INHERITANCE

Inheritance in Java **is a mechanism in which one object acquires all the properties and behaviors of a parent object.**

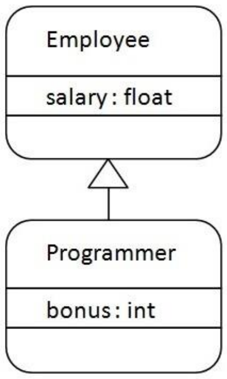
The syntax of Java Inheritance

```
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);
}
}
```

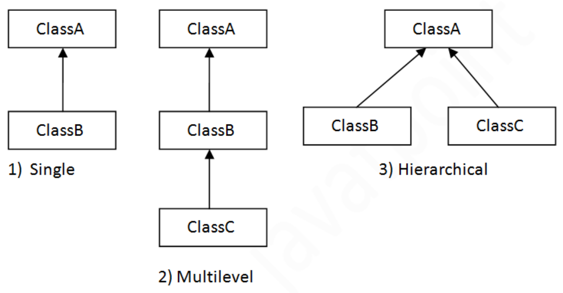
Output:

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

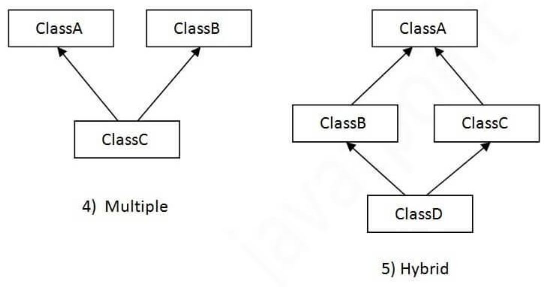
Types of inheritance in java

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.



When one class inherits multiple classes, it is known as multiple inheritance. For Example:



Single Inheritance

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.

```
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

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.

```
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

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.

```
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...
```

Hybrid Inheritance

Lets write this in a program to understand how this works:

```
          C
          ↑
          |
  ---------------
  ↑             ↑
  |             |
  A             B
  ↑
  |
  D
```

Program: This example is just to demonstrate the hybrid inheritance in Java. Although this example is meaningless, you would be able to see that how we have implemented two types of inheritance(single and hierarchical) together to form hybrid inheritance.

Class A and B extends class C → Hierarchical inheritance
Class D extends class A → Single inheritance

```
class C
{
  public void disp()
  {
          System.out.println("C");
  }
}

class A extends C
{
  public void disp()
  {
          System.out.println("A");
  }
}
class B extends C
{
  public void disp()
  {
          System.out.println("B");
  }

}
class D extends A
{
  public void disp()
  {
          System.out.println("D");
  }
  public static void main(String args[]){
          D obj = new D();
          obj.disp();
  }
}
```

Output:

```
D
```

Method Overriding

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

- The method must have the same name as in the parent class
- The method must have the same parameter as in the parent class.
- There must be an IS-A relationship (inheritance).

Example of method overriding

```
class Animal {
  public void move() {
    System.out.println("Animals can move");
  }
}

class Dog extends Animal {
  public void move() {
    System.out.println("Dogs can walk and run");
  }
}

public class TestDog {

  public static void main(String args[]) {
    Animal a = new Animal();   // Animal reference and object
    Animal b = new Dog();   // Animal reference but Dog object

    a.move();   // runs the method in Animal class
    b.move();   // runs the method in Dog class
  }
}
```

Output

```
Animals can move
Dogs can walk and run
```

In the above example, you can see that even though b is a type of Animal it runs the move method in the Dog class. The reason for this is: In compile time, the check is made on the reference type.

However, in the runtime, JVM figures out the object type and would run the method that belongs to that particular object.
Therefore, in the above example, the program will compile properly since Animal class has the method move. Then, at the runtime, it runs the method specific for that object.

Super Keyword

The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

1) super is used to refer immediate parent class instance variable.

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

```
class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
}
```

```
class TestSuper1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}}
```

Output:

```
black
white
```

2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```
class Animal{

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

```
Dog d=new Dog();
d.work();
}}
```

Output:

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

Final Keyword

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable.

1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
class Bike9
{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[])
{
 Bike9 obj=new  Bike9();
 obj.run();
 }
}//end of class
```

Output:

Compile Time Error

2) Java final class

If you make any class as final, you cannot extend it.

Example of final class

```
final class Bike{}
 class Honda1 extends Bike{
 void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[]){
  Honda1 honda= new Honda1();
  honda.run();
  }
}
```

Output:

Compile Time Error

Exception Handling

The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that the normal flow of the application can be maintained.

Java Exception Keywords

Keyword	Description
Try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
Catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.

throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

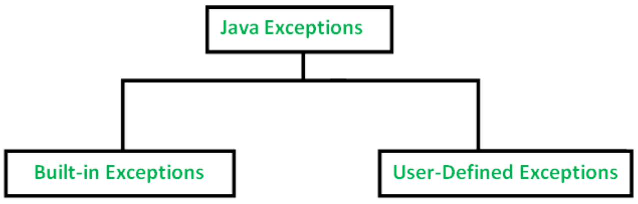
Java Exception Handling Example

```
public class JavaExceptionExample{
 public static void main(String args[]){
 try{
     //code that may raise exception
     int data=100/0;
      }
   catch(ArithmeticException e)
{
 System.out.println(e);
}
   //rest code of the program
   System.out.println("rest of the code...");
  }
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

Types of Exception



Built-in Exceptions:

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations. Below is the list of important built-in exceptions in Java.

1. **ArithmeticException:** It is thrown when an exceptional condition has occurred in an arithmetic operation.
2. **ArrayIndexOutOfBoundsException:** It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.

3. **ClassNotFoundException:** This Exception is raised when we try to access a class whose definition is not found
4. **FileNotFoundException:** This Exception is raised when a file is not accessible or does not open.
5. **IOException:** It is thrown when an input-output operation failed or interrupted
6. **InterruptedException:** It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.
7. **NoSuchFieldException:** It is thrown when a class does not contain the field (or variable) specified
8. **NoSuchMethodException:** It is thrown when accessing a method that is not found.
9. **NullPointerException:** This exception is raised when referring to the members of a null object. Null represents nothing
10. **NumberFormatException:** This exception is raised when a method could not convert a string into a numeric format.
11. **RuntimeException:** This represents an exception that occurs during runtime.
12. **StringIndexOutOfBoundsException:** It is thrown by String class methods to indicate that an index is either negative or greater than the size of the string
13. **IllegalArgumentException :** This exception will throw the error or error statement when the method receives an argument which is not accurately fit to the given relation or condition. It comes under the unchecked exception.
14. **IllegalStateException :** This exception will throw an error or error message when the method is not accessed for the particular operation in the application. It comes under the unchecked exception.

Examples of Built-in Exception

1. Arithmetic exception

```
// Java program to demonstrate ArithmeticException
class ArithmeticException_Demo
{
   public static void main(String args[])
   {
      try {
         int a = 30, b = 0;
         int c = a/b;  // cannot divide by zero
         System.out.println ("Result = " + c);
      }
      catch(ArithmeticException e) {
         System.out.println ("Can't divide a number by 0");
      }
   }
}
```

Output

Can't divide a number by 0

2. FileNotFound Exception

```
//Java program to demonstrate FileNotFoundException
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
 class File_notFound_Demo {

    public static void main(String args[])  {
        try {

            // Following file does not exist
            File file = new File("E://file.txt");

            FileReader fr = new FileReader(file);
        } catch (FileNotFoundException e) {
           System.out.println("File does not exist");
        }
    }
}
```

Output:

File does not exist

3.ArrayIndexOutOfBounds Exception

```
// Java program to demonstrate ArrayIndexOutOfBoundException
class ArrayIndexOutOfBound_Demo
{
    public static void main(String args[])
    {
        try{
            int a[] = new int[5];
            a[6] = 9; // accessing 7th element in an array of
                      // size 5
        }
```

```
      catch(ArrayIndexOutOfBoundsException e){
         System.out.println ("Array Index is Out Of Bounds");
      }
   }
}
```

Output: Array Index is Out Of Bounds

4. IllegalArgumentException: This program, checks whether the person is eligible for voting or not. If the age is greater than or equal to 18 then it will not throw any error. If the age is less than 18 then it will throw an error with the error statement.
Also, we can specify "throw new IllegalArgumentException()" without the error message. We can also specify Integer.toString(variable_name) inside the IllegalArgumentException() and It will print the argument name which is not satisfied the given condition.

```
import java.io.*;
  class GFG {
   public static void print(int a)
    {
        if(a>=18){
         System.out.println("Eligible for Voting");
         }
         else{
              throw new IllegalArgumentException("Not Eligible for Voting");

         }
           }
    public static void main(String[] args) {
        GFG.print(14);
    }
}
```

Output :

```
Exception in thread "main" java.lang.IllegalArgumentException: Not Eligible for Voting
at GFG.print(File.java:13)
at GFG.main(File.java:19)
```

Catch Multiple Exceptions

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

Points to remember

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

Example:

```
public class MultipleCatchBlock1 {

   public static void main(String[] args) {

         try{
             int a[]=new int[5];
             a[5]=30/0;
            }
            catch(ArithmeticException e)
              {
               System.out.println("Arithmetic Exception occurs");
              }
            catch(ArrayIndexOutOfBoundsException e)
              {
               System.out.println("ArrayIndexOutOfBounds Exception occurs");
              }
            catch(Exception e)
              {
               System.out.println("Parent Exception occurs");
              }
            System.out.println("rest of the code");
   }
}
```

Output:

```
Arithmetic Exception occurs
rest of the code
```

User-Defined Exceptions

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, the user can also create exceptions which are called 'user-defined Exceptions'.

The following steps are followed for the creation of a user-defined Exception.

- The user should create an exception class as a subclass of the Exception class. Since all the exceptions are subclasses of the Exception class, the user should also make his class a subclass of it. This is done as:

```
class MyException extends Exception
```

- We can write a default constructor in his own exception class.

```
MyException(){}
```

- We can also create a parameterized constructor with a string as a parameter. We can use this to store exception details. We can call the superclass(Exception) constructor from this and send the string there.

```
MyException(String str)

{

  super(str);

}
```

- To raise an exception of a user-defined type, we need to create an object to his exception class and throw it using the throw clause, as:

```
MyException me = new MyException("Exception details");

throw me;
```

- The following program illustrates how to create your own exception class MyException.
- Details of account numbers, customer names, and balance amounts are taken in the form of three arrays.
- In main() method, the details are displayed using a for-loop. At this time, a check is done if in any account the balance amount is less than the minimum balance amount to be apt in the account.
- If it is so, then MyException is raised and a message is displayed "Balance amount is less".

Example

```
// Java program to demonstrate user defined exception  and program throws an exception whenever balance amount is below Rs 1000
class MyException extends Exception
{
   //store account information
   private static int accno[] = {1001, 1002, 1003, 1004};
   private static String name[] =  {"Nish", "Shubh", "Sush", "Abhi", "Akash"};
     private static double bal[] =   {10000.00, 12000.00, 5600.0, 999.00, 1100.55};
     // default constructor
   MyException()
 {   }
     // parameterized constructor
   MyException(String str)
   { super(str); }

   // write main()
   public static void main(String[] args)
   {
       try  {
           // display the heading for the table
           System.out.println("ACCNO" + "\t" + "CUSTOMER" +  "\t" + "BALANCE");

           // display the actual account information
           for (int i = 0; i < 5 ; i++)
           {
               System.out.println(accno[i] + "\t" + name[i] +  "\t" + bal[i]);

               // display own exception if balance < 1000
               if (bal[i] < 1000)
               {
                   MyException me = new MyException("Balance is less than 1000");
                   throw me;
               }
           }
       } //end of try
         catch (MyException e)
{
           e.printStackTrace();
       }
   } }
```

Runtime Error

```
 MyException: Balance is less than 1000
    at MyException.main(fileProperty.java:36)
```

Output:

```
ACCNO    CUSTOMER    BALANCE
1001    Nish    10000.0
1002    Shubh    12000.0
1003    Sush    5600.0
1004    Abhi    999.0
```

Static binding

When type of the object is determined at compiled time(by the compiler), it is known as static binding.

If there is any private, final or static method in a class, there is static binding.

Example of static binding

```
class Dog{
 private void eat(){System.out.println("dog is eating...");}

 public static void main(String args[]){
  Dog d1=new Dog();
  d1.eat();
 }
}
```

Dynamic binding

When type of the object is determined at run-time, it is known as dynamic binding.

Example of dynamic binding

```
class Animal{
 void eat(){System.out.println("animal is eating...");}
}

class Dog extends Animal{
 void eat(){System.out.println("dog is eating...");}

 public static void main(String args[]){
  Animal a=new Dog();
  a.eat();
 }
}
```

Output:
```
dog is eating...
```

In the above example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal.So compiler doesn't know its type, only its base type.

Sr. No.	Key	Static Binding	Dynamic Binding
1	Basic	It is resolved at compile time	It is resolved at run time
2	Resolve mechanism	static binding use type of the class and fields	Dynamic binding uses object to resolve binding
3	Example	Overloading is an example of static binding	Method overriding is the example of Dynamic binding
4.	Type of Methods	private, final and static methods and variables uses static binding	Virtual methods use dynamic binding

Abstract class
A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented.

NOTE:

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

Abstract Method

A method which is declared as abstract and does not have implementation is known as an abstract method.

Example of abstract method:
abstract void printStatus();//no method body and abstract

Example of Abstract class that has an abstract method

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

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

Output:

```
running safely
```

INTERFACE

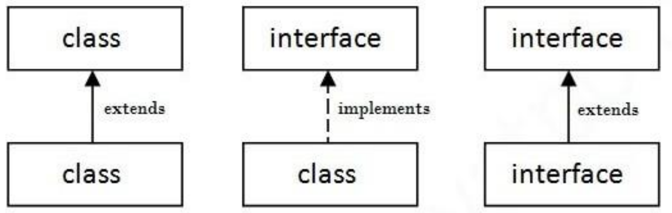
Another way to achieve abstraction in Java, is with interfaces. An interface is a completely "abstract class" that is used to group related methods with empty bodies.

There are mainly three reasons to use interface.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

The relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



Java Interface

In this example, the Printable interface has only one method, and its implementation is provided in the A6 class.

```
interface printable
{
void print();
}
class A6 implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
 }
}
```

Output:

```
Hello
```

Multiple inheritance in Java by interface

Let us look at the below program for a better understanding:

- In the below code, we have declared two abstract methods, i.e. execute1() and execute2().
- Both the methods execute1() and execute2() are written in interface A and interface B, respectively.

```
interface A
{
   public abstract void execute1();
}
interface B
{
   public abstract void execute2();
}
class C implements A,B
{
   public void execute1()
   {
      System.out.println("Haii.. I am from execute1");
   }
   public void execute2()
   {
      System.out.println("Haii.. I am from execute2");
   }
}
public class Main
{
        public static void main(String[] args)
        {
                C obj = new C(); // creating object of class C
                obj.execute1(); //calling method execute1
                obj.execute2(); // calling method execute2
        }
}
```

Output:

```
Haii.. I am from execute1
Haii.. I am from execute2
```

By seeing the above output, we can come to a **conclusion** that a **Java class can extend two interfaces (or more)**, which clearly supports it as an alternative to Multiple Inheritance in Java.