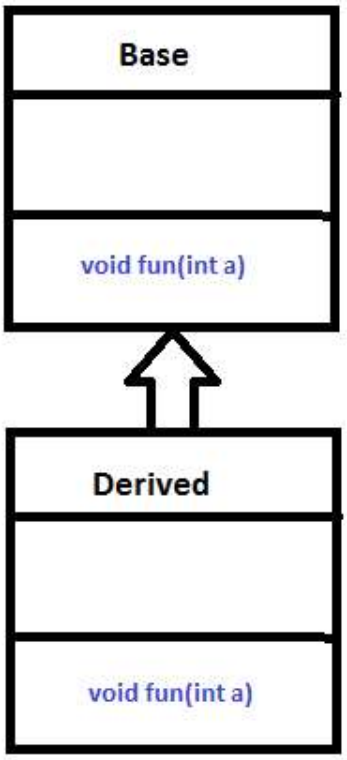
Output:

```
GrandChild's show()
```

1. **Overriding vs Overloading :**
    1. Overloading is about same method have different signatures. Overriding is about same method, same signature but different classes connected through inheritance.



Overloading



Overriding

    2. Overloading is an example of compiler-time polymorphism and overriding is an example of run time polymorphism.

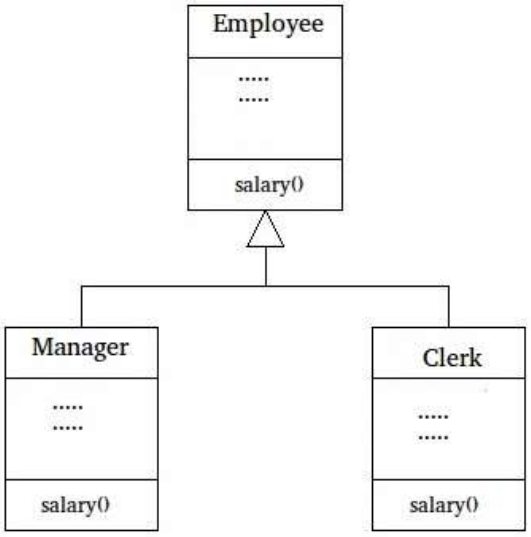
Why Method Overriding ?

As stated earlier, overridden methods allow Java to support run-time polymorphism. Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives while allowing subclasses to define the specific implementation of some or all of those methods. Overridden methods are another way that Java implements the “one interface, multiple methods” aspect of polymorphism. Dynamic Method Dispatch is one of the most

powerful mechanisms that object-oriented design brings to bear on code reuse and robustness. The ability to exist code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool. Overridden methods allow us to call methods of any of the derived classes without even knowing the type of derived class object.

When to apply Method Overriding ?(with example)

Overriding and Inheritance : Part of the key to successfully applying polymorphism is understanding that the superclasses and subclasses form a hierarchy which moves from lesser to greater specialization. Used correctly, the superclass provides all elements that a subclass can use directly. It also defines those methods that the derived class must implement on its own. This allows the subclass the flexibility to define its methods, yet still enforces a consistent interface. **Thus, by combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses.** Let's look at a more practical example that uses method overriding. Consider an employee management software for an organization, let the code has a simple base class Employee, the class has methods like raiseSalary(), transfer(), promote(), .. etc. Different types of employees like Manager, Engineer, ..etc may have their implementations of the methods present in base class Employee. In our complete software, we just need to pass a list of employees everywhere and call appropriate methods without even knowing the type of employee. For example, we can easily raise the salary of all employees by iterating through the list of employees. Every type of employee may have its logic in its class, we don't need to worry because if raiseSalary() is present for a specific employee type, only that method would be called.



Java

```java
// A Simple Java program to demonstrate application
// of overriding in Java

// Base Class
class Employee {
    public static int base = 10000;
    int salary()
    {
        return base;
    }
}

// Inherited class
class Manager extends Employee {
    // This method overrides salary() of Parent
    int salary()
    {
        return base + 20000;
    }
}

// Inherited class
class Clerk extends Employee {
    // This method overrides salary() of Parent
    int salary()
    {
        return base + 10000;
    }
}

// Driver class
class Main {
    // This method can be used to print the salary of
    // any type of employee using base class reference
    static void printSalary(Employee e)
    {
        System.out.println(e.salary());
    }

    public static void main(String[] args)
    {
        Employee obj1 = new Manager();

        // We could also get type of employee using
        // one more overridden method.loke getType()
        System.out.print("Manager's salary : ");
        printSalary(obj1);

        Employee obj2 = new Clerk();
        System.out.print("Clerk's salary : ");
        printSalary(obj2);
    }
}
```

Output:

```
Manager's salary : 30000
Clerk's salary : 20000
```

Related Article:

- Dynamic Method Dispatch or Runtime Polymorphism in Java
- Overriding equals() method of Object class
- Overriding toString() method of Object class
- Overloading in java
- Output of Java program | Set 18 (Overriding)

This article is contributed by **Twinkle Tyagi and Gaurav Miglani**. If you like GeeksforGeeks and would like to contribute, you can also write an article using write.geeksforgeeks.org or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Last Updated : 30 Mar, 2023

225

Similar Reads

1. Overriding equals method in Java
2. Overriding toString() Method in Java
3. Overriding methods from different packages in Java
4. Exception Handling with Method Overriding in Java
5. Difference Between Method Overloading and Method Overriding in Java
6. Different Ways to Prevent Method Overriding in Java
7. Java Program to Use Method Overriding in Inheritance for Subclasses
8. Java - Covariant Method Overriding with Examples
9. When We Need to Prevent Method Overriding in Java ?

Difference Between Method Overloading and Method Overriding in Java

MKS075

The differences between Method Overloading and Method Overriding in Java are as follows:

Method Overloading	Method Overriding
Method overloading is a compile-time polymorphism.	Method overriding is a run-time polymorphism.
It helps to increase the readability of the program.	It is used to grant the specific implementation of the method which is already provided by its parent class or superclass.
It occurs within the class.	It is performed in two classes with inheritance relationships.
Method overloading may or may not require inheritance.	Method overriding always needs inheritance.
In method overloading, methods must have the same name and different signatures.	In method overriding, methods must have the same name and same signature.
In method overloading, the return type can or can not be the same, but we just have to change the parameter.	In method overriding, the return type must be the same or co-variant.

Method Overloading	Method Overriding
Static binding is being used for overloaded methods.	Dynamic binding is being used for overriding methods.
Poor Performance due to compile time polymorphism.	It gives better performance. The reason behind this is that the binding of overridden methods is being done at runtime.
Private and final methods can be overloaded.	Private and final methods can't be overridden.
Argument list should be different while doing method overloading.	Argument list should be same in method overriding.

Method Overloading**:**

Method Overloading is a **Compile time polymorphism**. In method overloading, more than one method shares the same method name with a different signature in the class. In method overloading, the return type can or can not be the same, but we have to change the parameter because, in java, we can not achieve the method overloading by changing only the return type of the method.

Example of Method Overloading:

Java

```
import java.io.*;
```

```java
class MethodOverloadingEx {

    static int add(int a, int b)
    {
      return a + b;
    }

    static int add(int a, int b, int c)
    {
        return a + b + c;
    }

    public static void main(String args[])
    {
          System.out.println("add() with 2 parameters");
        System.out.println(add(4, 6));

          System.out.println("add() with 3 parameters");
        System.out.println(add(4, 6, 7));
    }
}
```

Output

```
add() with 2 parameters
10
add() with 3 parameters
17
```

Method Overriding:

Method Overriding is a **Run time polymorphism**. In method overriding, the derived class provides the specific implementation of the method that is already provided by the base class or parent class. In method overriding, the return type must be the same or co-variant (return type may vary in the same direction as the derived class).

Example: Method Overriding

Java

```java
import java.io.*;

class Animal {

    void eat()
    {
        System.out.println("eat() method of base class");
        System.out.println("eating.");
```

```
    }
}

class Dog extends Animal {

    void eat()
    {
        System.out.println("eat() method of derived class");
        System.out.println("Dog is eating.");
    }
}

class MethodOverridingEx {

    public static void main(String args[])
    {
        Dog d1 = new Dog();
        Animal a1 = new Animal();

        d1.eat();
        a1.eat();

        Animal animal = new Dog();
        // eat() method of animal class is overridden by
        // base class eat()
        animal.eat();
    }
}
```

C++

```
#include<iostream>
#include<stdio.h>
using namespace std;
class Animal {
     public:
    void eat()
    {
        cout<<"eat() method of base class"<<endl;
        cout<<"eating."<<endl;
    }
};

class Dog:public Animal {
     public:
    void eat()
    {
        cout<<"eat() method of derived class"<<endl;
        cout<<"Dog is eating."<<endl;
    }
};

int main()
{
    Dog d1;
```

```
    d1.eat();

    Animal a1;
    a1.eat();

    return 0;
}
```

Output

```
eat() method of derived class
Dog is eating.
eat() method of base class
eating.
eat() method of derived class
Dog is eating.
```

Output explanation: Here, we can see that a method eat() has overridden in the derived class name **Dog** that is already provided by the base class name **Animal**. When we create the instance of class Dog and call the eat() method, we see that only derived class eat() method run instead of base class method eat(), and When we create the instance of class Animal and call the eat() method, we see that only base class eat() method run instead of derived class method eat().

Last Updated : 22 Oct, 2022

Similar Reads

1. Difference between Method Overloading and Method Overriding in Python
2. Function Overloading vs Function Overriding in C++
3. Overriding equals method in Java
4. Overriding toString() Method in Java
5. Exception Handling with Method Overriding in Java
6. Java - Covariant Method Overriding with Examples
7. When We Need to Prevent Method Overriding in Java ?

Inheritance in Java

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/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:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

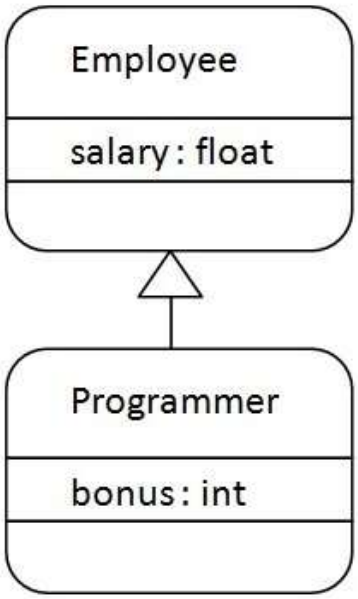
The syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Java Inheritance Example



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
```

```
    System.out.println("Programmer salary is:"+p.salary);
    System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

Test it Now

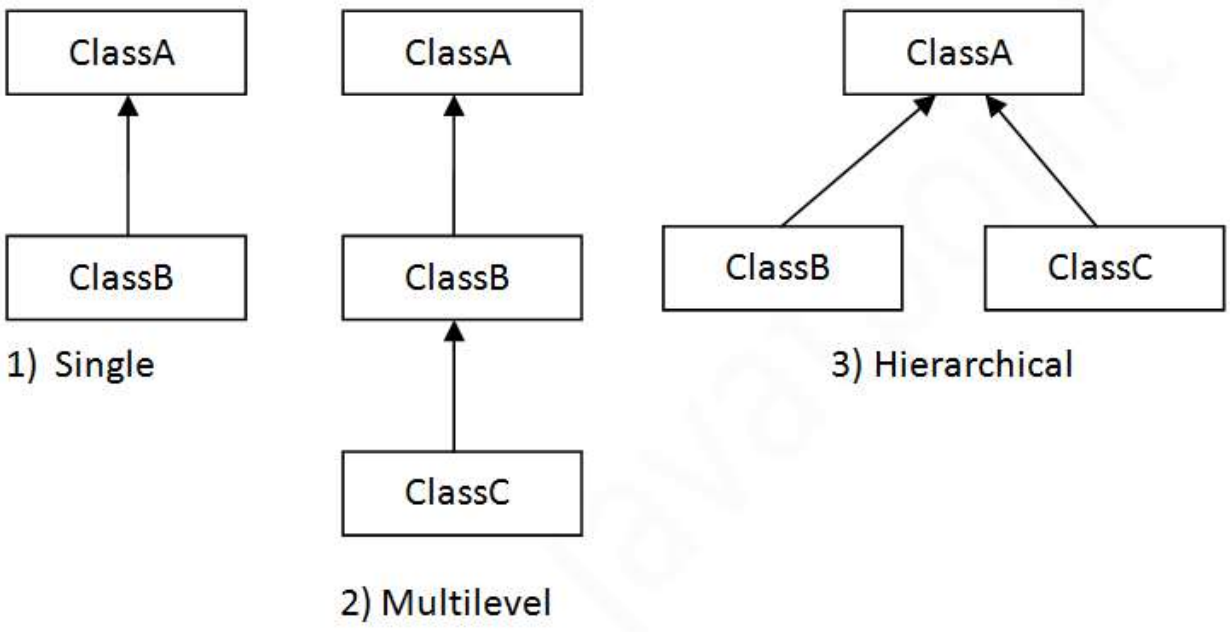
```
Programmer salary is:40000.0
Bonus of programmer is:10000
```

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

Types of inheritance in java

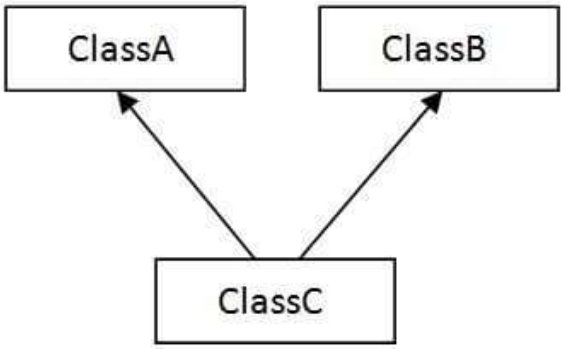
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

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

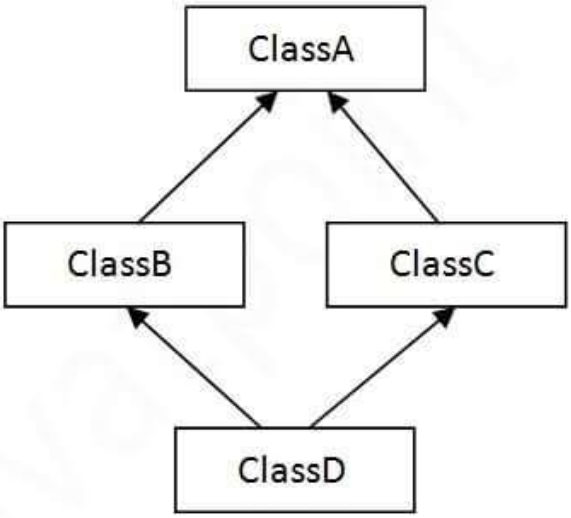


> Note: Multiple inheritance is not supported in Java through class

When one class inherits multiple classes, it is known as mult



4) Multiple



5) Hybrid

.

Single Inheritance Example

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

File: TestInheritance.java

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

Output:

```
barking...
eating...
```

Multilevel Inheritance Example

When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

File: TestInheritance2.java

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```

Output:

```
weeping...
barking...
eating...
```

Hierarchical Inheritance Example

When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

File: TestInheritance3.java

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}}
```

Output:

```
meowing...
eating...
```

Q) Why multiple inheritance is not supported in java?

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

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Ja
So whether you have same method or different, there will b

```
class A{
void msg(){System.out.println("Hello");}
```

```
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were

 public static void main(String args[]){
   C obj=new C();
   obj.msg();//Now which msg() method would be invoked?
}
}
```

Test it Now

```
Compile Time Error
```

← Prev

Next →

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share

Aggregation in Java

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

Consider a situation, Employee object contains many informations such as id, name, emailId etc. It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc. as given below.

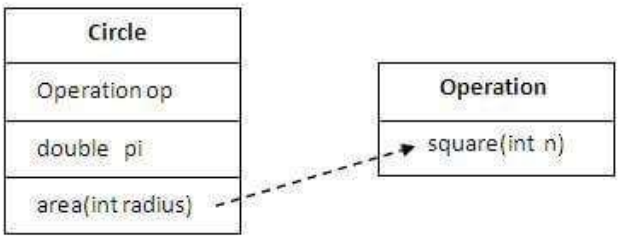
```
class Employee{
int id;
String name;
Address address;//Address is a class
...
}
```

In such case, Employee has an entity reference address, so relationship is Employee HAS-A address.

Why use Aggregation?

- For Code Reusability.

Simple Example of Aggregation



In this example, we have created the reference of Operation class in the Circle class.

```
class Operation{
```

```
 int square(int n){
  return n*n;
 }
}

class Circle{
 Operation op;//aggregation
 double pi=3.14;

 double area(int radius){
   op=new Operation();
   int rsquare=op.square(radius);//code reusability (i.e. delegates the method call).
   return pi*rsquare;
 }



 public static void main(String args[]){
   Circle c=new Circle();
   double result=c.area(5);
   System.out.println(result);
 }
}
```

Test it Now

```
Output:78.5
```

When use Aggregation?

- Code reuse is also best achieved by aggregation when there is no is-a relationship.
- Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

Understanding meaningful example of Aggr

In this example, Employee has an object of Address, addre
state, country etc. In such case relationship is Employee HA

Address.java

```
public class Address {
String city,state,country;

public Address(String city, String state, String country) {
    this.city = city;
    this.state = state;
    this.country = country;
}

}
```

Emp.java

```
public class Emp {
int id;
String name;
Address address;

public Emp(int id, String name,Address address) {
    this.id = id;
    this.name = name;
    this.address=address;
}

void display(){
System.out.println(id+" "+name);
System.out.println(address.city+" "+address.state+" "+address.country);
}

public static void main(String[] args) {
Address address1=new Address("gzb","UP","india");
Address address2=new Address("gno","UP","india");

Emp e=new Emp(111,"varun",address1);
Emp e2=new Emp(112,"arun",address2);

e.display();
e2.display();

}
```

```
}
```

Test it Now

```
Output:111 varun
       gzb UP india
       112 arun
       gno UP india
```

download this example

← Prev

Next →

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share

Learn Latest Tutorials


Splunk


SPSS


Swagger

Java Inheritance

‹ Previous Next ›

Java Inheritance (Subclass and Superclass)

In Java, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

- **subclass** (child) - the class that inherits from another class
- **superclass** (parent) - the class being inherited from

To inherit from a class, use the `extends` keyword.

In the example below, the `Car` class (subclass) inherits the attributes and methods from the `Vehicle` class (superclass):

Example

Get your own Java Server

```
class Vehicle {
  protected String brand = "Ford";        // Vehicle attribute
  public void honk() {                    // Vehicle method
    System.out.println("Tuut, tuut!");
  }
}

class Car extends Vehicle {
  private String modelName = "Mustang";    // Car attribute
  public static void main(String[] args) {
```

```
    // Call the honk() method (from the Vehicle class) on the myCar object
    myCar.honk();

    // Display the value of the brand attribute (from the Vehicle class) and
    System.out.println(myCar.brand + " " + myCar.modelName);
  }
}
```

Try it Yourself »

Did you notice the `protected` modifier in Vehicle?

We set the **brand** attribute in **Vehicle** to a `protected` access modifier. If it was set to `private`, the Car class would not be able to access it.

Why And When To Use "Inheritance"?

when you create a new class.

Tip: Also take a look at the next chapter, Polymorphism, which uses inherited methods to perform different tasks.

The final Keyword

If you don't want other classes to inherit from a class, use the `final` keyword:

```
final class Vehicle {
  ...
}

class Car extends Vehicle {
  ...
}
```

The output will be something like this:

```
Main.java:9: error: cannot inherit from final Vehicle
class Main extends Vehicle {
                   ^
1 error)
```

Try it Yourself »

‹ Previous

Next ›