FINAL REPORT 7
JAVA INHERITANCE

Explanations

Inheritance in Java

Inheritance is an important pillar of OOP(Object Oriented Programming). It is the mechanism in java by which one class is allow to inherit the features(fields and methods) of another class. **Important terminology:**

- **Super Class:** The class whose features are inherited is known as super class(or a base class or a parent class).
- **Sub Class:** The class that inherits the other class is known as sub class(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.

The keyword used for inheritance is **extends**.
Syntax :

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

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

filter_none

edit

play_arrow

brightness_4

```
//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);
    }
}
```

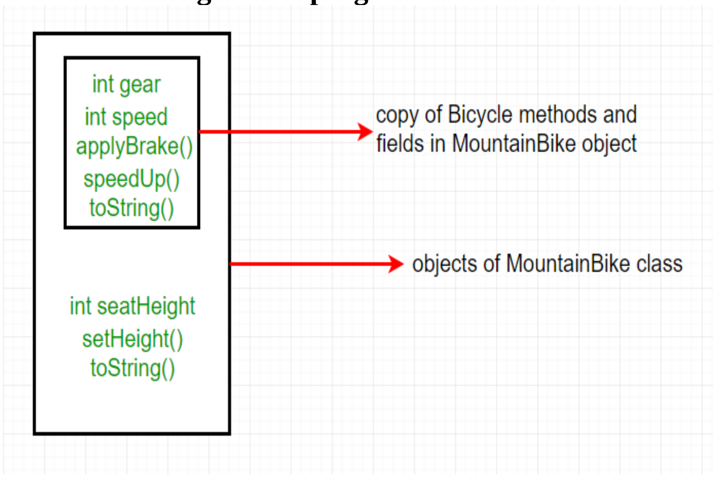
```java
// derived class
class MountainBike extends Bicycle
{

    // 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 above program, when an object of MountainBike class is created, a copy of the all methods and fields of the superclass acquire 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 object of subclass is created, not the superclass.

Illustrative image of the program:

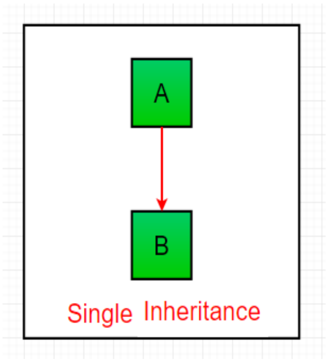

int gear
int speed
applyBrake()
speedUp()
toString()
copy of Bicycle methods and fields in MountainBike object
objects of MountainBike class
int seatHeight
setHeight()
toString()

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

Types of Inheritance in Java

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

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



filter_none

edit

play_arrow

brightness_4

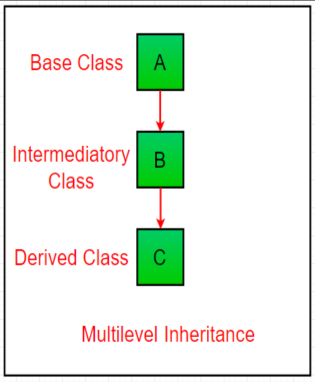
```
//Java program to illustrate the
// concept of single inheritance
import java.util.*;
import java.lang.*;
import java.io.*;

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

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

Output:

```
Geeks
for
Geeks
```

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 below image, the 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.



filter_none

edit

play_arrow

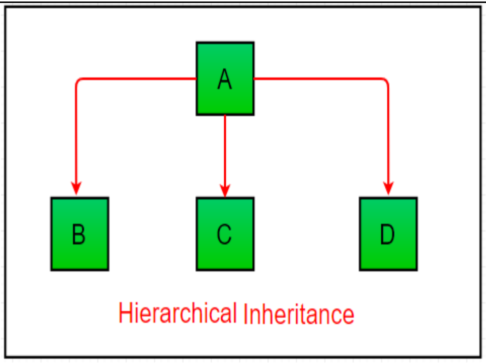
brightness_4

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

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 sub class.In below image, the class A serves as a base class for the derived class B,C and D.



filter_none

edit

play_arrow

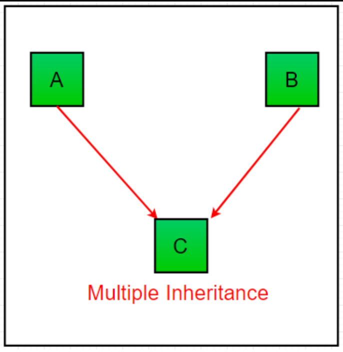
brightness_4

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


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 one

{

    /*............*/

}


// Drived class

public class Main

{

    public static void main(String[] args)

    {

        three g = new three();

        g.print_geek();

        two t = new two();

        t.print_for();

        g.print_geek();

    }

}
```

Output:

```
Geeks
for
Geeks
```

4. **Multiple Inheritance (Through Interfaces) :** In Multiple inheritance ,one class can have more than one superclass and inherit features from all parent classes. Please note that Java does **not** support multiple inheritance with classes. In java, we can achieve multiple inheritance only through Interfaces. In image below, Class C is derived from interface A and B.



filter_none

edit

play_arrow

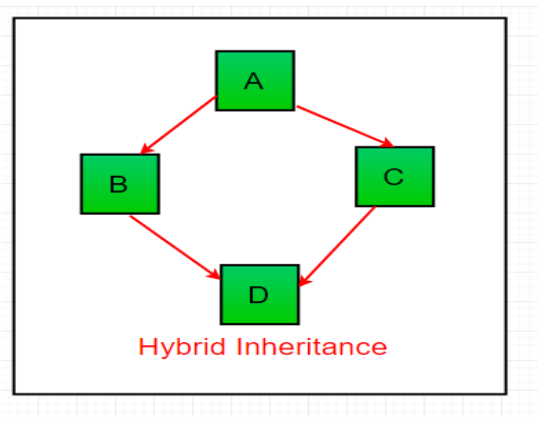
brightness_4

```
// Java program to illustrate the

// concept of Multiple inheritance

import java.util.*;

import java.lang.*;

import java.io.*;


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
Geeks
```

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 inheritance with classes, the hybrid inheritance is also not possible with classes. In java, we can achieve hybrid inheritance only through Interfaces.



Important facts about inheritance in Java

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

What all can be done in a Subclass?

In sub-classes we can inherit members as is, replace them, hide them, or supplement them with new members:

- The inherited fields can be used directly, just like any other fields.
- We can declare new fields in the subclass that are not in the superclass.
- The inherited methods can be used directly as they are.
- We can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus overriding it (as in 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.

Interface in Java

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).

- Interfaces specify what a class must do and not how. It is the blueprint of the class.
- An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.
- A Java library example is, Comparator Interface. If a class implements this interface, then it can be used to sort a collection.

Syntax :

```
interface <interface_name> {

    // declare constant fields
    // declare methods that abstract
    // by default.
}
```

To declare an interface, use **interface** keyword. It is used to provide total abstraction. That means all the methods in an interface are declared with an empty body and are public and all fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface. To implement interface use **implements** keyword.

Why do we use interface ?

- It is used to achieve total abstraction.
- Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance .
- It is also used to achieve loose coupling.
- Interfaces are used to implement abstraction. So the question arises why use interfaces when we have abstract classes?

  The reason is, abstract classes may contain non-final variables, whereas variables in interface are final, public and static.

  filter_none

  edit

  play_arrow

  brightness_4

  ```
  // A simple interface

  interface Player

  {

      final int id = 10;

      int move();
  ```

```
        }
```

To implement an interface we use keyword: implements

filter_none

edit

play_arrow

brightness_4

```
// Java program to demonstrate working of
// interface.
import java.io.*;

// A simple interface
interface In1
{
    // public, static and final
    final int a = 10;

    // public and abstract
    void display();
}

// A class that implements the interface.
class TestClass implements In1
{
    // Implementing the capabilities of
    // interface.
    public void display()
    {
        System.out.println("Geek");
    }



    // Driver Code

    public static void main (String[] args)

    {

        TestClass t = new TestClass();

        t.display();

        System.out.println(a);

    }

}
```

Output:

```
Geek
10
```

A real-world example:
Let’s consider the example of vehicles like bicycle, car, bike………, they have common functionalities. So we make an interface and put all these common functionalities. And lets Bicycle, Bike, car ….etc implement all these functionalities in their own class in their own way.
filter_none

edit

play_arrow

brightness_4

```
import java.io.*;



interface Vehicle {



    // all are the abstract methods.

    void changeGear(int a);

    void speedUp(int a);

    void applyBrakes(int a);

}
```

```
class Bicycle implements Vehicle{

    int speed;
    int gear;

     // to change gear
    @Override
    public void changeGear(int newGear){

        gear = newGear;
    }

    // to increase speed
    @Override
    public void speedUp(int increment){

        speed = speed + increment;
    }

    // to decrease speed
    @Override
    public void applyBrakes(int decrement){

        speed = speed - decrement;
    }

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

    }

}


class Bike implements Vehicle {


    int speed;

    int gear;


    // to change gear

    @Override

    public void changeGear(int newGear){


        gear = newGear;

    }


    // to increase speed

    @Override

    public void speedUp(int increment){


        speed = speed + increment;

    }


    // to decrease speed

    @Override

    public void applyBrakes(int decrement){


        speed = speed - decrement;

    }
```

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


}
class GFG {


    public static void main (String[] args) {


        // creating an inatance of Bicycle
        // doing some operations
        Bicycle bicycle = new Bicycle();
        bicycle.changeGear(2);
        bicycle.speedUp(3);
        bicycle.applyBrakes(1);


        System.out.println("Bicycle present state :");
        bicycle.printStates();


        // creating instance of the bike.
        Bike bike = new Bike();
        bike.changeGear(1);
        bike.speedUp(4);
        bike.applyBrakes(3);


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

}
```

Output;

```
Bicycle present state :
speed: 2 gear: 2
Bike present state :
speed: 1 gear: 1
```

New features added in interfaces in JDK 8

1. Prior to JDK 8, interface could not define implementation. We can now add default implementation for interface methods. This default implementation has special use and does not affect the intention behind interfaces.

   Suppose we need to add a new function in an existing interface. Obviously the old code will not work as the classes have not implemented those new functions. So with the help of default implementation, we will give a default body for the newly added functions. Then the old codes will still work.

   filter_none

   edit

   play_arrow

   brightness_4

```
// An example to show that interfaces can

// have methods from JDK 1.8 onwards

interface In1

{

    final int a = 10;

    default void display()

    {

        System.out.println("hello");

    }

}
```

```
// A class that implements the interface.

class TestClass implements In1

{

    // Driver Code

    public static void main (String[] args)

    {

        TestClass t = new TestClass();

        t.display();

    }

}
```

Output :

```
hello
```

2. Another feature that was added in JDK 8 is that we can now define static methods in interfaces which can be called independently without an object. Note: these methods are not inherited.

   filter_none

   edit

   play_arrow

   brightness_4

```
// An example to show that interfaces can

// have methods from JDK 1.8 onwards

interface In1

{

    final int a = 10;

    static void display()

    {

        System.out.println("hello");

    }

}
```

```
// A class that implements the interface.
class TestClass implements In1
{
    // Driver Code
    public static void main (String[] args)
    {
        In1.display();
    }
}
```

Output :

```
hello
```

Important points about interface or summary of article:

- We can’t create instance(interface can’t be instantiated) of interface but we can make reference of it that refers to the Object of its implementing class.
- A class can implement more than one interface.
- An interface can extends another interface or interfaces (more than one interface) .
- A class that implements interface must implements all the methods in interface.
- All the methods are public and abstract. And all the fields are public, static, and final.
- It is used to achieve multiple inheritance.
- It is used to achieve loose coupling.

Method Overriding in Java

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

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

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.

3. There must be an IS-A relationship (inheritance).

Example of method overriding

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

```
//Java Program to illustrate the use of Java Method Overriding
//Creating a parent class.
class Vehicle{
  //defining a method
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike2 extends Vehicle{
  //defining the same method as in the parent class
  void run(){System.out.println("Bike is running safely");}

  public static void main(String args[]){
  Bike2 obj = new Bike2();//creating object
  obj.run();//calling method
  }
}
```

Output:

```
Bike is running safely
```