Unit IV

Inheritance, Packages and Exception Handling

UNIT IV
INHERITANCE, PACKAGES AND EXCEPTION HANDLING

Syllabus

Inheritances: member access and inheritance, super class references, Using super, multilevel hierarchy, constructor call sequence, method overriding, dynamic method dispatch, abstract classes, Object class.

Packages and Interfaces: defining a package, finding packages and CLASSPATH, access protection, importing packages, interfaces (defining, implementation, nesting, applying), variables in interfaces, extending interfaces, instance of operator. fundamental, exception types, uncaught exceptions, try, catch, throw, throws, finally, multiple catch clauses, nested try statements, built-in exceptions, custom exceptions (creating your own exception sub classes).

Managing I/O: Streams, Byte Streams and Character Streams, Predefined Streams, Reading console Input, Writing Console Output, Print Writer class.

INHERITANCE

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

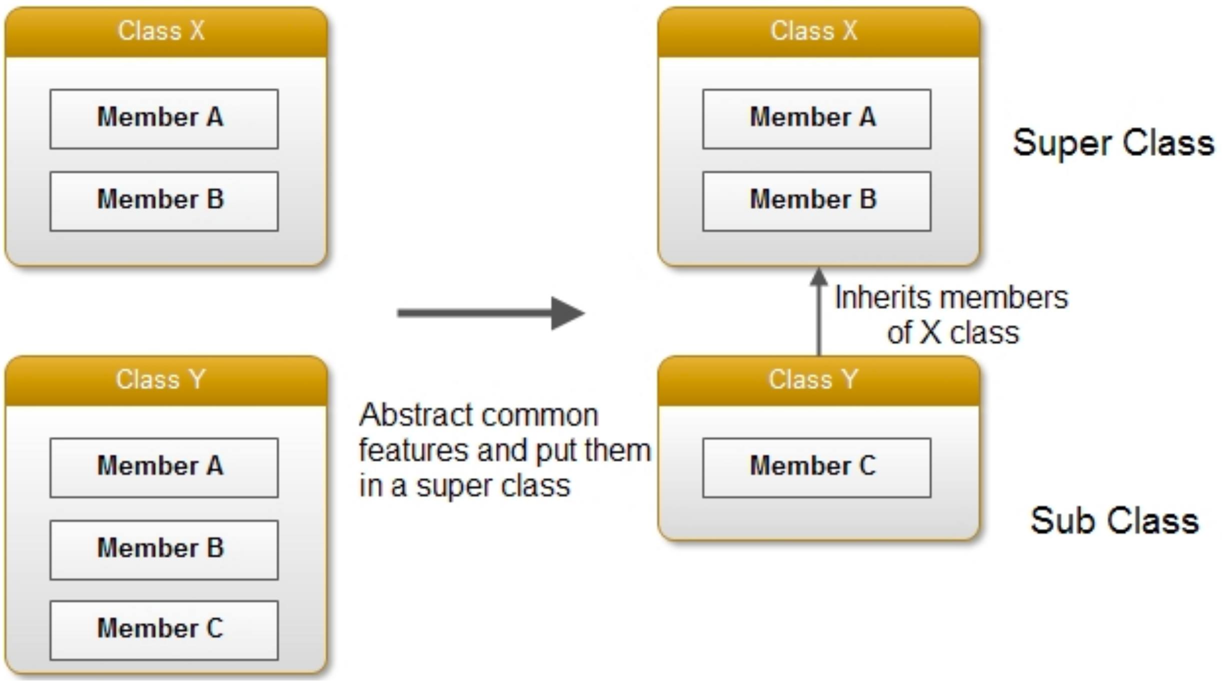
- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

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.

Concept of Inheritance



In the above diagram, Class X contains member A and B. Later on, if a Class Y is to be added in the program which contains identical members A and B to that of Class X along with an additional member C, then instead of writing the code for members A and B again in Class Y, it is better to inherit these members from an already existing Class X and only add additional member C in it as shown in the figure. Thus by inheriting the common features of the superclass into the subclass, the size of the code is reduced. It also helps to ensure consistency as common features do not exist in several classes and therefore, only need to be modified or tested once. The Class X from which the features are inherited is called the superclass and the Class Y that inherits the features is called the subclass.

In Fig, the arrow pointing in the upward direction from Class Y to Class X indicates that the Class Y is derived from the subclass Class X.

Advantages of Inheritance

- **Minimizing duplicate code:** Key benefits of Inheritance include minimizing the identical code as it allows sharing of the common code among other subclasses.
- **Flexibility:** Inheritance makes the code flexible to change, as you will adjust only in one place, and the rest of the code will work smoothly.
- **Overriding:** With the help of Inheritance, you can override the methods of the base class.
- **Data Hiding:** The base class in Inheritance decides which data to be kept private, such that the derived class will not be able to alter it.

Disadvantages of Inheritance

- **No Independence:** One of the main disadvantages of Inheritance in Java is that two classes, both the base and inherited class, get tightly bounded by each other. In simple terms, Programmers can not use these classes independently of each other.
- **Decreases Execution Speed:** Another con of Inheritance is that it decreases the execution speed because Inheritance execution takes time and effort.
- **Refactoring the Code:** If the user deletes the Super Class, then they have to refactor it if they have used

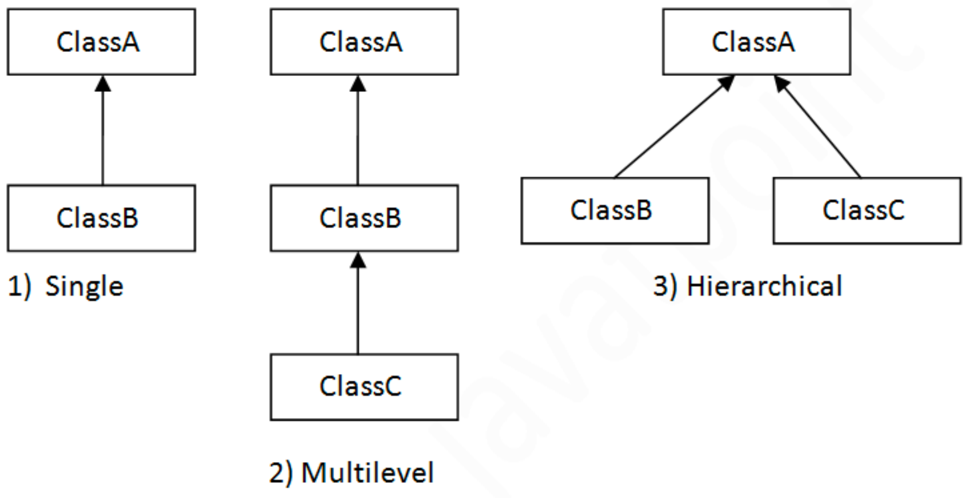
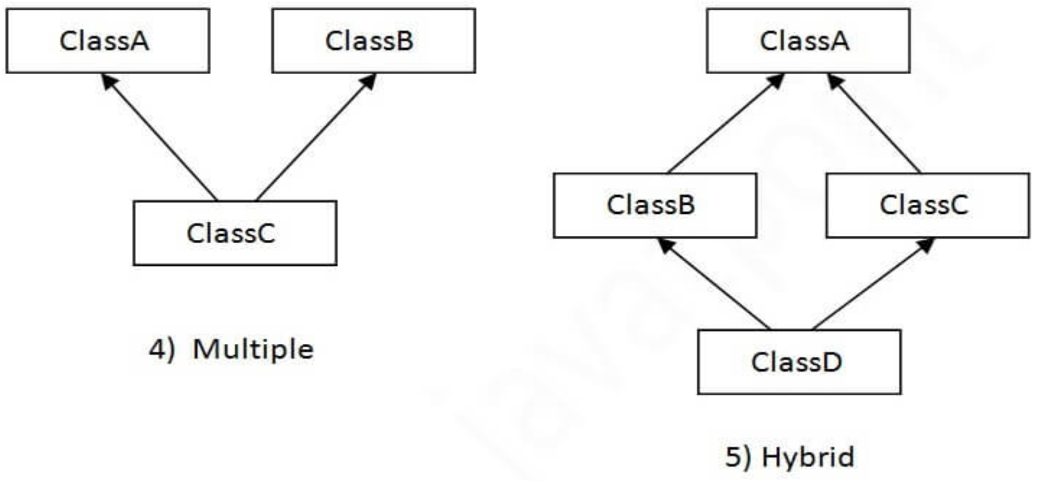
The syntax of Java Inheritance

```
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.

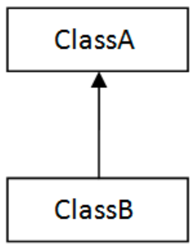
Types of Inheritance

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

Using Class
 ClassA ClassB 1) Single ClassA ClassB ClassC 2) Multilevel ClassA ClassB ClassC 3) Hierarchical
Using Interfaces
 ClassA ClassB ClassC 4) Multiple ClassA ClassB ClassC ClassD 5) Hybrid

1. **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.


1) Single

Program

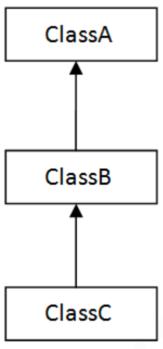
File: Main.java

```java
// Java program to illustrate the
// concept of single inheritance
import java.io.*;
import java.lang.*;
import java.util.*;
class one {
     public void print_hello()
     {
          System.out.println("Hello");
     }
}

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

Output

```
Hello
Welcome
Hello
```

2. **Multilevel Inheritance**



2) Multilevel

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.

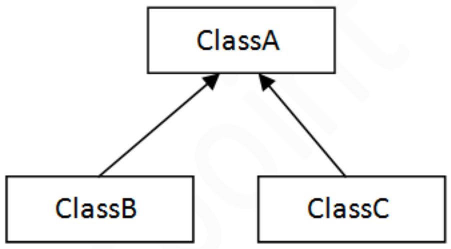
File: Main.java

```
// Java program to illustrate the
// concept of Multilevel inheritance
import java.io.*;
import java.lang.*;
import java.util.*;
class one {
      public void print_hello()
      {
            System.out.println("Hello");
      }
}
class two extends one {
      public void print_welcome() { System.out.println("Welcome"); }
}
class three extends two {
      public void print_hello ()
      {
            System.out.println("Hello");
      }
}
// Drived class
public class Main {
      public static void main(String[] args)
      {
            three g = new three();
            g.print_hello();
            g.print_welcome();
            g.print_hello();
      }
}
```

Output:

```
Hello

Welcome

Hello
```

3. **Hierarchical Inheritance**



3) Hierarchical

When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the below image, class A serves as a base class for the derived class B, C and D.

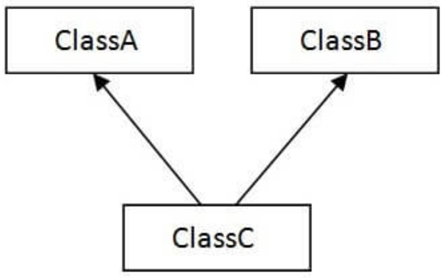
File:Test.java

```
// 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):



4) Multiple

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

File:Test.java

```
// Java program to illustrate the concept of Multiple inheritance
import java.io.*;
import java.lang.*;
import java.util.*;
interface one {
     public void print_hello();
}
interface two {
     public void print_welcome();
}
interface three extends one, two {
     public void print_hello();
}
class child implements three {
     @Override public void print_hello(){
          System.out.println("Hello");
     }
     public void print_welcome() { System.out.println("Welcome"); }
}
public class Main {
     public static void main(String[] args)
     {
          child c = new child();
          c.print_hello();
          c.print_welcome();
          c.print_hello();
     }
}
```

Output:

```
Hello
Welcome
Hello
```

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 Test class inherits Parent1 and Parent2 classes. If Parent1 and Parent2 classes have the same method and you call it from child class object, there will be ambiguity to call the method of Parent1 or Parent2 class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

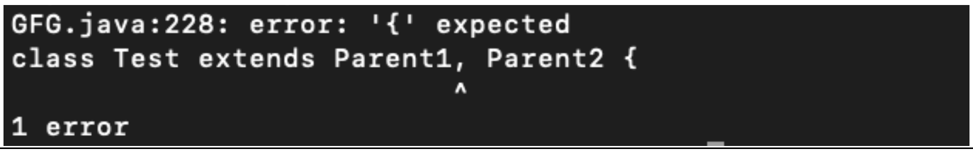
Using Class, Java doen not support multiple inheritance.

File:Test.java

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

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

  class Parent1{
  void msg(){System.out.println("Hello");}
  }
  class Parent2{
  void msg(){System.out.println("Welcome");}
  }
```

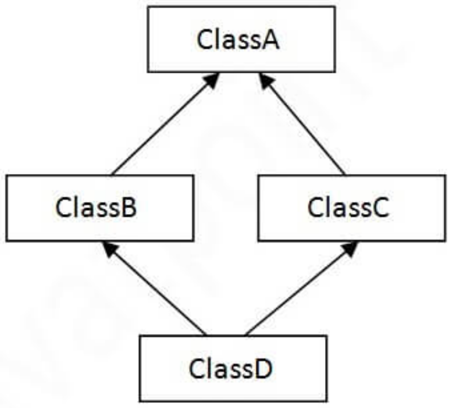
```
   class Test extends Parent1, Parent2{//suppose if it were

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

Output:



```
GFG.java:228: error: '{' expected
class Test extends Parent1, Parent2 {
                          ^
1 error
```

5. **Hybrid Inheritance (Through Interfaces):**



5) Hybrid

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. It is the combination of two or more types of inheritance.

Member access and inheritance

Superclass members can be inherited to subclass provided they are eligible by access modifiers. The behavior of access specifiers in the case of inheritance in java is as follows:

1. **The private members of the superclass** cannot be inherited to the subclass because the private members of superclass are not available to the subclass directly. They are only available in their own class.
2. **The default members of the parent class** can be inherited to the derived class within the same package.

3. **The protected members of a parent class** can be inherited to a derived class but the usage of protected members is limited within the package.
4. **Public members** can be inherited to all subclasses.

1. The private members of the superclass

Let's create a program where we will understand private members of superclass are not accessible in subclass but protected members are available in subclass.

```
package inheritance;
class Baseclass
{
 private int x = 30;
 protected int y = 50;
 private void m1()
 {
    System.out.println("Base class m1 method");
 }
protected void m2()
{
   System.out.println("Base class m2 method");
 }
}
class Derivedclass extends Baseclass
{

 }
public class MainClass
{
 public static void main(String[] args)
 {
   Derivedclass d = new Derivedclass(); // Private members cannot be
accessed due to not available in subclass.
     d.m2();
   System.out.println("y = " +d.y);
  }
 }
```

Output:

```
       Base class m2 method
       y = 50
```

When we call m1() method using subclass reference variable, m1 is not available to call. But m2() method is available. Similarly, private variable x is also not available to subclass

2. The default and public members of the parent class

Let us take another example program to understand default and public members of superclass that are accessible in the subclass.

```
package inheritance;
public class Identity
{
   String name = "Deep";
   public int age = 28;
void m1()
{
  System.out.println("Name: " +name);
 }
public void m2()
{
  System.out.println("Age: " +age);
 }
}
public class Person extends Identity
{

 }
public class Mytest
{
 public static void main(String[] args)
 {
   Person p = new Person();
     p.m1();
     p.m2();
    System.out.println("Name: " +p.name);
    System.out.println("Age: " +p.age);
  }
 }
```

Output:

```
        Name: Deep

        Age: 28

        Name: Deep

        Age: 28
```

As you can see in the above program, default and public members can be easily accessible in the subclass within the same package.

3. **Protected members of a parent class**

The protected members of a parent class can be inherited to a derived class but the usage of protected members is limited within the package

```
class Employee
{
    protected int id = 101;
    protected String name = "Jack";
}
public class ProtectedDemo extends Employee
{
    private String dept = "Networking";
    public void display()
    {
        System.out.println("Employee Id : "+id);
        System.out.println("Employee name : "+name);
        System.out.println("Employee Department : "+dept);
    }
    public static void main(String args[])
    {
        ProtectedDemo pd = new ProtectedDemo();
        pd.display();
    }
}
```

Output:

```
Employee Id : 101
Employee name : Jack
Employee Department : Networking
```

Super Class Reference

A reference variable of a superclass can be used to a refer any subclass object derived from that superclass. If the methods are present in SuperClass, but overridden by SubClass, it will be the overridden method that will be executed. The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

For example :

```
class Animal{
Animal()
{System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");
}
}
public class TestSuper3{
public static void main(String args[]){
Dog d=new Dog();
}}
```

Output:

```
animal is created
dog is created
```

Advantage : We can use superclass reference to hold any subclass object derived from it.

Disadvantage : By using superclass reference, we will have access **only** to those parts(methods and variables) of the object defined by the superclass.

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

Output:

```
black
white
```

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

1. **super can be used to invoke immediate 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...
```

In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.

To call the parent class method, we need to use super keyword.

2. **super is used to invoke parent class constructor.**

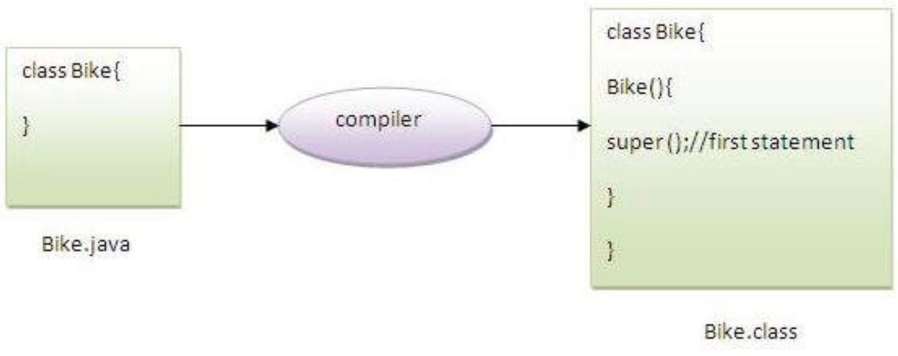
The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

```
  class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");
}
}
public class TestSuper3{
public static void main(String args[]){
Dog d=new Dog();
}}
```

Output:

```
animal is created
dog is created
```

Note: super() is added in each class constructor automatically by compiler if there is no super() or this().



As we know well that default constructor is provided by compiler automatically if there is no constructor. But, it also adds super() as the first statement.

Another example of super keyword where super() is provided by the compiler implicitly.

```
  class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
System.out.println("dog is created");
}
}
public class TestSuper4{
public static void main(String args[]){
Dog d=new Dog();
}}
```

Output:

```
animal is created
dog is created
```

Constructor call sequence or Order Constructor call in inheritance

Constructors in Java

A constructor in Java is similar to a method with a few differences. Constructor has the same name as the class name. A constructor doesn't have a return type.

A Java program will automatically create a constructor if it is not already defined in the program. It is executed when an instance of the class is created.

A constructor cannot be static, abstract, final or synchronized. It cannot be overridden.

Java has two types of constructors:

1. Default constructor

2. Parameterized constructor

While implementing inheritance in a Java program, every class has its own constructor. Therefore the execution of the constructors starts after the object initialization. It follows a certain sequence according to the class hierarchy. There can be different orders of execution depending on the type of inheritance.

Different ways of the order of constructor execution in Java

1. Order of execution of constructor in Single inheritance

In single level inheritance, the constructor of the base class is executed first.

```
class ParentClass   {
    ParentClass()  {
        System.out.println("ParentClass constructor executed.");
    }
}

class ChildClass extends ParentClass{
    ChildClass()   {
        System.out.println("ChildClass constructor executed.");
    }
}

public class OrderofExecution1  {
    public static void main(String ar[])
    {
        System.out.println("Order of constructor execution...");
        new ChildClass();
    }
}
```

Output:

```
Order of constructor execution...
ParentClass constructor executed.
ChildClass constructor executed.
```

In the above code, after creating an instance of ***ChildClass*** the ***ParentClass*** constructor is invoked first and then the ***ChildClass.***

2. Order of execution of constructor in Multilevel inheritance

In multilevel inheritance, all the upper class constructors are executed when an instance of bottom most child class is created.

```
class College
{
    /* Constructor */
    College()
    {
        System.out.println("College constructor executed");
    }
}

class Department extends College
{
    /* Constructor */
    Department()
    {
        System.out.println("Department constructor executed");
    }
}

class Student extends Department
{
    /* Constructor */
    Student()
    {
    System.out.println("Student constructor executed");
    }
}
public class OrderofExecution2
{
        /* Driver Code */
    public static void main(String ar[])
    {
        /* Create instance of Student class */
        System.out.println("Order   of   constructor   execution   in
Multilevel inheritance...");
        new Student();
    }
}
```

Output:

```
Order of constructor execution in Multilevel inheritance...
College constructor executed
Department constructor executed
Student constructor executed
```

In the above code, an instance of Student class is created and it invokes the constructors of College, Department and Student accordingly.

3. Calling same class constructor using this keyword

Here, inheritance is not implemented. But there can be multiple constructors of a single class and those constructors can be accessed using this keyword.

```
  public class OrderofExecution3
{
    /* Default constructor */
    OrderofExecution3()
    {
        this("CallParam");
        System.out.println("Default constructor executed.");
    }
    /* Parameterized constructor */
    OrderofExecution3(String str)
    {
        System.out.println("Parameterized constructor executed.");
    }
    /* Driver Code */
    public static void main(String ar[])
    {
        /* Create instance of the class */
        System.out.println("Order of constructor execution...");
        OrderofExecution3 obj = new OrderofExecution3();
    }
}
```

Output:

```
Order of constructor execution...
Parameterized constructor executed.
Default constructor executed.
```

In the above code, the parameterized constructor is called first even when the default constructor is called while object creation. It happens because ***this*** keyword is used as the first line of the default constructor.

4. Calling superclass constructor using super keyword

A child class constructor or method can access the base class constructor or method using the super keyword.

```
  /* Parent Class */
class ParentClass
{
    int a;
    ParentClass(int x)
    {
        a = x;
    }
}
/* Child Class */
class ChildClass extends ParentClass
{
    int b;
    ChildClass(int x, int y)
    {
        /* Accessing ParentClass Constructor */
        super(x);
        b = y;
    }
    /* Method to show value of a and b */
    void Show()
    {
        System.out.println("Value of a : "+a+"\nValue of b : "+b);
    }
}

public class OrderofExecution4
{
    /* Driver Code */
    public static void main(String ar[])
    {
        System.out.println("Order of constructor execution...");
        ChildClass d = new ChildClass(79, 89);
        d.Show();
    }
}
```

Output:

```
Order of constructor execution...
Value of a : 79
Value of b : 89
```

In the above code, the ***ChildClass*** calls the ***ParentClass*** constructor using a ***super*** keyword that determines the order of execution of constructors.

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).

Program without over riding	**Program with over riding**

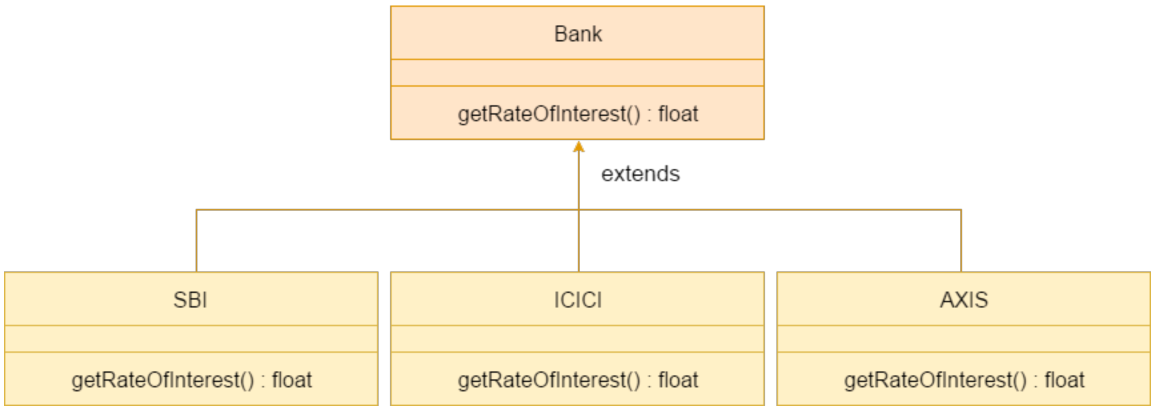
`class Vehicle{`
`void run()`
`{System.out.println("Vehicle is running");}`
`}`
`//Creating a child class`
`public class Bike extends Vehicle{`
`  public static void main(String args[]){`
`  Bike obj = new Bike();`
`  obj.run();`
`  }`
`}`
	`class Vehicle{`
	`  `**`void run()`**
	`{System.out.println("Vehicle is running");}`
	`}`
	`//Creating a child class`
	`public class Bike2 extends Vehicle{`
	`  `**`void run()`**
	`{System.out.println("Bike is running safely");}`
	`  public static void main(String args[]){`
	`  Bike2 obj = new Bike2();`
	`  obj.run();//calling method`
	`  }`
	`}`

Output: `Vehicle is running`	Output: `Bike is running safely`

Case Study : banking Application

A real example of Java Inheritance, Method Overriding

Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7%, and 9% rate of interest.


Bank
getRateOfInterest() : float
extends
SBI
getRateOfInterest() : float
ICICI
getRateOfInterest() : float
AXIS
getRateOfInterest() : float

```
//Creating a parent class.
class Bank{
int getRateOfInterest(){return 0;}
}
//Creating child classes.
class SBI extends Bank{
int getRateOfInterest(){return 8;}
}

class ICICI extends Bank{
int getRateOfInterest(){return 7;}
}
class AXIS extends Bank{
int getRateOfInterest(){return 9;}
}
//Test class to create objects and call the methods
public class Test2{
public static void main(String args[]){
SBI s=new SBI();
ICICI i=new ICICI();
AXIS a=new AXIS();
System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
System.out.println("ICICI Rate of Interest: 
"+i.getRateOfInterest());
System.out.println("AXIS Rate of Interest: 
"+a.getRateOfInterest());
}
}
```

Output:

```
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9
```

Dynamic method dispatch

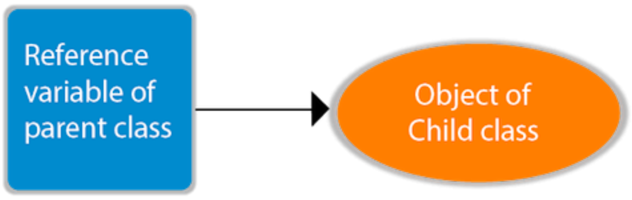
Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Let's first understand the upcasting before Runtime Polymorphism.

Upcasting

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:



```
1. class A{}
2. class B extends A{}
1. A a=new B();//upcasting
```

For upcasting, we can use the reference variable of class type or an interface type. For Example:

```
1. interface I{}
2. class A{}
3. class B extends A implements I{}
```

Here, the relationship of B class would be:

```
B IS-A A
B IS-A I
B IS-A Object
```

Since Object is the root class of all classes in Java, so we can write B IS-A Object.

Example of Java Runtime Polymorphism

In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

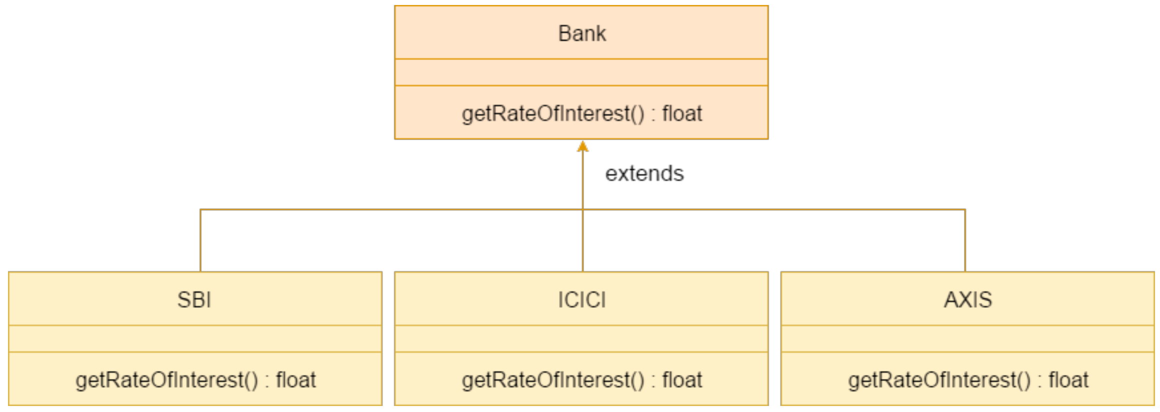
```
//Creating a parent class.
class Bank{
int getRateOfInterest(){return 0;}
}
//Creating child classes.
class SBI extends Bank{
int getRateOfInterest(){return 8;}
}

class Bike{
  void run(){System.out.println("running");}
}
public class Splendor extends Bike{
  void run(){System.out.println("running safely with 60km");}

  public static void main(String args[]){
    Bike b = new Splendor();//upcasting
    b.run();
  }
}
```

Output:

```
running safely with 60km
```

Java Runtime Polymorphism Example: Bank

Consider a scenario where Bank is a class that provides a method to get the rate of interest. However, the rate of interest may differ according to banks. For example, SBI, ICICI, and AXIS banks are providing 8.4%, 7.3%, and 9.7% rate of interest.


Bank
getRateOfInterest() : float
extends
SBI
getRateOfInterest() : float
ICICI
getRateOfInterest() : float
AXIS
getRateOfInterest() : float

```
class Bank{
float getRateOfInterest(){return 0;}
}
class SBI extends Bank{
float getRateOfInterest(){return 8.4f;}
}
class ICICI extends Bank{
float getRateOfInterest(){return 7.3f;}
}
class AXIS extends Bank{
float getRateOfInterest(){return 9.7f;}
}
public class TestPolymorphism{
public static void main(String args[]){
Bank b;
b=new SBI();
System.out.println("SBI Rate of Interest: "+b.getRateOfInterest(
));
b=new ICICI();
System.out.println("ICICI Rate of Interest: "+b.getRateOfInteres
t());
b=new AXIS();
System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest
());
}
}
```

Output:

```
SBI Rate of Interest: 8.4
 ICICI Rate of Interest: 7.3
 AXIS Rate of Interest: 9.7
```

Abstract class in Java

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

Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.

- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

Example of abstract class

1. **abstract class** A{}

Abstract Method in Java

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

Example of abstract method

1. **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();
}
public 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
```

Understanding the real scenario of Abstract class

Mostly, we don't know about the implementation class (which is hidden to the end user), and an object of the implementation class is provided by the **factory method**.

A **factory method** is a method that returns the instance of the class.

```
abstract class Bank{
abstract int getRateOfInterest();
}
class SBI extends Bank{
```

```
int getRateOfInterest(){return 7;}
}
class PNB extends Bank{
int getRateOfInterest(){return 8;}
}
public class TestBank{
public static void main(String args[]){
Bank b;
b=new SBI();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()
+" %");
b=new PNB();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()
+" %");
}}
```

Output:

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

Abstract class having constructor, data member and methods

An abstract class can have a data member, abstract method, method body (non-abstract method), constructor, and even main() method.

```
//Example of an abstract class that has abstract and non-
abstract methods
 abstract class Bike{
   Bike(){System.out.println("bike is created");}
   abstract void run();
   void changeGear(){System.out.println("gear changed");}
 }
//Creating a Child class which inherits Abstract class
 class Honda extends Bike{
 void run(){System.out.println("running safely..");}
 }
//Creating a Test class which calls abstract and non-
abstract methods
 public class TestAbstraction2{
 public static void main(String args[]){
  Bike obj = new Honda();
  obj.run();
  obj.changeGear();
 }
}
```

Output:

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

Rule: If there is an abstract method in a class, that class must be abstract.

Rule: If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract.

Object class in Java

The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.

The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, know as upcasting.

Let's take an example, there is getObject() method that returns an object but it can be of any type like Employee,Student etc, we can use Object class reference to refer that object. For example:

1. Object obj=getObject();//we don't know what object will be returned from this metho
   d

The Object class provides some common behaviors to all the objects such as object can be compared, object can be cloned, object can be notified etc.

Methods of Object class

The Object class provides many methods. They are as follows:

Method	**Description**
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.

protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout)throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout,int nanos)throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait()throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize()throws Throwable	is invoked by the garbage collector before object is being garbage collected.

toString(): The toString() provides a String representation of an object and is used to convert an object to String. The default toString() method for class Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@', and the unsigned hexadecimal representation of the hash code of the object. In other words, it is defined as:

```
// Default behavior of toString() is to print class name, then
// @, then unsigned hexadecimal representation of the hash code
// of the object

public String toString()
{
   return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

It is always recommended to override the **toString()** method to get our own String representation of Object.

Packages and Interfaces:

A **java package** is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

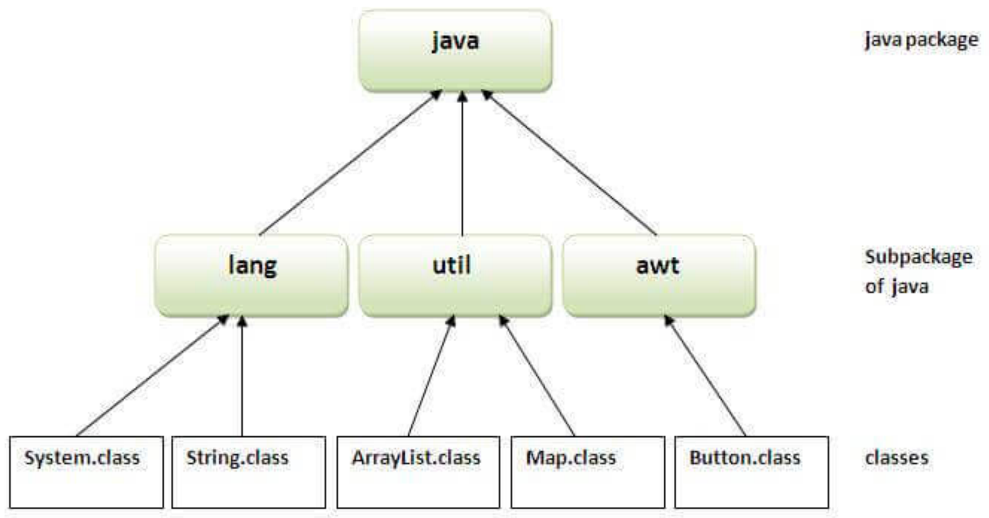
Here, we will have the detailed learning of creating and using user-defined packages.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.



Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc.

A **Package** can be defined as a grouping of related types (classes, interfaces, enumerations and annotations) providing access protection and namespace management.

Some of the existing packages in Java are −

- **java.lang** − bundles the fundamental classes
- **java.io** − classes for input , output functions are bundled in this package

Programmers can define their own packages to bundle group of classes/interfaces, etc. It is a good practice to group related classes implemented by you so that a programmer can easily determine that the classes, interfaces, enumerations, and annotations are related.

Since the package creates a new namespace there won't be any name conflicts with names in other packages. Using packages, it is easier to provide access control and it is also easier to locate the related classes.

Creating a Package

While creating a package, you should choose a name for the package and include a **package** statement along with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.

The package statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

If a package statement is not used then the class, interfaces, enumerations, and annotation types will be placed in the current default package.

To compile the Java programs with package statements, you have to use -d option as shown below.

```
javac -d Destination_folder file_name.java
```

Then a folder with the given package name is created in the specified destination, and the compiled class files will be placed in that folder.

Example

Let us look at an example that creates a package called **animals**. It is a good practice to use names of packages with lower case letters to avoid any conflicts with the names of classes and interfaces.

Following package example contains interface named *animals* −

```
/* File name : Animal.java */
package animals;

interface Animal {
   public void eat();
   public void travel();
}
```

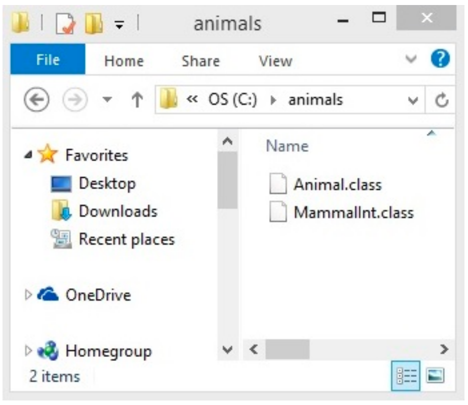
Now, let us implement the above interface in the same package *animals* −

```
   package animals;
   /* File name : MammalInt.java */
   public class MammalInt implements Animal {
      public void eat() {
         System.out.println("Mammal eats");
      }

      public void travel() {
         System.out.println("Mammal travels");
      }
      public int noOfLegs() {
         return 0;
      }
      public static void main(String args[]) {
         MammalInt m = new MammalInt();
         m.eat();
         m.travel();
      }
   }
```

Now compile the java files as shown below −

```
   $ javac -d . Animal.java
   $ javac -d . MammalInt.java
```

Now a package/folder with the name **animals** will be created in the current directory and these class files will be placed in it as shown below.



Animal.class
MammalInt.class

You can execute the class file within the package and get the result as shown below.

```
   Mammal eats
   Mammal travels
```

Importing Packages

The import Keyword

If a class wants to use another class in the same package, the package name need not be used. Classes in the same package find each other without any special syntax.

Example

Here, a class named Boss is added to the payroll package that already contains Employee. The Boss can then refer to the Employee class without using the payroll prefix, as demonstrated by the following Boss class.

```
package payroll;
public class Boss {
   public void payEmployee(Employee e) {
      e.mailCheck();
   }
}
```

What happens if the Employee class is not in the payroll package? The Boss class must then use one of the following techniques for referring to a class in a different package.

- The fully qualified name of the class can be used. For example −

```
payroll.Employee
```

- The package can be imported using the import keyword and the wild card (*). For example −

```
import payroll.*;
```

- The class itself can be imported using the import keyword. For example −

```
import payroll.Employee;
```

Note − A class file can contain any number of import statements. The import statements must appear after the package statement and before the class declaration.

The Directory Structure of Packages

Two major results occur when a class is placed in a package −

- The name of the package becomes a part of the name of the class, as we just discussed in the previous section.
- The name of the package must match the directory structure where the corresponding bytecode resides.

Here is simple way of managing your files in Java −

Put the source code for a class, interface, enumeration, or annotation type in a text file whose name is the simple name of the type and whose extension is **.java**.

For example −

```
// File Name :  Car.java
package vehicle;

public class Car {
   // Class implementation.
}
```

Now, put the source file in a directory whose name reflects the name of the package to which the class belongs −

```
....\vehicle\Car.java
```

Now, the qualified class name and pathname would be as follows −

- Class name → vehicle.Car
- Path name → vehicle\Car.java (in windows)

In general, a company uses its reversed Internet domain name for its package names.

Example − A company's Internet domain name is apple.com, then all its package names would start with com.apple. Each component of the package name corresponds to a subdirectory.

Example − The company had a com.apple.computers package that contained a Dell.java source file, it would be contained in a series of subdirectories like this −

```
....\com\apple\computers\Dell.java
```

At the time of compilation, the compiler creates a different output file for each class, interface and enumeration defined in it. The base name of the output file is the name of the type, and its extension is **.class**.

For example −

```
// File Name: Dell.java
package com.apple.computers;

public class Dell {
}

class Ups {
}
```

Now, compile this file as follows using -d option −

```
$javac -d . Dell.java
```

The files will be compiled as follows −

```
.\com\apple\computers\Dell.class
.\com\apple\computers\Ups.class
```

You can import all the classes or interfaces defined in *\com\apple\computers\* as follows −

```
import com.apple.computers.*;
```

Like the .java source files, the compiled .class files should be in a series of directories that reflect the package name. However, the path to the .class files does not have to be the same as the path to the .java source files. You can arrange your source and class directories separately, as −

```
<path-one>\sources\com\apple\computers\Dell.java
<path-two>\classes\com\apple\computers\Dell.class
```

By doing this, it is possible to give access to the classes directory to other programmers without revealing your sources. You also need to manage source and class files in this manner so that the compiler and the Java Virtual Machine (JVM) can find all the types your program uses.

The full path to the classes directory, <path-two>\classes, is called the class path, and is set with the CLASSPATH system variable. Both the compiler and the JVM construct the path to your .class files by adding the package name to the class path.

Say <path-two>\classes is the class path, and the package name is com.apple.computers, then the compiler and JVM will look for .class files in <path-two>\classes\com\apple\computers.

A class path may include several paths. Multiple paths should be separated by a semicolon (Windows) or colon (Unix). By default, the compiler and the JVM search the current directory and the JAR file containing the Java platform classes so that these directories are automatically in the class path.

Set CLASSPATH System Variable

To display the current CLASSPATH variable, use the following commands in Windows and UNIX (Bourne shell) −

```
In Windows → C:\> set CLASSPATH
In UNIX → % echo $CLASSPATH
```

To delete the current contents of the CLASSPATH variable, use −

```
In Windows → C:\> set CLASSPATH =
In UNIX → % unset CLASSPATH; export CLASSPATH
```

To set the CLASSPATH variable −

```
In Windows → set CLASSPATH = C:\users\jack\java\classes
In UNIX → % CLASSPATH = /home/jack/java/classes; export CLASSPATH
```

Access Protection In Packages

In java, the access modifiers define the accessibility of the class and its members. For example, private members are accessible within the same class members only. Java has four access modifiers, and they are default, private, protected, and public.

In java, the package is a container of classes, sub-classes, interfaces, and sub-packages. The class acts as a container of data and methods. So, the access modifier decides the accessibility of class members across the different packages.

In java, the accessibility of the members of a class or interface depends on its access specifiers. The following table provides information about the visibility of both data members and methods.

Packages are meant for encapsulating, it works as containers for classes and other subpackages. Class acts as containers for data and methods. There are four categories, provided by Java regarding the visibility of the class members between classes and packages:

- Subclasses in the same package
- Non-subclasses in the same package
- Subclasses in different packages
- Classes that are neither in the same package nor subclasses

The three main access modifiers private, public and protected provides a range of ways to access required by these categories.

Simply remember, private cannot be seen outside of its class, public can be access from anywhere, and protected can be accessible in subclass only in the hierarchy.

A class can have only two access modifier, one is default and another is public. If the class has default access then it can only be accessed within the same package by any other code. But if the class has public access then it can be access from any where by any other code.

Example

```
package pckg1;
public class PCKG1_ClassOne{
  int a = 1;
  private int pri_a = 2;
  protected int pro_a = 3;
  public int pub_a = 4;
  public PCKG1_ClassOne() {
    System.out.println("base class constructor called");
    System.out.println("a = " + a);
    System.out.println("pri_a = " + pri_a);
    System.out.println("pro_a "+ pro_a);
    System.out.println("pub_a "+ pub_a);
  }
}
```

The above file PCKG1_ClassOne belongs to package pckg1, and contains data members with all access modifiers.

Interfaces

- An interface in Java is a blueprint of a class. It has static constants and abstract methods.
- **The interface** in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.
- In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.
- Java Interface also represents the IS-A relationship.
- It cannot be instantiated just like the abstract class.
- Since Java 8, we can have default and static methods in an interface.
- Since Java 9, we can have private methods in an interface.

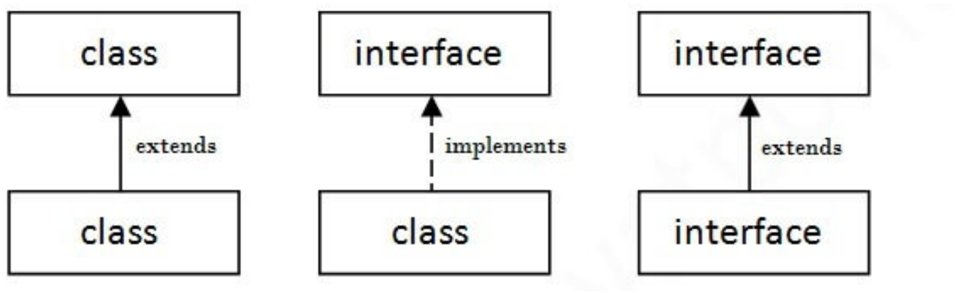
There are mainly three reasons to use interface. They are given below.

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

How to declare an interface?
An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.
Syntax:

```
interface <interface_name>{
    // declare constant fields
    // declare methods that abstract
    // by default.
}
```

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.
The relationship between class and interface



Java Interface Example

In this example, java interface which provides the implementation of Bank interface.

```
interface Bank{
float rateOfInterest();
}
class SBI implements Bank{
public float rateOfInterest(){return 9.15f;}
}
class PNB implements Bank{
public float rateOfInterest(){return 9.7f;}
}
class TestInterface2{
public static void main(String[] args){
Bank b=new SBI();
System.out.println("ROI: "+b.rateOfInterest());
}}
```

```
Output
ROI: 9.15
```

Multiple inheritance is not supported through class in java, but it is possible by an interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in the case of class because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class.

Nested Interface

An interface, i.e., declared within another interface or class, is known as a nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred to by the outer interface or class. It can't be accessed directly.

Points to remember for nested interfaces

There are given some points that should be remembered by the java programmer.

- The nested interface must be public if it is declared inside the interface, but it can have any access modifier if declared within the class.
- Nested interfaces are declared static

Syntax of nested interface which is declared within the interface

```
interface interface_name{
 ...
 interface nested_interface_name{
  ...
 }
}
```

Syntax of nested interface which is declared within the class

```
class class_name{
 ...
 interface nested_interface_name{
  ...
 }
}
```

Example :

```
interface Showable{
  void show();
  interface Message{
   void msg();
  }
}
class TestNestedInterface1 implements Showable.Message{
 public void msg(){System.out.println("Hello nested interface");}

 public static void main(String args[]){
  Showable.Message message=new TestNestedInterface1();//upcasting
here
  message.msg();
 }
}
```

```
Output
hello nested interface
```

As you can see in the above example, we are accessing the Message interface by its outer interface Showable because it cannot be accessed directly. It is just like the almirah inside the room; we cannot access the almirah directly because we must enter the room first. In the collection framework, the sun microsystem has provided a nested interface Entry. Entry is the subinterface of Map, i.e., accessed by Map.Entry.

Variables in Interfaces

In java, an interface is a completely abstract class. An interface is a container of abstract methods and static final variables. The interface contains the static final variables. The variables defined in an interface can not be modified by the class that implements the interface, but it may use as it defined in the interface.

1. The variable in an interface is public, static, and final by default.
2. If any variable in an interface is defined without public, static, and final keywords then, the compiler automatically adds the same.
3. No access modifier is allowed except the public for interface variables.
4. Every variable of an interface must be initialized in the interface itself.
5. The class that implements an interface can not modify the interface variable, but it may use as it defined in the interface.

```
interface SampleInterface{

   int UPPER_LIMIT = 100;

   //int LOWER_LIMIT; // Error - must be initialised

}

public      class      InterfaceVariablesExample      implements
SampleInterface{

   public static void main(String[] args) {

         System.out.println("UPPER LIMIT = " + UPPER_LIMIT);

         // UPPER_LIMIT = 150; // Can not be modified
   }
}
```

```
Output
UPPER_LIMIT = 100
```

Extending Interface in Java with Example

Like classes, an interface can also extend another interface. This means that an interface can be sub interfaces from other interfaces.

The new sub-interface will inherit all members of the super interface similar to subclasses. It can be done by using the keyword "extends". It has the following general form:

```
Syntax:
  interface interfaceName2 extends interfaceName1
  {
    // body of interfaceName2.
  }
Ex. Multiple inheritance
```

Key points:

1. An interface cannot extend classes because it would violate rules that an interface can have only abstract methods and constants.
2. An interface can extend Interface1, Interface2.

Implementing Interface in Java with Example

An interface is used as "superclass" whose properties are inherited by a class. A class can implement one or more than one interface by using a keyword **implements** followed by a list of interfaces separated by commas.

When a class implements an interface, it must provide an implementation of all methods declared in the interface and all its super interfaces.

Otherwise, the class must be declared abstract. The general syntax of a class that implements an interface is as follows:

Syntax:

```
1. accessModifier class className implements interfaceName
   {
    // method implementations;
    // member declaration of class;
   }
2. A more general form of interface implementation is given below.
  accessModifier  class  className  extends  superClass  implements
interface1, interface2,.. .
  {
     // body of className.
  }
```

This general form shows that a class can extend another class while implementing interfaces.

Key points:

1. All methods of interfaces when implementing in a class must be declared as public otherwise you will get a compile-time error if any other modifier is specified.
2. Class extends class implements interface.
3. Class extends class implements Interface1, Interface2…

The implementation of interfaces can have the following general forms as shown in the below figure.

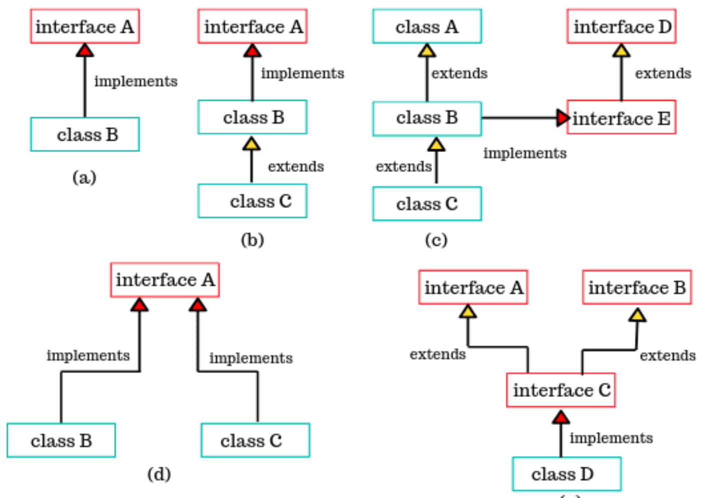


Fig: Various forms of interface implementation

The interface is also used to declare a set of constants that can be used in multiple classes. The constant values will be available to any classes that implement interface because it is by default public, static, and final.

The values can also be used in any method as part of the variable declaration or anywhere in the class.

Java instanceof

The java instanceof operator is used to test whether the object is an instance of the specified type (class or subclass or interface).

The instanceof in java is also known as type comparison operator because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false.

Simple example of java instanceof

Let's see the simple example of instance operator where it tests the current class.

```
class Simple1{
 public static void main(String args[]){
 Simple1 s=new Simple1();
 System.out.println(s instanceof Simple1);//true
 }
}
Output
True
```

Exceptions

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.

In this tutorial, we will learn about Java exceptions, it's types, and the difference between checked and unchecked exceptions.

What is Exception in Java?

Dictionary Meaning: Exception is an abnormal condition.

In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

What is Exception Handling?

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

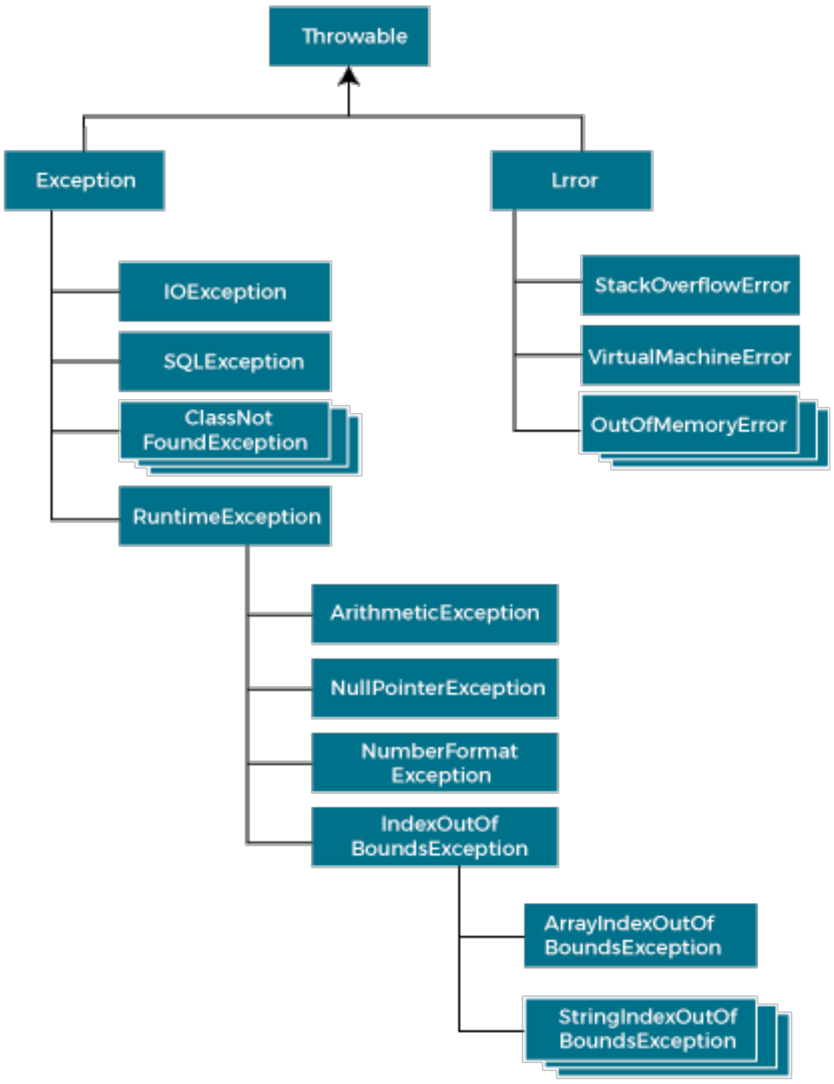
Advantage of Exception Handling

The core advantage of exception handling is to maintain the normal flow of the application. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions. Let's consider a scenario:

```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in Java.

Hierarchy of Java Exception classes



Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error

Java Exception Keywords

Java provides five keywords that are used to handle the exception. The following table describes each.

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

Let's see an example of Java Exception Handling in which we are using a try-catch statement to handle the exception.

```
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...
```

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
int a=50/0;//ArithmeticException
```

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```
String s=null;
System.out.println(s.length());//NullPointerException
```

3) A scenario where NumberFormatException occurs

If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a string variable that has characters; converting this variable into digit will cause NumberFormatException.

```
String s="abc";
int i=Integer.parseInt(s);//NumberFormatException
```

4) A scenario where ArrayIndexOutOfBoundsException occurs

When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

```
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```

EXAMPLE

```
import java.io.*;

class ListOfNumbers {

  // create an integer array
  private int[] list = {5, 6, 8, 9, 2};

  // method to write data from array to a fila
  public void writeList() {
    PrintWriter out = null;

    try {
      System.out.println("Entering try statement");

      // creating a new file OutputFile.txt
      out = new PrintWriter(new FileWriter("OutputFile.txt"));

      // writing values from list array to Output.txt
      for (int i = 0; i < 7; i++) {
        out.println("Value at: " + i + " = " + list[i]);
      }
    }

    catch (Exception e) {
      System.out.println("Exception => " + e.getMessage());
    }

    finally {
      // checking if PrintWriter has been opened
      if (out != null) {
        System.out.println("Closing PrintWriter");
        // close PrintWriter
        out.close();
      }

      else {
        System.out.println("PrintWriter not open");
      }
    }

  }
}
class Main {
  public static void main(String[] args) {
    ListOfNumbers list = new ListOfNumbers();
    list.writeList();
  }
}
```

Output

```
Entering try statement

Exception => Index 5 out of bounds for length 5

Closing PrintWriter
```

Multiple catch clauses

```
class Main {
  public static void main(String[] args) {
    try {
      int array[] = new int[10];
      array[10] = 30 / 0;
    } catch (ArithmeticException e) {
      System.out.println(e.getMessage());
    } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

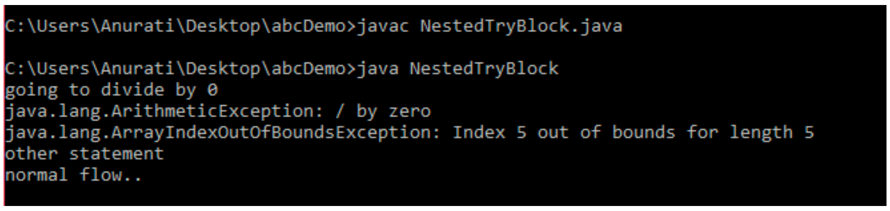
Output

```
/ by zero
```

Nested try blocks

```
public class NestedTryBlock{
 public static void main(String args[]){
 //outer try block
  try{
  //inner try block 1
    try{
     System.out.println("going to divide by 0");
     int b =39/0;
   }
    //catch block of inner try block 1
    catch(ArithmeticException e)
    {
      System.out.println(e);
    }


    //inner try block 2
    try{
    int a[]=new int[5];

    //assigning the value out of array bounds
     a[5]=4;
     }
    //catch block of inner try block 2
    catch(ArrayIndexOutOfBoundsException e)
    {
       System.out.println(e);
    }


    System.out.println("other statement");
  }
  //catch block of outer try block
  catch(Exception e)
  {
    System.out.println("handled the exception (outer catch)");
  }

  System.out.println("normal flow..");
 }
}
```

Output



```
C:\Users\Anurati\Desktop\abcDemo>javac NestedTryBlock.java

C:\Users\Anurati\Desktop\abcDemo>java NestedTryBlock
going to divide by 0
java.lang.ArithmeticException: / by zero
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5
other statement
normal flow..
```

Types of Exception in Java with Examples

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.

Built-in Exceptions

Built-in exceptions are the exceptions which 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.

- ArithmeticException
  - It is thrown when an exceptional condition has occurred in an arithmetic operation.
- 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.
- ClassNotFoundException
  - This Exception is raised when we try to access a class whose definition is not found
- FileNotFoundException
  - This Exception is raised when a file is not accessible or does not open.
- IOException
  - It is thrown when an input-output operation failed or interrupted
- InterruptedException
  - It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.
- NoSuchFieldException
  - It is thrown when a class does not contain the field (or variable) specified
- NoSuchMethodException
  - It is thrown when accessing a method which is not found.
- NullPointerException
  - This exception is raised when referring to the members of a null object. Null represents nothing
- NumberFormatException
  - This exception is raised when a method could not convert a string into a numeric format.
- RuntimeException
  - This represents any exception which occurs during runtime.
- StringIndexOutOfBoundsException

- It is thrown by String class methods to indicate that an index is either negative or greater than the size of the string

```
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
```

User-Defined Exceptions

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

Following steps are followed for the creation of user-defined Exception.

- The user should create an exception class as a subclass of Exception class. Since all the exceptions are subclasses of 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 super class(Exception) constructor from this and send the string there.

```
MyException(String str)
{
   super(str);
}
```

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

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

```
throw me;
```

- The following program illustrates how to create 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, 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".

```
class MyException extends Exception
{
    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();
        }
    }
}
```

Run time 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
```

Managing I/O

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, object, localized characters, etc.

Stream

A stream can be defined as a sequence of data. There are two kinds of Streams −

- InPutStream − The InputStream is used to read data from a source.
- OutPutStream − The OutputStream is used for writing data to a destination.

Java provides strong but flexible support for I/O related to files and networks but this tutorial covers very basic functionality related to streams and I/O. We will see the most commonly used examples one by one −

Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, FileInputStream and FileOutputStream. Following is an example which makes use of these two classes to copy an input file into an output file −

```
import java.io.*;
public class CopyFile {

   public static void main(String args[]) throws IOException {
      FileInputStream in = null;
      FileOutputStream out = null;

      try {
         in = new FileInputStream("input.txt");
         out = new FileOutputStream("output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

Now let's have a file **input.txt** with the following content −

```
This is test for copy file.
```

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following −

```
$javac CopyFile.java
$java CopyFile
```

Character Streams

Java Byte streams are used to perform input and output of 8-bit bytes, whereas Java Character streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, FileReader and FileWriter. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

We can re-write the above example, which makes the use of these two classes to copy an input file (having unicode characters) into an output file −

```
import java.io.*;
public class CopyFile {

   public static void main(String args[]) throws IOException {
      FileReader in = null;
      FileWriter out = null;

      try {
         in = new FileReader("input.txt");
         out = new FileWriter("output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

Now let's have a file **input.txt** with the following content −

```
This is test for copy file.
```

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following

```
$javac CopyFile.java
$java CopyFile
```

Predefined Streams or Standard Streams

All the programming languages provide support for standard I/O where the user's program can take input from a keyboard and then produce an output on the computer screen. If you are aware of C or C++ programming languages, then you must be aware of three standard devices STDIN, STDOUT and STDERR. Similarly, Java provides the following three standard streams −

- **Standard Input** − This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**.
- **Standard Output** − This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as **System.out**.
- **Standard Error** − This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as **System.err**.

Following is a simple program, which creates **InputStreamReader** to read standard input stream until the user types a "q" −

```
import java.io.*;
public class ReadConsole {

   public static void main(String args[]) throws IOException {
      InputStreamReader cin = null;

      try {
         cin = new InputStreamReader(System.in);
         System.out.println("Enter characters, 'q' to quit.");
         char c;
         do {
            c = (char) cin.read();
            System.out.print(c);
         } while(c != 'q');
      }finally {
         if (cin != null) {
            cin.close();
         }
      }
   }
}
```

Let's keep the above code in ReadConsole.java file and try to compile and execute it as shown in the following program. This program continues to read and output the same character until we press 'q' −

```
$javac ReadConsole.java
$java ReadConsole
Enter characters, 'q' to quit.
1
1
e
e
q
q
```

Reading Console input

By default, to read from system console, we can use the Console class. This class provides methods to access the character-based console, if any, associated with the current Java process. To get access to Console, call the method System.console().

Console gives three ways to read the input:

- String readLine() – reads a single line of text from the console.
- char[] readPassword() – reads a password or encrypted text from the console with echoing disabled
- Reader reader() – retrieves the Reader object associated with this console. This reader is supposed to be used by sophisticated applications.

For example, Scanner object which utilizes the rich parsing/scanning functionality on top of the underlying Reader.

1. **Java program to read console input with readLine()**

```
Console console = System.console();

if(console == null) {
    System.out.println("Console is not available to current JVM
process");
    return;
}

String userName = console.readLine("Enter the username: ");
```

```
System.out.println("Entered username: " + userName);
```

```
Output Console
Enter the username: lokesh
Entered username: lokesh
```

2. **Java program to read console input with readPassword()**

```
Console console = System.console();

if(console == null) {
    System.out.println("Console is not available to current JVM
process");
    return;
}

char[] password = console.readPassword("Enter the password: ");
System.out.println("Entered password: " + new String(password));
```

```
Output Console
Enter the password:     //input will not visible in the console
Entered password: passphrase
```

3. **Java program to read console input with reader()**

```
Console console = System.console();

if(console == null) {
    System.out.println("Console is not available to current JVM
process");
    return;
}

Reader consoleReader = console.reader();
Scanner scanner = new Scanner(consoleReader);

System.out.println("Enter age:");
int age = scanner.nextInt();
System.out.println("Entered age: " + age);

scanner.close();
```

```
Output Console
Enter age:
12
Entered age: 12
```

Writing Console Output

The easiest way to write the output data to console is System.out.println() statements. Still, we can use printf() methods to write formatted text to console.

1. **Java program to write to console with System.out.println**

```
System.out.println("Hello, world!");
System.out.println() method example
```

Program output

```
Hello, world!
```

2. **Java program to write to console with printf()**

The printf(String format, Object... args) method takes an output string and multiple parameters which are substituted in the given string to produce the formatted output content. This formatted output is written in the console.

```
String name = "Lokesh";
int age = 38;
console.printf("My name is %s and my age is %d", name, age);
```

Program output

```
My name is Lokesh and my age is 38
```

The above listed methods for reading the input and writing output to the console provide alot of flexibility to read inputs in different formats and in different ways.

Print Writer class

Java PrintWriter class is the implementation of Writer class. It is used to print the formatted representation of objects to the text-output stream.

Class declaration

Let's see the declaration for Java.io.PrintWriter class:

```
public class PrintWriter extends Writer
```

Methods of PrintWriter class

Method	**Description**
void println(boolean x)	It is used to print the boolean value.
void println(char[] x)	It is used to print an array of characters.
void println(int x)	It is used to print an integer.
PrintWriter append(char c)	It is used to append the specified character to the writer.
PrintWriter append(CharSequence ch)	It is used to append the specified character sequence to the writer.

PrintWriter append(CharSequence ch, int start, int end)	It is used to append a subsequence of specified character to the writer.
boolean checkError()	It is used to flushes the stream and check its error state.
protected void setError()	It is used to indicate that an error occurs.
protected void clearError()	It is used to clear the error state of a stream.
PrintWriter format(String format, Object... args)	It is used to write a formatted string to the writer using specified arguments and format string.
void print(Object obj)	It is used to print an object.
void flush()	It is used to flushes the stream.
void close()	It is used to close the stream.

Java PrintWriter Example

Let's see the simple example of writing the data on a **console** and in a **text file testout.txt** using Java PrintWriter class.

```
package com.javatpoint;
import java.io.File;
import java.io.PrintWriter;
public class PrintWriterExample {
    public static void main(String[] args) throws Exception {
             //Data to write on Console using PrintWriter
      PrintWriter writer = new PrintWriter(System.out);
      writer.write("Dr. Selva Mary’s Learning Tutorial ");
 writer.flush();
      writer.close();
//Data to write in File using PrintWriter
      PrintWriter writer1 =null;
         writer1 = new PrintWriter(new File("D:\\testout.txt"));
         writer1.write("Like Java, Spring, Hibernate, Android, PHP
etc.");
                         writer1.flush();
         writer1.close();
    }
}
```

Output

```
Dr. Selva Mary’s Learning Tutorial
```

****