

## UNIT II PACKAGE, INHERITANCE AND EXCEPTION HANDLING

Package Access - Java API Packages - Basics of Inheritance - Forms of Inheritance - Sub Classes and Subclass Types - Super keyword - Final - Method Overriding - Abstract Classes - Interfaces.

Exception Handling: Java Exception Hierarchy - Exception Types - Throwing and Catching exceptions - Declaring New Exception Types.

### 2.1 Java Package

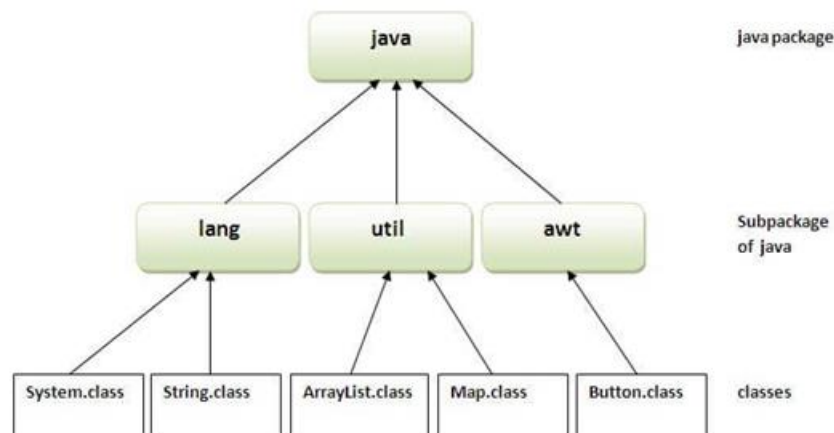
A **java package** is a group of similar types of classes, interfaces and sub-packages.

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.

#### 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.



#### Simple example of java package

The **package** keyword is used to create a package in java.

```
//save as Simple.java
package mypack;
public class Simple{
    public static void main(String args[]){
        System.out.println("Welcome to package");
    }
}
```

#### How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

1. javac -d directory javafilename

For **example**

1. javac -d . Simple.java

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

#### How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

**To Compile:** javac -d . Simple.java

**To Run:** java mypack.Simple

Output: Welcome to package

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

### How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.\*;
2. import package.classname;
3. fully qualified name.

#### 1) Using packagename.\*

If you use package.\* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.\*

//save by A.java

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

//save by B.java

```
package mypack;
import pack.*;
```

```
class B{
    public static void main(String args[]){
        A obj = new A();
        obj.msg();
    }
}
```

Output: Hello

#### 2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

//save by A.java

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

//save by B.java

```
package mypack;
import pack.A;
```

```
class B{
    public static void main(String args[]){
        A obj = new A();
        obj.msg();
    }
}
```

Output: Hello

#### 3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

#### Example of package by import fully qualified name

//save by A.java

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

//save by B.java

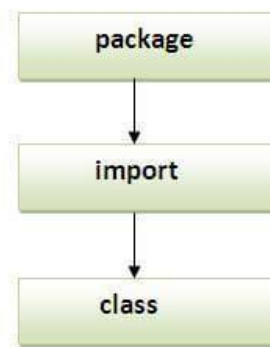
```
package mypack;  
class B{  
    public static void main(String args[]){  
        pack.A obj = new pack.A();//using fully qualified name  
        obj.msg();  
    }  
}
```

Output:Hello

Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

*Note: Sequence of the program must be package then import then class.*



#### Subpackage in java

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

Let's take an example, Sun Microsystems has defined a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

The standard of defining package is domain.company.package e.g. com.javatpoint.bean or org.sssit.dao.

#### Example of Subpackage

```
package com.javatpoint.core;  
class Simple{  
    public static void main(String args[]){  
        System.out.println("Hello subpackage");  
    }  
}
```

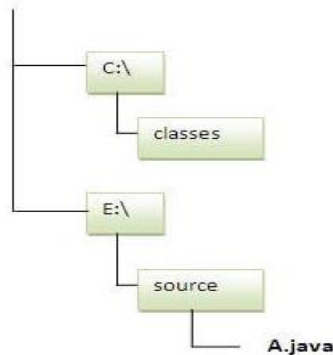
**To Compile:** javac -d . Simple.java

**To Run:** java com.javatpoint.core.Simple

Output:Hello subpackage

How to send the class file to another directory or drive?

I want to put the class file of A.java source file in classes folder of c: drive. For example:



//save as Simple.java

```
package mypack;
public class Simple{
    public static void main(String args[]){
        System.out.println("Welcome to package");
    }
}
```

To Compile:

```
e:\sources> javac -d c:\classes Simple.java
```

To Run:

To run this program from e:\source directory, you need to set classpath of the directory where the class file resides.

```
e:\sources> set classpath=c:\classes;.;
```

```
e:\sources> java mypack.Simple
```

Another way to run this program by -classpath switch of java:

The -classpath switch can be used with javac and java tool.

To run this program from e:\source directory, you can use -classpath switch of java that tells where to look for class file. For example:

```
e:\sources> java -classpath c:\classes mypack.Simple
```

Output: Welcome to package

### Ways to load the class files or jar files

There are two ways to load the class files temporary and permanent.

- Temporary
  - By setting the classpath in the command prompt
  - By -classpath switch
- Permanent
  - By setting the classpath in the environment variables
  - By creating the jar file, that contains all the class files, and copying the jar file in the jre/lib/ext folder.

Rule: There can be only one public class in a java source file and it must be saved by the public class name.

//save as C.java otherwise Compile Time Error

```
class A{}
class B{}
public class C{}
```

### How to put two public classes in a package?

If you want to put two public classes in a package, have two java source files containing one public class, but keep the package name same. For example:

//save as A.java

```
package javatpoint;
```

```
public class A{ }
```

//save as B.java

```
package javatpoint;
```

```
public class B{ }
```

What is static import feature of Java5?

Click [Static Import](#) feature of Java5.

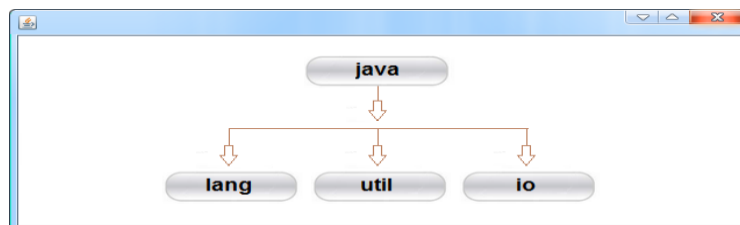
### Access Modifiers in Java

As the name suggests access modifiers in Java helps to restrict the scope of a class, constructor , variable , method or data member. There are four types of access modifiers available in java:

1. Default – No keyword required
2. Private
3. Protected
4. Public

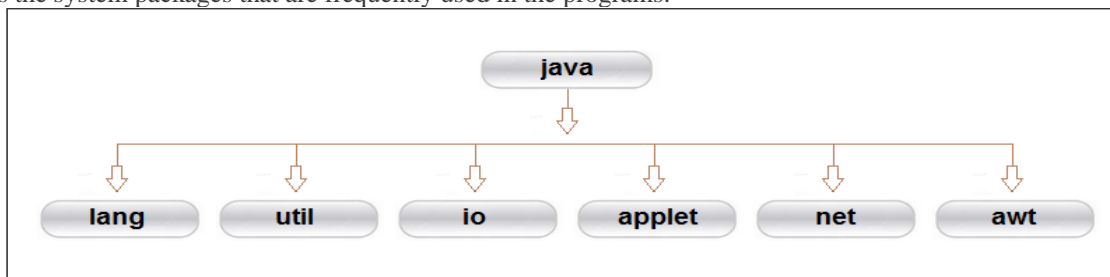
	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

## 2.2 Java-API Packages



### API Packages

Java API(Application Program Interface) provides a large numbers of classes grouped into different packages according to functionality. Most of the time we use the packages available with the the Java API. Following figure shows the system packages that are frequently used in the programs.



### Java System Packages and Their Classes

<b>java.lang</b>	Language support classes. They include classes for primitive types, string, math functions, thread and exceptions.
<b>java.util</b>	Language utility classes such as vectors, hash tables, random numbers, data, etc.
<b>java.io</b>	Input/output support classes. They provide facilities for the input and output of data.
<b>java.applet</b>	Classes for creating and implementing applets.
<b>java.net</b>	Classes for networking. They include classes for communicating with local computers as well as with internet servers.
<b>java.awt</b>	Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.

## 2.3 Inheritance in Java

1. [Inheritance](#)
2. [Types of Inheritance](#)
3. [Why multiple inheritance is not possible in java in case of class?](#)

**Inheritance in java** is a mechanism in which one object acquires all the properties and behaviors of parent object. It is an important part of OPPs(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 parent class, and you can add new methods and fields also.

Inheritance represents the **IS-A relationship**, also known as *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 previous class.

### Syntax of Java Inheritance

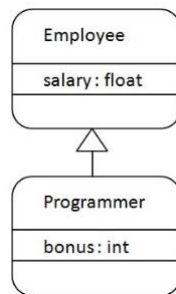
**class** Subclass-name **extends** Superclass-name

```
{  
    //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 parent or super class 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. Relationship between 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

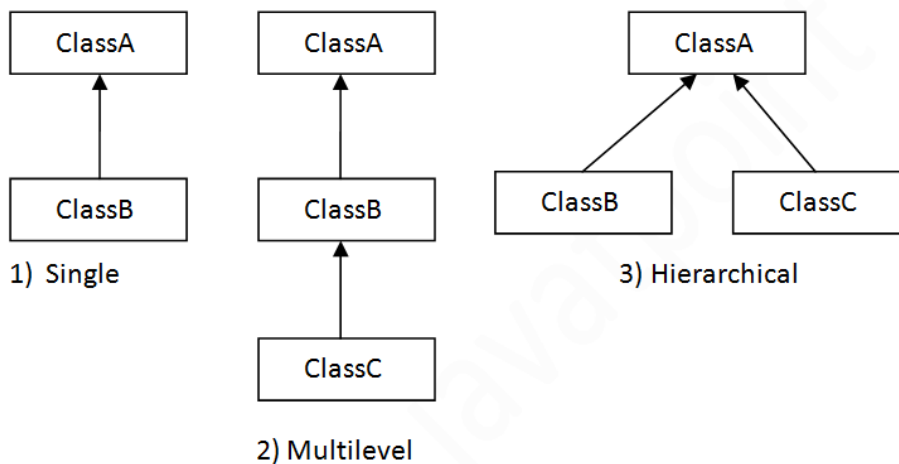
Bonus of programmer is:10000

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

## 2.4 & 2.5 Forms/ Types of inheritance in java

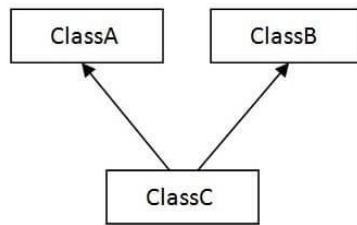
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only.

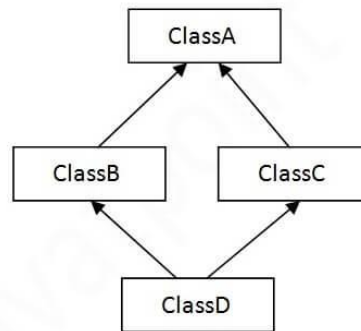


Note: Multiple inheritance is not supported in java through class.

When a class extends multiple classes i.e. known as multiple inheritance. For Example:



4) Multiple



5) Hybrid

### Single Inheritance Example

*File: TestInheritance.java*

```

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

#### **Output:**

barking...  
eating...

### Multilevel Inheritance Example

*File: TestInheritance2.java*

```

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

#### **Output:**

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



### Hierarchical Inheritance Example

File: TestInheritance3.java

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

Output:

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

### Q) Why multiple inheritance is not supported in java?

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

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

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

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

**Output**

```
Compile Time Error
```

## 2. 6 super keyword in java

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

```

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.

### 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...

```

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.

### 3) 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");
}
}

```

```

}
class TestSuper3{
public static void main(String args[]){
Dog d=new Dog();
}}

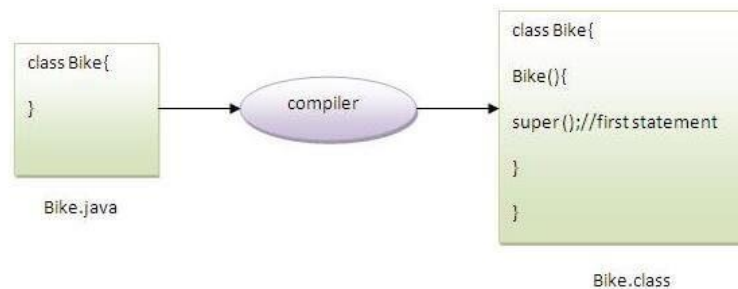
```

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");
}
}
class TestSuper4{
public static void main(String args[]){
Dog d=new Dog();
}}

```

Output:

animal is created

dog is created

**super example: real use**

Let's see the real use of super keyword. Here, Emp class inherits Person class so all the properties of Person will be inherited to Emp by default. To initialize all the property, we are using parent class constructor from child class. In such way, we are reusing the parent class constructor.

```

class Person{
int id;
String name;
Person(int id,String name){
this.id=id;
this.name=name;
}
}
class Emp extends Person{
float salary;
Emp(int id,String name,float salary){
super(id,name);//reusing parent constructor
this.salary=salary;
}
}

```

```

void display(){System.out.println(id+" "+name+" "+salary);}
}
class TestSuper5{
public static void main(String[] args){
Emp e1=new Emp(1,"ankit",45000f);
e1.display();
}}

```

Output:

```
1 ankit 45000
```

## 2.7 Final Keyword In Java

1. [Final variable](#)
2. [Final method](#)
3. [Final class](#)
4. [Is final method inherited ?](#)
5. [Blank final variable](#)
6. [Static blank final variable](#)
7. [Final parameter](#)
8. [Can you declare a final constructor](#)

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. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.



### 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 speed limit, 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();
}
}

```

**Test it Now**

Output:Compile Time Error

### 2) Java final method

If you make any method as final, you cannot override it.

Example of final method

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

class Honda extends Bike{
    void run(){System.out.println("running safely with 100kmph");}
    public static void main(String args[]){
        Honda honda= new Honda();
        honda.run();
    }
}
```

Output:Compile Time Error

### 3) 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

### Q) Is final method inherited?

Ans) Yes, final method is inherited but you cannot override it. For Example:

```
class Bike{
    final void run(){System.out.println("running...");}
}
class Honda2 extends Bike{
    public static void main(String args[]){
        new Honda2().run();
    }
}
```

Output:running...

### Q) What is blank or uninitialized final variable?

A final variable that is not initialized at the time of declaration is known as blank final variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.

It can be initialized only in constructor.

Example of blank final variable

```
class Student{
    int id;
    String name;
    final String PAN_CARD_NUMBER;
    ...
}
```

Que) Can we initialize blank final variable?

Yes, but only in constructor. For example:

```

class Bike10{
    final int speedlimit;//blank final variable

    Bike10(){
        speedlimit=70;
        System.out.println(speedlimit);
    }

    public static void main(String args[]){
        new Bike10();
    }
}

```

Output: 70

### static blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

Example of static blank final variable

```

class A{
    static final int data;//static blank final variable
    static{ data=50;}
    public static void main(String args[]){
        System.out.println(A.data);
    }
}

```

### Q) What is final parameter?

If you declare any parameter as final, you cannot change the value of it.

```

class Bike11{
    int cube(final int n){
        n=n+2;//can't be changed as n is final
        n*n*n;
    }
    public static void main(String args[]){
        Bike11 b=new Bike11();
        b.cube(5);
    }
}

```

Output: Compile Time Error

### Q) Can we declare a constructor final?

No, because constructor is never inherited

## 2.8 Method Overriding in Java

1. [Understanding problem without method overriding](#)
2. [Can we override the static method](#)
3. [method overloading vs 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 subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

### Usage of Java Method Overriding

- Method overriding is used to provide specific implementation of a method that is already provided by its super class.
- Method overriding is used for runtime polymorphism

### Rules for Java Method Overriding

1. method must have same name as in the parent class
2. method must have same parameter as in the parent class.
3. must be IS-A relationship (inheritance).

### Understanding the problem without method overriding

Let's understand the problem that we may face in the program if we don't use method overriding.

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

Output: Vehicle is running

Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

### Example of method overriding

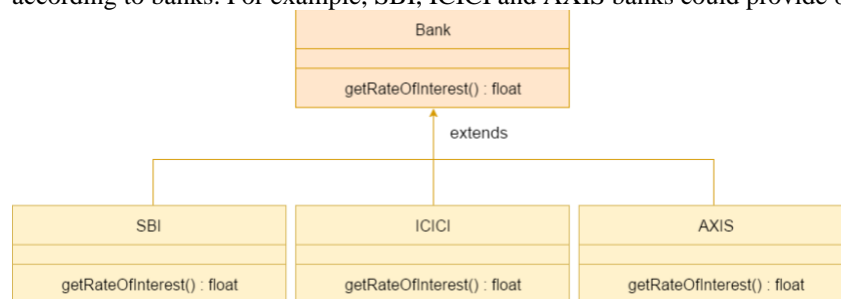
In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method is same and there is IS-A relationship between the classes, so there is method overriding.

```
class Vehicle{  
    void run(){System.out.println("Vehicle is running");}  
}  
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();  
    }  
}
```

Output: Bike is running safely

### Real example of Java Method Overriding

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



```
class Bank{
```

```

int getRateOfInterest(){return 0;}
}

class SBI extends Bank{
int getRateOfInterest(){return 8;}
}

class ICICI extends Bank{
int getRateOfInterest(){return 7;}
}
class AXIS extends Bank{
int getRateOfInterest(){return 9;}
}

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

Can we override static method?

No, static method cannot be overridden. It can be proved by runtime polymorphism, so we will learn it later.

### Why we cannot override static method?

because static method is bound with class whereas instance method is bound with object. Static belongs to class area and instance belongs to heap area.

### Can we override java main method?

No, because main is a static method.

### Difference between method overloading and method overriding in java

There are many differences between method overloading and method overriding in java. A list of differences between method overloading and method overriding are given below:

No.	Method Overloading	Method Overriding
1)	Method overloading is used to <i>increase the readability</i> of the program.	Method overriding is used to <i>provide the specific implementation</i> of the method that is already provided by its super class.
2)	Method overloading is performed <i>within class</i> .	Method overriding occurs <i>in two classes</i> that have IS-A (inheritance) relationship.
3)	In case of method overloading, <i>parameter must be different</i> .	In case of method overriding, <i>parameter must be same</i> .



4)	Method overloading is the example of <i>compile time polymorphism</i> .	Method overriding is the example of <i>run time polymorphism</i> .
5)	In java, method overloading can't be performed by changing return type of the method only. <i>Return type can be same or different</i> in method overloading. But you must have to change the parameter.	<i>Return type must be same or covariant</i> in method overriding.

#### Java Method Overloading example

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

#### Java Method Overriding example

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

## 2.9 Abstract class in Java

A class that is declared with abstract keyword, is known as abstract class in java. It can have abstract and non-abstract methods (method with body).

Before learning java abstract class, let's understand the abstraction in java first.

#### Abstraction in Java

**Abstraction** is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

#### Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

#### Abstract class in Java

A class that is declared as abstract is known as **abstract class**. It needs to be extended and its method implemented. It cannot be instantiated.

#### Example abstract class

1. **abstract class** A{ }

#### abstract method

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

#### Example abstract method

1. **abstract void** printStatus();//no body and abstract

#### Example of abstract class that has abstract method

In this example, Bike the 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();
}
}

```

running safely..

#### Understanding the real scenario of abstract class

In this example, Shape is the abstract class, its implementation is provided by the Rectangle and Circle classes. Mostly, we don't know about the implementation class (i.e. hidden to the end user) and object of the implementation class is provided by the **factory method**.

A **factory method** is the method that returns the instance of the class. We will learn about the factory method later.

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

#### File: TestAbstraction1.java

```

abstract class Shape{
    abstract void draw();
}

//In real scenario, implementation is provided by others i.e. unknown by end user
class Rectangle extends Shape{
    void draw(){System.out.println("drawing rectangle");}
}
class Circle1 extends Shape{
    void draw(){System.out.println("drawing circle");}
}

//In real scenario, method is called by programmer or user
class TestAbstraction1{
    public static void main(String args[]){
        Shape s=new Circle1();//In real scenario, object is provided through method e.g. getShape() method
        s.draw();
    }
}

```

drawing circle

#### Another example of abstract class in java

File: TestBank.java

```

abstract class Bank{
    abstract int getRateOfInterest();
}

class SBI extends Bank{
    int getRateOfInterest(){return 7;}
}

class PNB extends Bank{
    int getRateOfInterest(){return 8;}
}

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

```

Rate of Interest is: 7 %

Rate of Interest is: 8 %

### Abstract class having constructor, data member, methods etc.

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

File: TestAbstraction2.java

//example of abstract class that have method body

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

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

```
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 any abstract method in a class, that class must be abstract.

```
class Bike12{
    abstract void run();
}
```

Output:

compile time error

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

### Another real scenario of abstract class

The abstract class can also be used to provide some implementation of the interface. In such case, the end user may not be forced to override all the methods of the interface.

**Note: If you are beginner to java, learn interface first and skip this example.**

```
interface A{
    void a();
    void b();
    void c();
    void d();
}
```

```
abstract class B implements A{
    public void c(){System.out.println("I am c");}
}
```

```
class M extends B{
    public void a(){System.out.println("I am a");}
    public void b(){System.out.println("I am b");}
    public void d(){System.out.println("I am d");}
}
```

```

class Test5{
public static void main(String args[]){
A a=new M();
a.a();
a.b();
a.c();
a.d();
}}

```

Output:I am a

```

I am b
I am c
I am d

```

## 2.10 Interface in Java

1. [Interface](#)
2. [Example of Interface](#)
3. [Multiple inheritance by Interface](#)
4. [Why multiple inheritance is supported in Interface while it is not supported in case of class.](#)
5. [Marker Interface](#)
6. [Nested Interface](#)

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 methods and variables but the methods declared in interface contain only method signature, not body.

Java Interface also **represents IS-A relationship**.

It cannot be instantiated just like abstract class.

**Why use Java 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 interface?

Interface is declared by using interface keyword. It provides total abstraction; means all the methods in interface are declared with empty body and are public and all fields are public, static and final by default. A class that implement interface must implement all the methods declared in the interface.

**Syntax:**

```

interface <interface_name>{

```

```

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

```

```

}

```

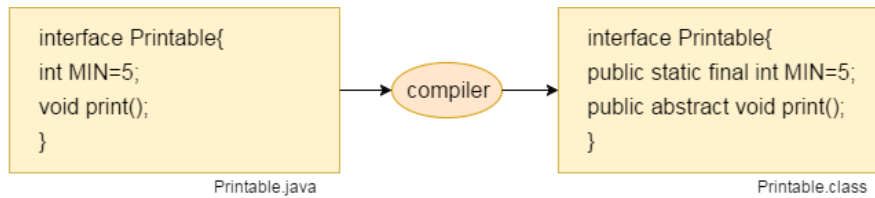
### **Java 8 Interface Improvement**

Since Java 8, interface can have default and static methods which is discussed later.

**Internal addition by compiler**

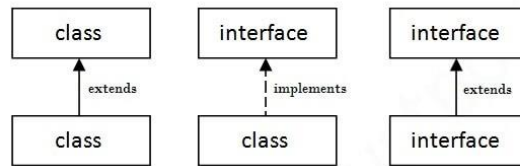
In other words, Interface fields are public, static and final by default, and methods are public and abstract.

The java compiler adds public and abstract keywords before the interface method. More, it adds public, static and final keywords before data members.



### Understanding 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 Example

In this example, Printable interface has only one method, its implementation is provided in the A 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

### Java Interface Example: Drawable

In this example, Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In real scenario, interface is defined by someone but implementation is provided by different implementation providers. And, it is used by someone else. The implementation part is hidden by the user which uses the interface.

File: TestInterface1.java

```
//Interface declaration: by first user
interface Drawable{
void draw();
}

//Implementation: by second user
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class Circle implements Drawable{
public void draw(){System.out.println("drawing circle");}
}

//Using interface: by third user
class TestInterface1{
public static void main(String args[]){
```

```
Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDrawable()
d.draw();
}}
```

Output:

drawing circle

### Java Interface Example: Bank

Let's see another example of java interface which provides the implementation of Bank interface.

File: TestInterface2.java

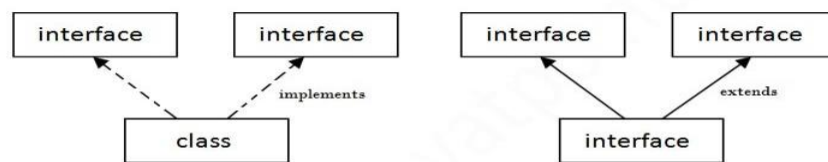
```
interface Bank{
float rateOfInterest();
}
class SBI implements Bank{
public float rateOfInterest(){return 9.15f;}
}
class PNB implements Bank{
public float rateOfInterest(){return 9.7f;}
}
class TestInterface2{
public static void main(String[] args){
Bank b=new SBI();
System.out.println("ROI: "+b.rateOfInterest());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

Output:

ROI: 9.15

### Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.



Multiple Inheritance in Java

```
interface Printable{
void print();
}
interface Showable{
void show();
}
class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}
```

```
public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
}
}
```

Output:Hello  
Welcome

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

As we have explained in the inheritance chapter, multiple inheritance is not supported in case of class because of ambiguity. But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class. For example:

```
interface Printable{
    void print();
}
interface Showable{
    void print();
}
```

```
class TestInterface3 implements Printable, Showable{
    public void print(){System.out.println("Hello");}
    public static void main(String args[]){
        TestInterface3 obj = new TestInterface3();
        obj.print();
    }
}
```

Output:

Hello

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestInterface1, so there is no ambiguity.

### Interface inheritance

A class implements interface but one interface extends another interface .

```
interface Printable{
    void print();
}
interface Showable extends Printable{
    void show();
}
class TestInterface4 implements Showable{
    public void print(){System.out.println("Hello");}
    public void show(){System.out.println("Welcome");}

    public static void main(String args[]){
        TestInterface4 obj = new TestInterface4();
        obj.print();
        obj.show();
    }
}
```

Output:

Hello

Welcome

### Java 8 Default Method in Interface

Since Java 8, we can have method body in interface. But we need to make it default method. Let's see an example:

File: TestInterfaceDefault.java

```
interface Drawable{
    void draw();
    default void msg(){System.out.println("default method");}
}
class Rectangle implements Drawable{
    public void draw(){System.out.println("drawing rectangle");}
}
class TestInterfaceDefault{
```

```

public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
d.msg();
}}

```

Output:

```

drawing rectangle
default method

```

#### Java 8 Static Method in Interface

Since Java 8, we can have static method in interface. Let's see an example:

File: TestInterfaceStatic.java

```

interface Drawable{
void draw();
static int cube(int x){return x*x*x;}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}

```

```

class TestInterfaceStatic{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
System.out.println(Drawable.cube(3));
}}

```

Output:

```

drawing rectangle
27

```

#### Q) What is marker or tagged interface?

An interface which has no member is known as marker or tagged interface. For example: Serializable, Cloneable, Remote etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

//How Serializable interface is written?

```

public interface Serializable{
}

```

#### Nested Interface in Java

Note: An interface can have another interface i.e. known as nested interface. We will learn it in detail in the nested classes chapter. For example:

```

interface printable{
void print();
interface MessagePrintable{
void msg();
}
}

```

## 2B.1 Exception Handling in Java

1. [Exception Handling](#)
2. [Advantage of Exception Handling](#)
3. [Hierarchy of Exception classes](#)
4. [Types of Exception](#)
5. [Scenarios where exception may occur](#)

The **exception handling in java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.



In this page, we will learn about java exception, its type and the difference between checked and unchecked exceptions.

### What is exception

**Dictionary Meaning:** Exception is an abnormal condition.

In java, 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, IO, SQL, Remote etc.

### Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application**. Exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

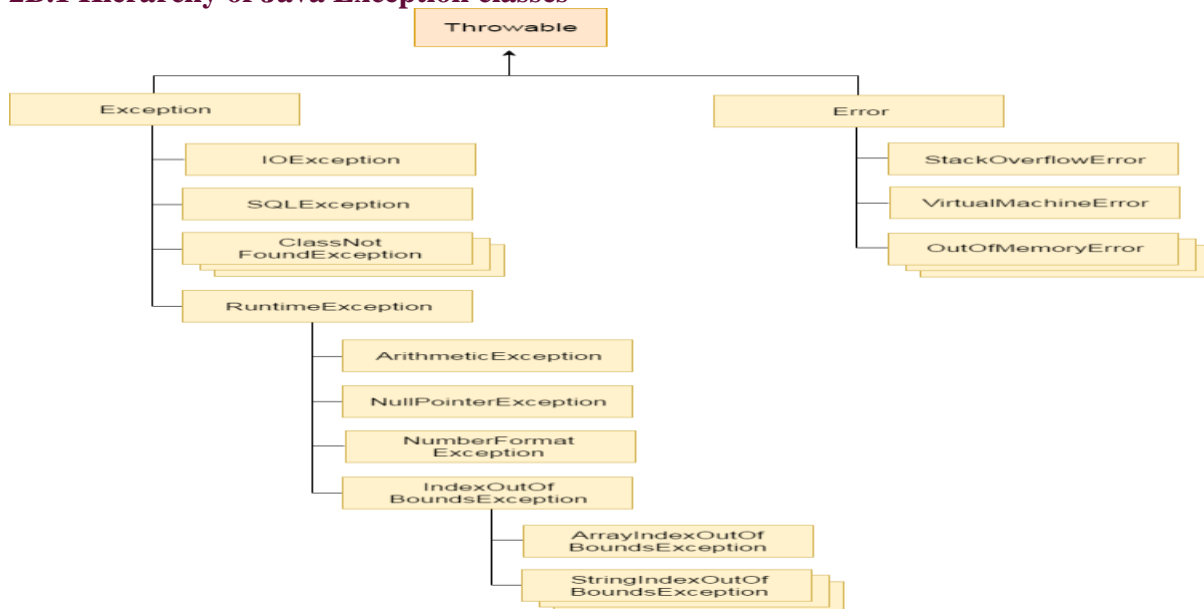
```
statement 1;  
statement 2;  
statement 3;  
statement 4;  
statement 5; //exception occurs  
statement 6;  
statement 7;
```

Suppose there is 10 statements in your program and there occurs an exception at statement 5, rest of the code will not be executed i.e. statement 6 to 10 will not run. If we perform exception handling, rest of the statement will be executed. That is why we use exception handling in java.

Do You Know ?

- What is the difference between checked and unchecked exceptions ?
- What happens behind the code `int data=50/0;` ?
- Why use multiple catch block ?
- Is there any possibility when finally block is not executed ?
- What is exception propagation ?
- What is the difference between throw and throws keyword ?
- What are the 4 rules for using exception handling with method overriding ?

## 2B.1 Hierarchy of Java Exception classes



## **2B .2 Types of Exception**

There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception. The sun microsystem says there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

### **Difference between checked and unchecked exceptions**

#### **1) Checked Exception**

The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

#### **2) Unchecked Exception**

The classes that extend RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

#### **3) Error**

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

### **Common scenarios where exceptions may occur**

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

#### **1) Scenario where ArithmeticException occurs**

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

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

#### **2) Scenario where NullPointerException occurs**

If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.

```
String s=null;
```

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

#### **3) Scenario where NumberFormatException occurs**

The wrong formatting of any value, may occur NumberFormatException. Suppose I have a string variable that have characters, converting this variable into digit will occur NumberFormatException.

```
String s="abc";
```

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

#### **4) Scenario where ArrayIndexOutOfBoundsException occurs**

If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:

```
int a[]=new int[5];
```

```
a[10]=50; //ArrayIndexOutOfBoundsException
```

## **2B.3 Java Exception Handling Keywords**

There are 5 keywords used in java exception handling.

1. try
2. catch
3. finally
4. throw
5. throws

Java try-catch

Java try block

Java try block is used to enclose the code that might throw an exception. It must be used within the method.

Java try block must be followed by either catch or finally block.

### **Syntax of java try-catch**

```
try{
```

```
//code that may throw exception
```

```

}catch(Exception_class_Name ref){} Syntax of try-finally block
try{
//code that may throw exception
}finally{}

```

### Java catch block

Java catch block is used to handle the Exception. It must be used after the try block only. You can use multiple catch block with a single try.

### Problem without exception handling

Let's try to understand the problem if we don't use try-catch block.

```

public class Testtrycatch1{
    public static void main(String args[]){
        int data=50/0;//may throw exception
        System.out.println("rest of the code...");
    }
}

```

Output:

Exception in thread main java.lang.ArithmeticException:/ by zero

As displayed in the above example, rest of the code is not executed (in such case, rest of the code... statement is not printed).

There can be 100 lines of code after exception. So all the code after exception will not be executed.

### Solution by exception handling

Let's see the solution of above problem by java try-catch block.

```

public class Testtrycatch2{
    public static void main(String args[]){
        try{
            int data=50/0;
        }catch(ArithmeticException e){System.out.println(e);}
        System.out.println("rest of the code...");
    }
}

```

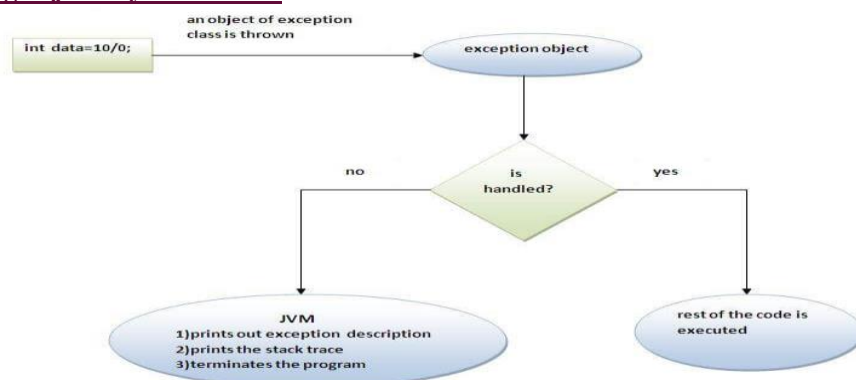
Output:

Exception in thread main java.lang.ArithmeticException:/ by zero

rest of the code...

Now, as displayed in the above example, rest of the code is executed i.e. rest of the code... statement is printed.

### Internal working of java try-catch block



The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).

- Causes the program to terminate.

But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

### Java catch multiple exceptions

#### Java Multi catch block

If you have to perform different tasks at the occurrence of different Exceptions, use java multi catch block.

Let's see a simple example of java multi-catch block.

```
public class TestMultipleCatchBlock{
    public static void main(String args[]){
        try{
            int a[]=new int[5];
            a[5]=30/0;
        }
        catch(ArithmeticException e){System.out.println("task1 is completed");}
        catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
        catch(Exception e){System.out.println("common task completed");}

        System.out.println("rest of the code...");
    }
}
```

Output:task1 completed  
rest of the code...

**Rule:** At a time only one Exception is occurred and at a time only one catch block is executed.

Rule: All catch blocks must be ordered from most specific to most general i.e. catch for ArithmeticException must come before catch for Exception .

```
class TestMultipleCatchBlock1{
    public static void main(String args[]){
        try{
            int a[]=new int[5];
            a[5]=30/0;
        }
        catch(Exception e){System.out.println("common task completed");}
        catch(ArithmeticException e){System.out.println("task1 is completed");}
        catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
        System.out.println("rest of the code...");
    }
}
```

Output:  
Compile-time error

#### Java Nested try block

The try block within a try block is known as nested try block in java.

#### Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

#### Syntax:

```
....
try
{
    statement 1;
    statement 2;
    try
    {
```

```

        statement 1;
        statement 2;
    }
    catch(Exception e)
    {
    }
}
catch(Exception e)
{
}
....

```

#### Java nested try example

Let's see a simple example of java nested try block.

```

class Excep6{
    public static void main(String args[]){
        try{
            try{
                System.out.println("going to divide");
                int b =39/0;
            }catch(ArithmeticException e){System.out.println(e);}

            try{
                int a[]=new int[5];
                a[5]=4;
            }catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}

            System.out.println("other statement");
        }catch(Exception e){System.out.println("handeled");}

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

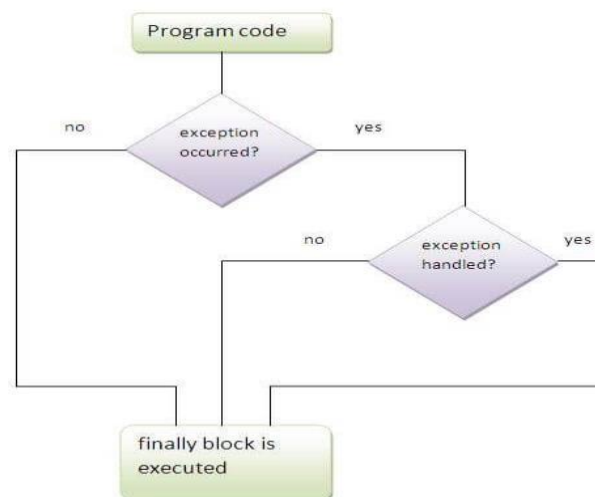
```

#### Java finally block

**Java finally block** is a block that is used *to execute important code* such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not.

Java finally block follows try or catch block.



Note: If you don't handle exception, before terminating the program, JVM executes finally block(if any).

#### Why use java finally

- Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

### Usage of Java finally

Let's see the different cases where java finally block can be used.

#### Case 1

Let's see the java finally example where **exception doesn't occur**.

```
class TestFinallyBlock{
    public static void main(String args[]){
        try{
            int data=25/5;
            System.out.println(data);
        }
        catch(NullPointerException e){System.out.println(e);}
        finally{System.out.println("finally block is always executed");}
        System.out.println("rest of the code...");
    }
}
```

Output:5  
finally block is always executed  
rest of the code...

#### Case 2

Let's see the java finally example where **exception occurs and not handled**.

```
class TestFinallyBlock1{
    public static void main(String args[]){
        try{
            int data=25/0;
            System.out.println(data);
        }
        catch(NullPointerException e){System.out.println(e);}
        finally{System.out.println("finally block is always executed");}
        System.out.println("rest of the code...");
    }
}
```

Output:finally block is always executed  
Exception in thread main java.lang.ArithmeticException:/ by zero

#### Case 3

Let's see the java finally example where **exception occurs and handled**.

```
public class TestFinallyBlock2{
    public static void main(String args[]){
        try{
            int data=25/0;
            System.out.println(data);
        }
        catch(ArithmeticException e){System.out.println(e);}
        finally{System.out.println("finally block is always executed");}
        System.out.println("rest of the code...");
    } }
}
```

Output:Exception in thread main java.lang.ArithmeticException:/ by zero  
finally block is always executed  
rest of the code...

Rule: For each try block there can be zero or more catch blocks, but only one finally block.

Note: The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

### Java throw exception

#### Java throw keyword

The Java throw keyword is used to explicitly throw an exception.

We can throw either checked or unchecked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

The syntax of java throw keyword is given below.

1. **throw** exception;

Let's see the example of throw IOException.

1. **throw new** IOException("sorry device error);

**java throw keyword example**

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
public class TestThrow1 {
    static void validate(int age){
        if(age<18)
            throw new ArithmeticException("not valid");
        else
            System.out.println("welcome to vote");
    }
    public static void main(String args[]){
        validate(13);
        System.out.println("rest of the code...");
    }
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:not valid
```

#### Java Exception propagation

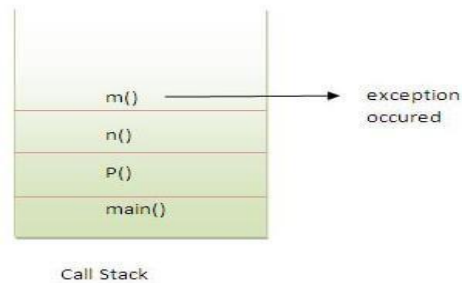
An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method. If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.

Rule: By default Unchecked Exceptions are forwarded in calling chain (propagated).

#### Program of Exception Propagation

```
class TestExceptionPropagation1 {
    void m(){
        int data=50/0;
    }
    void n(){
        m();
    }
    void p(){
        try{
            n();
        } catch (Exception e){System.out.println("exception handled");}
    }
    public static void main(String args[]){
        TestExceptionPropagation1 obj=new TestExceptionPropagation1();
        obj.p();
        System.out.println("normal flow...");
    }
}
```

```
Output:exception handled
normal flow...
```



In the above example exception occurs in m() method where it is not handled, so it is propagated to previous n() method where it is not handled, again it is propagated to p() method where exception is handled.

Exception can be handled in any method in call stack either in main() method, p() method, n() method or m() method

**Rule:** By default, Checked Exceptions are not forwarded in calling chain (propagated).

#### Program which describes that checked exceptions are not propagated

```
class TestExceptionPropagation2{
    void m(){
        throw new java.io.IOException("device error");//checked exception
    }
    void n(){
        m();
    }
    void p(){
        try{
            n();
        }catch(Exception e){System.out.println("exception handled");}
    }
    public static void main(String args[]){
        TestExceptionPropagation2 obj=new TestExceptionPropagation2();
        obj.p();
        System.out.println("normal flow");
    }
}
```

Output:Compile Time Error

#### Java throws keyword

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

#### Syntax of java throws

```
return_type method_name() throws exception_class_name{
    //method code
}
```

#### Which exception should be declared

**Ans)** checked exception only, because:

- **unchecked Exception:** under your control so correct your code.
- **error:** beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

#### Advantage of Java throws keyword

**Now Checked Exception can be propagated** (forwarded in call stack).

It provides information to the caller of the method about the exception.



### Java throws example

Let's see the example of java throws clause which describes that checked exceptions can be propagated by throws keyword.

```
import java.io.IOException;
class Testthrows1{
    void m()throws IOException{
        throw new IOException("device error");//checked exception
    }
    void n()throws IOException{
        m();
    }
    void p(){
        try{
            n();
        }catch(Exception e){System.out.println("exception handled");}
    }
    public static void main(String args[]){
        Testthrows1 obj=new Testthrows1();
        obj.p();
        System.out.println("normal flow...");
    }
}
```

Output:  
exception handled  
normal flow...

Rule: If you are calling a method that declares an exception, you must either caught or declare the exception.

There are two cases:

1. **Case1:** You caught the exception i.e. handle the exception using try/catch.
2. **Case2:** You declare the exception i.e. specifying throws with the method.

### Case1: You handle the exception

- In case you handle the exception, the code will be executed fine whether exception occurs during the program or not.

```
import java.io.*;
class M{
    void method()throws IOException{
        throw new IOException("device error");
    }
}
public class Testthrows2{
    public static void main(String args[]){
        try{
            M m=new M();
            m.method();
        }catch(Exception e){System.out.println("exception handled");}

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

Output:exception handled  
normal flow...

### Case2: You declare the exception

- A) In case you declare the exception, if exception does not occur, the code will be executed fine.
- B) In case you declare the exception if exception occurs, an exception will be thrown at runtime because throws does not handle the exception.

A)Program if exception does not occur

```
import java.io.*;
class M{
    void method()throws IOException{
        System.out.println("device operation performed");
    }
}
class Testthrows3{
    public static void main(String args[])throws IOException{//declare exception
        M m=new M();
        m.method();
```

```
        System.out.println("normal flow...");
    } }
Output:device operation performed
normal flow...
```

B)Program if exception occurs

```
import java.io.*;
class M{
    void method()throws IOException{
        throw new IOException("device error");
    }
}
class Testthrows4{
    public static void main(String args[])throws IOException{//declare exception
        M m=new M();
        m.method();
        System.out.println("normal flow...");
    }
}
```

```
Output:Runtime Exception
Difference between throw and throws
Click me for details
```

Que) Can we rethrow an exception?

Yes, by throwing same exception in catch block.

Difference between throw and throws in Java

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

### Java throw example

```
void m(){  
throw new ArithmeticException("sorry");  
}
```

### Java throws example

```
void m()throws ArithmeticException{  
//method code  
}
```

### Java throw and throws example

```
void m()throws ArithmeticException{  
throw new ArithmeticException("sorry");  
}
```

### Difference between final, finally and finalize

There are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

No.	final	finally	finalize
1)	Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
2)	Final is a keyword.	Finally is a block.	Finalize is a method.

### Java final example

```
class FinalExample{  
public static void main(String[] args){  
final int x=100;  
x=200;//Compile Time Error  
}}}
```

### Java finally example

```
class FinallyExample{  
public static void main(String[] args){  
try{  
int x=300;  
}catch(Exception e){System.out.println(e);}  
finally{System.out.println("finally block is executed");}  
}}}
```

### Java finalize example

```
class FinalizeExample{  
public void finalize(){System.out.println("finalize called");}  
public static void main(String[] args){  
FinalizeExample f1=new FinalizeExample();  
FinalizeExample f2=new FinalizeExample();  
f1=null;  
f2=null;  
System.gc();  
}}}
```

### ExceptionHandling with MethodOverriding in Java

There are many rules if we talk about methodoverriding with exception handling. The Rules are as follows:

- If the superclass method does not declare an exception

- If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.
- **If the superclass method declares an exception**
  - If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

If the superclass method does not declare an exception

1) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception.

```
import java.io.*;
class Parent{
    void msg(){System.out.println("parent");}
}
class TestExceptionChild extends Parent{
    void msg()throws IOException{
        System.out.println("TestExceptionChild");
    }
    public static void main(String args[]){
        Parent p=new TestExceptionChild();
        p.msg();
    }
}
```

Output:Compile Time Error

2) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but can declare unchecked exception.

```
import java.io.*;
class Parent{
    void msg(){System.out.println("parent");}
}
class TestExceptionChild1 extends Parent{
    void msg()throws ArithmeticException{
        System.out.println("child");
    }
    public static void main(String args[]){
        Parent p=new TestExceptionChild1();
        p.msg();
    }
}
```

Output:child

If the superclass method declares an exception

1) Rule: If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

Example in case subclass overridden method declares parent exception

```
import java.io.*;
class Parent{
    void msg()throws ArithmeticException{System.out.println("parent");}
}
class TestExceptionChild2 extends Parent{
    void msg()throws Exception{System.out.println("child");}
```

```

public static void main(String args[]){
    Parent p=new TestExceptionChild2();
    try{
        p.msg();
    }catch(Exception e){ }
}

```

Output:Compile Time Error

#### Example in case subclass overridden method declares same exception

```

import java.io.*;
class Parent{
    void msg()throws Exception{System.out.println("parent");}
}

class TestExceptionChild3 extends Parent{
    void msg()throws Exception{System.out.println("child");}

    public static void main(String args[]){
        Parent p=new TestExceptionChild3();
        try{
            p.msg();
        }catch(Exception e){ }
    }
}

```

Output:child

#### Example in case subclass overridden method declares subclass exception

```

import java.io.*;
class Parent{
    void msg()throws Exception{System.out.println("parent");}
}

class TestExceptionChild4 extends Parent{
    void msg()throws ArithmeticException{System.out.println("child");}

    public static void main(String args[]){
        Parent p=new TestExceptionChild4();
        try{
            p.msg();
        }catch(Exception e){ }
    }
}

```

Output:child

#### Example in case subclass overridden method declares no exception

```

import java.io.*;
class Parent{
    void msg()throws Exception{System.out.println("parent");}
}

class TestExceptionChild5 extends Parent{
    void msg(){System.out.println("child");}
}

```

```

public static void main(String args[]){
    Parent p=new TestExceptionChild5();
    try{
        p.msg();
    }catch(Exception e){ }
}
}

```

Output:child

## 2B.4 Java Custom Exception

If you are creating your own Exception that is known as custom exception or user-defined exception. Java custom exceptions are used to customize the exception according to user need.

By the help of custom exception, you can have your own exception and message.

Let's see a simple example of java custom exception.

```

class InvalidAgeException extends Exception{
    InvalidAgeException(String s){
        super(s);
    }
}
class TestCustomException1{

    static void validate(int age)throws InvalidAgeException{
        if(age<18)
            throw new InvalidAgeException("not valid");
        else
            System.out.println("welcome to vote");
    }

    public static void main(String args[]){
        try{
            validate(13);
        }catch(Exception m){System.out.println("Exception occured: "+m);}

        System.out.println("rest of the code...");
    }
}

```

Output:Exception occured: InvalidAgeException:not valid  
rest of the code...