Java - OOPS

Java is an Object-Oriented Language. As a language that has the Object-Oriented feature, Java supports the following fundamental concepts −

* Polymorphism
* Inheritance
* Encapsulation
* Abstraction
* Classes
* Objects
* Instance
* Method
* Message Parsing

**Classes in Java**

A class is a blueprint from which individual objects are created.

[**Class** − A class can be defined as a template/blueprint that describes the behavior/state that the object of its type support.]

### Class in Java

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

**fields**

**methods**

**constructors**

**blocks**

**nested class and interface**

### Syntax to declare a class:

**class** <class\_name>{

    field;

    method;

}

### Instance variable in Java

A variable which is created inside the class but outside the method, is known as instance variable. Instance variable doesn't get memory at compile time. It gets memory at run time when object(instance) is created. That is why, it is known as instance variable.

### Method in Java

In java, a method is like functi

on i.e. used to expose behavior of an object.

#### Advantage of Method

Code Reusability

Code Optimization

### new keyword in Java

The new keyword is used to allocate memory at run time. All objects get memory in Heap memory area.

Following is a sample of a class.

Example

public class Dog {

String breed;

int age;

String color;

void barking()

{

System.out.println(“dog is barking”);

}

void hungry()

{

System.out.println(“dog is hungry”);

}

void sleeping()

{

System.out.println(“dog is Sleeping”);

}

}

A class can contain any of the following variable types.

* **Local variables**  (inside Method)− Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
* **Instance variables** (inside class without static)− Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
* **Class variables** (with static )− Class variables are variables declared within a class, outside any method, with the static keyword.

A class can have any number of methods to access the value of various kinds of methods. In the above example, barking(), hungry() and sleeping() are methods.

**Constructors**

When discussing about classes, one of the most important sub topic would be constructors. Every class has a constructor. If we do not explicitly write a constructor for a class, the Java compiler builds a default constructor for that class.

Each time a new object is created, at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.

Following is an example of a constructor −

Example

public class Puppy {

public Puppy() {

}

public Puppy(String name) {

// This constructor has one parameter, *name*.

}

}

Java also supports [Singleton Classes](https://www.tutorialspoint.com/java/java_using_singleton.htm) where you would be able to create only one instance of a class.

**Note** − We have two different types of constructors. We are going to discuss constructors in detail in the subsequent chapters.

### Object in Java



An entity that has state and behavior is known as an object e.g. chair, bike, marker, pen, table, car etc. It can be physical or logical (tangible and intangible). The example of intangible object is banking system.

* [**Object** − Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating. An object is an instance of a class.]

An object has three characteristics:

**state:** represents data (value) of an object.

**behavior:** represents the behavior (functionality) of an object such as deposit, withdraw etc.

**identity:** Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But, it is used internally by the JVM to identify each object uniquely.

For Example: Pen is an object. Its name is Reynolds, color is white etc. known as its state. It is used to write, so writing is its behavior.

**Object is an instance of a class.** Class is a template or blueprint from which objects are created. So object is the instance(result) of a class.

**Object Definitions:**

Object is a real world entity.

Object is a run time entity.

Object is an entity which has state and behavior.

Object is an instance of a class.

**Accessing Instance Variables and Methods**

Instance variables and methods are accessed via created objects. To access an instance variable, following is the fully qualified path −

/\* First create an object \*/

ObjectReference = new Constructor();

/\* Now call a variable as follows \*/

ObjectReference.variableName;

/\* Now you can call a class method as follows \*/

ObjectReference.MethodName();

Example

This example explains how to access instance variables and methods of a class.

public class Puppy

{

int puppyAge;

public Puppy(String name)

{

// This constructor has one parameter, *name*.

System.out.println("Name chosen is :" + name );

}

public void setAge( int age )

{ puppyAge = age; }

public int getAge( )  
 { System.out.println("Puppy's age is :" + puppyAge );

return puppyAge;

}

public static void main(String []args) {

/\* Object creation \*/

Puppy myPuppy = new Puppy( "tommy" );

/\* Call class method to set puppy's age \*/

myPuppy.setAge( 2 );

/\* Call another class method to get puppy's age \*/

myPuppy.getAge( );

/\* You can access instance variable as follows as well \*/

System.out.println("Variable Value :" + myPuppy.puppyAge );

}

}

If we compile and run the above program, then it will produce the following result −

Output

Name chosen is :tommy

Puppy's age is :2

Variable Value :2

### Object and Class Example: main within class

In this example, we have created a Student class that have two data members id and name. We are creating the object of the Student class by new keyword and printing the objects value.

Here, we are creating main() method inside the class.

*File: Student.java*

**class** Student{

**int** id=20;//field or data member or instance variable

 String name=”msc”;

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

  Student s1=**new** Student();//creating an object of Student

  System.out.println(s1.id);//accessing member through reference variable

  System.out.println(s1.name);

 }

}

Output:

0

null

### Object and Class Example: main outside class

In real time development, we create classes and use it from another class. It is a better approach than previous one. Let's see a simple example, where we are having main() method in another class.

We can have multiple classes in different java files or single java file. If you define multiple classes in a single java source file, it is a good idea to save the file name with the class name which has main() method.

*File: TestStudent1.java*

**class** Student

{

**int** id=90;

 String name=”msc”;

Void getdata()

{}

Void putdata(){}

}

**class** TestStudent1

{

**public** **static** **void** main(String args[])

{

  Student s1=**new** Student();

 // System.out.println(s1.id);

 // System.out.println(s1.name);

S1.getdata();

S1.putdata();

 }

}

Output:

0

null

## 3 Ways to initialize object

There are 3 ways to initialize object in java.

By reference variable

By method

By constructor

### Object and Class Example: Initialization through reference

Initializing object simply means storing data into object. Let's see a simple example where we are going to initialize object through reference variable.

*File: TestStudent2.java*

**class** Student{

**int** id;

 String name;

}

**class** TestStudent2{

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

  Student s1=**new** Student();

  s1.id=101;

  s1.name="Sonoo";

  System.out.println(s1.id+" "+s1.name);//printing members with a white space

 }

}

Output:

101 Sonoo

We can also create multiple objects and store information in it through reference variable.

*File: TestStudent3.java*

**class** Student{

**int** id;

 String name;

}

**class** TestStudent3{

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

  //Creating objects

  Student s1=**new** Student();

  Student s2=**new** Student();

  //Initializing objects

  s1.id=101;

  s1.name="Sonoo";

  s2.id=102;

  s2.name="Amit";

  //Printing data

  System.out.println(s1.id+" "+s1.name);

  System.out.println(s2.id+" "+s2.name);

 }

}

Output:

101 Sonoo

102 Amit

### 2) Object and Class Example: Initialization through method

In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

*File: TestStudent4.java*

**class** Student{

**int** rollno;

 String name;

  student(**int** r, String n){

  rollno=r;

  name=n;

 }

**void** display(){System.out.println(rollno+" "+name);}

}

**class** TestStudent4

{

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

  Student s1=**new** Student(111,"Karan");  );

  Student s2=**new** Student(222,"Aryan");  );

    s1.display();

  s2.display();

 }

}

Output:

111 Karan

222 Aryan



As you can see in the above figure, object gets the memory in heap memory area. The reference variable refers to the object allocated in the heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

### 3) Object and Class Example: Initialization through constructor

We will learn about constructors in java later.

### Object and Class Example: Employee

Let's see an example where we are maintaining records of employees.

*File: TestEmployee.java*

**class** Employee{

**int** id;

    String name;

**float** salary;

**void** insert(**int** i, String n, **float** s) {

        id=i;

        name=n;

        salary=s;

    }

**void** display(){System.out.println(id+" "+name+" "+salary);}

}

**public** **class** TestEmployee {

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

    Employee e1=**new** Employee();

    Employee e2=**new** Employee();

    Employee e3=**new** Employee();

    e1.insert(101,"ajeet",45000);

    e2.insert(102,"irfan",25000);

    e3.insert(103,"nakul",55000);

    e1.display();

    e2.display();

    e3.display();

}

}

Output:

101 ajeet 45000.0

102 irfan 25000.0

103 nakul 55000.0

### Object and Class Example: Rectangle

There is given another example that maintains the records of Rectangle class.

*File: TestRectangle1.java*

**class** Rectangle{

**int** length;

**int** width;

**void** insert(**int** l, **int** w){

  length=l;

  width=w;

 }

**void** calculateArea(){System.out.println(length\*width);}

}

**class** TestRectangle1{

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

  Rectangle r1=**new** Rectangle();

  Rectangle r2=**new** Rectangle();

  r1.insert(11,5);

  r2.insert(3,15);

  r1.calculateArea();

  r2.calculateArea();

}

}

Output:

55

45

## What are the different ways to create an object in Java?

There are many ways to create an object in java. They are:

By new keyword

By newInstance() method

By clone() method

By deserialization

By factory method etc.

We will learn these ways to create object later.

## Anonymous object

Anonymous simply means nameless. An object which has no reference is known as anonymous object. It can be used at the time of object creation only.

If you have to use an object only once, anonymous object is a good approach. For example:

**new** Calculation();//anonymous object

Calling method through reference:

Calculation c=**new** Calculation();

c.fact(5);

Calling method through anonymous object

**new** Calculation().fact(5);

Let's see the full example of anonymous object in java.

**class** Calculation{

**void** fact(**int**  n){

**int** fact=1;

**for**(**int** i=1;i<=n;i++){

   fact=fact\*i;

  }

 System.out.println("factorial is "+fact);

}

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

**new** Calculation().fact(5);//calling method with anonymous object

}

}

Output:

Factorial is 120

### Creating multiple objects by one type only

We can create multiple objects by one type only as we do in case of primitives.

Initialization of primitive variables:

**int** a=10, b=20;

Initialization of refernce variables:

Rectangle r1=**new** Rectangle(), r2=**new** Rectangle();//creating two objects

Let's see the example:

**class** Rectangle{

**int** length;

**int** width;

**void** insert(**int** l,**int** w){

  length=l;

  width=w;

 }

**void** calculateArea(){System.out.println(length\*width);}

}

**class** TestRectangle2{

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

  Rectangle r1=**new** Rectangle(),r2=**new** Rectangle();//creating two objects

  r1.insert(11,5);

  r2.insert(3,15);

  r1.calculateArea();

  r2.calculateArea();

}

}

Output:

55

45

### Real World Example: Account

*File: TestAccount.java*

**class** Account{

**int** acc\_no;

String name;

**float** amount;

**void** insert(**int** a,String n,**float** amt){

acc\_no=a;

name=n;

amount=amt;

}

**void** deposit(**float** amt){

amount=amount+amt;

System.out.println(amt+" deposited");

}

**void** withdraw(**float** amt){

**if**(amount<amt){

System.out.println("Insufficient Balance");

}**else**{

amount=amount-amt;

System.out.println(amt+" withdrawn");

}

}

**void** checkBalance(){System.out.println("Balance is: "+amount);}

**void** display(){System.out.println(acc\_no+" "+name+" "+amount);}

}

**class** TestAccount{

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

Account a1=**new** Account();

a1.insert(832345,"Ankit",1000);

a1.display();

a1.checkBalance();

a1.deposit(40000);

a1.checkBalance();

a1.withdraw(15000);

a1.checkBalance();

}}

Output:

832345 Ankit 1000.0

Balance is: 1000.0

40000.0 deposited

Balance is: 41000.0

15000.0 withdrawn

Balance is: 26000.0

# Inheritance in Java

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

* **Super Class:**The class whose features are inherited is known as super class(or a base class or a parent class).
* **Sub Class:** The class that inherits the other class is known as sub class(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
* **Reusability:**Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

**How to use inheritance in Java**

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

class derived-class extends base-class

{

//methods and fields

}

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

|  |
| --- |
| //Java program to illustrate the  // concept of inheritance    // base class  class Bicycle  {      // the Bicycle class has two fields      public int gear;      public int speed;        // the Bicycle class has one constructor      public Bicycle(int gear, int speed)      {          this.gear = gear;          this.speed = speed;      }        // the Bicycle class has three methods      public void applyBrake(int decrement)      {          speed -= decrement;      }        public void speedUp(int increment)      {          speed += increment;      }        // toString() method to print info of Bicycle      public String toString()      {          return("No of gears are "+gear                  +"\n"                  + "speed of bicycle is "+speed);      }  }    // derived class  class MountainBike extends Bicycle  {        // the MountainBike subclass adds one more field      public int seatHeight;        // the MountainBike subclass has one constructor      public MountainBike(int gear,int speed,                          int startHeight)      {          // invoking base-class(Bicycle) constructor          super(gear, speed);          seatHeight = startHeight;      }        // the MountainBike subclass adds one more method      public void setHeight(int newValue)      {          seatHeight = newValue;      }        // overriding toString() method      // of Bicycle to print more info      @Override      public String toString()      {          return (super.toString()+                  "\nseat height is "+seatHeight);      }    }    // driver class  public class Test  {      public static void main(String args[])      {            MountainBike mb = new MountainBike(3, 100, 25);          System.out.println(mb.toString());        }  } |

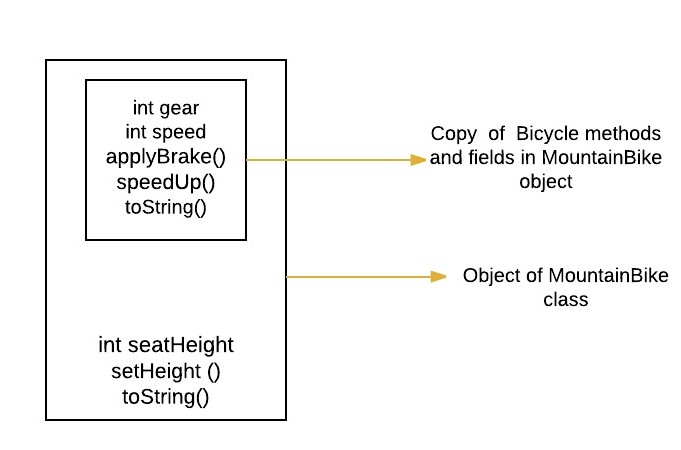
Run on IDE

Output:

No of gears are 3

speed of bicycle is 100

seat height is 25

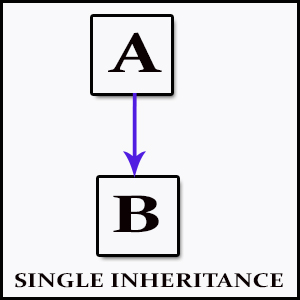
In above program, when an object of MountainBike class is created, a copy of the all methods and fields of the superclass acquire memory in this object. That is why, by using the object of the subclass we can also access the members of a superclass.  
Please note that during inheritance only object of subclass is created, not the superclass. For more, refer [Java Object Creation of Inherited Class](http://www.geeksforgeeks.org/gfact-52-java-object-creation-of-inherited-classes/).  
**Illustrative image of the program:**  
[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/f.jpeg)

In practice, inheritance and [polymorphism](http://www.geeksforgeeks.org/overriding-in-java/) are used together in java to achieve fast performance and readability of code.

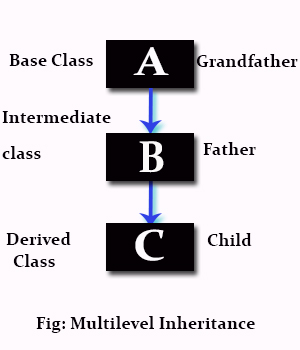
**Types of Inheritance in Java**

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

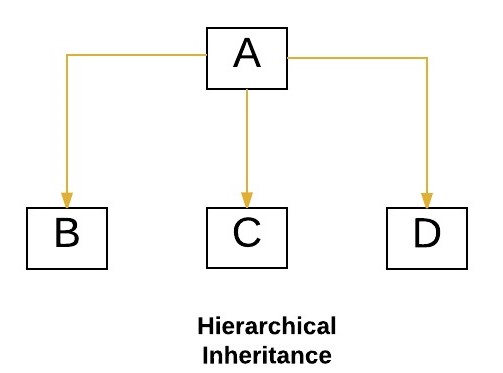
* 1. **Single Inheritance :**In single inheritance, subclasses inherit the features of one superclass. In image below, the class A serves as a base class for the derived class B.

[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/Single_Inheritance.jpg)

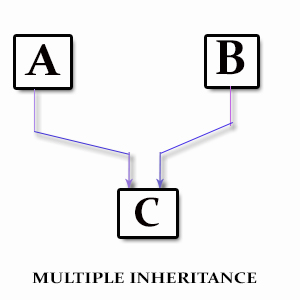
* 1. **Multilevel Inheritance :**In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In below image, the class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the[grandparent’s members](http://www.geeksforgeeks.org/g-fact-91/).

[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/Multilevel_Inheritance.jpg)

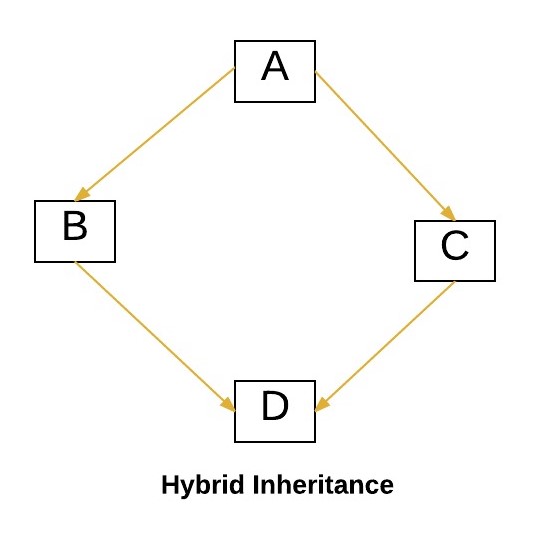
* 1. **Hierarchical Inheritance :**In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one sub class.In below image, the class A serves as a base class for the derived class B,C and D.

[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/hie.jpeg)

* 1. [**Multiple Inheritance**](http://www.geeksforgeeks.org/java-and-multiple-inheritance/)**(Through Interfaces) :**In Multiple inheritance ,one class can have more than one superclass and inherit features from all parent classes. Please note that Java does **not** support [multiple inheritance](http://www.geeksforgeeks.org/java-and-multiple-inheritance/) with classes. In java, we can achieve multiple inheritance only through [Interfaces](http://quiz.geeksforgeeks.org/interfaces-in-java/). In image below, Class C is derived from interface A and B.

[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/Multiple_Inheritance.jpg)

* 1. **Hybrid Inheritance(Through Interfaces) :**It is a mix of two or more of the above types of inheritance. Since java doesn’t support multiple inheritance with classes, the hybrid inheritance is also not possible with classes. In java, we can achieve hybrid inheritance only through [Interfaces](http://quiz.geeksforgeeks.org/interfaces-in-java/).

[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/hybrid.jpeg)

**Important facts about inheritance in Java**

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

**What all can be done in a Subclass?**

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

* The inherited fields can be used directly, just like any other fields.
* We can declare new fields in the subclass that are not in the superclass.
* The inherited methods can be used directly as they are.
* We can write a new instance method in the subclass that has the same signature as the one in the superclass, thus [overriding](http://www.geeksforgeeks.org/overriding-in-java/) it (as in example above, toString() method is overridden).
* We can write a new static method in the subclass that has the same signature as the one in the superclass, thus [hiding](http://www.geeksforgeeks.org/g-fact-63/) it.
* We can declare new methods in the subclass that are not in the superclass.
* We can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword [super](http://quiz.geeksforgeeks.org/super-keyword/).

# Inheritance in Java

1. [Inheritance](https://www.javatpoint.com/inheritance-in-java)
2. [Types of Inheritance](https://www.javatpoint.com/inheritance-in-java#inheritancetypes)
3. [Why multiple inheritance is not possible in java in case of class?](https://www.javatpoint.com/inheritance-in-java#inheritancenotmultiple)

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

The idea behind inheritance in java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of 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.

### Syntax of Java Inheritance

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

{

   //methods and fields

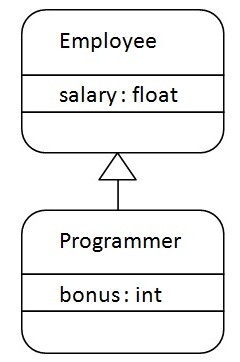
// body of subclass

}

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);

}

}

Programmer salary is:40000.0

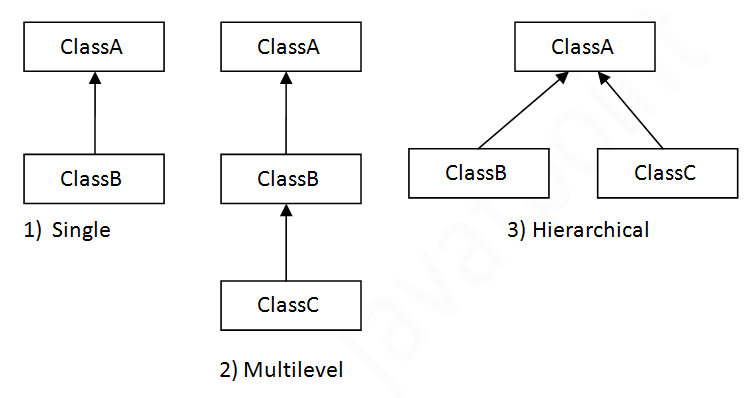
Bonus of programmer is:10000

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

## Types of inheritance in java

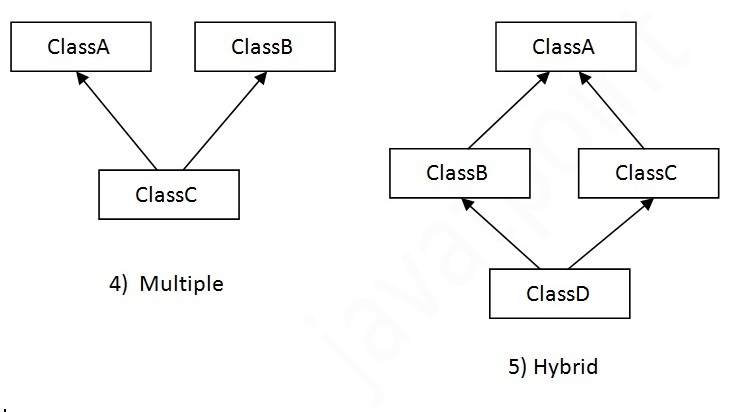
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

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



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

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



## 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...

## 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?

}

}

Compile Time Error

**interface**

An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.

Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

An interface is similar to a class in the following ways −

* An interface can contain any number of methods.
* An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
* The byte code of an interface appears in a **.class** file.
* Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including −

* You cannot instantiate an interface.
* An interface does not contain any constructors.
* All of the methods in an interface are abstract.
* An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
* An interface is not extended by a class; it is implemented by a class.
* An interface can extend multiple interfaces.

**Declaring Interfaces**

The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface −

Example

Following is an example of an interface −

/\* File name : NameOfInterface.java \*/

import java.lang.\*;

// Any number of import statements

public interface NameOfInterface {

// Any number of final, static fields

// Any number of abstract method declarations\

}

Interfaces have the following properties −

* An interface is implicitly abstract. You do not need to use the **abstract** keyword while declaring an interface.
* Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
* Methods in an interface are implicitly public.

Example

/\* File name : Animal.java \*/

interface Animal {

public void eat();

public void travel();

}

**Implementing Interfaces**

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

Example

/\* 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();

}

}

This will produce the following result −

Output

Mammal eats

Mammal travels

When overriding methods defined in interfaces, there are several rules to be followed −

* Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.
* The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
* An implementation class itself can be abstract and if so, interface methods need not be implemented.

When implementation interfaces, there are several rules −

* A class can implement more than one interface at a time.
* A class can extend only one class, but implement many interfaces.
* An interface can extend another interface, in a similar way as a class can extend another class.

Extending Interfaces

An interface can extend another interface in the same way that a class can extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

The following Sports interface is extended by Hockey and Football interfaces.

Example

// Filename: Sports.java

public interface Sports {

public void setHomeTeam(String name);

public void setVisitingTeam(String name);

}

// Filename: Football.java

public interface Football extends Sports {

public void homeTeamScored(int points);

public void visitingTeamScored(int points);

public void endOfQuarter(int quarter);

}

// Filename: Hockey.java

public interface Hockey extends Sports {

public void homeGoalScored();

public void visitingGoalScored();

public void endOfPeriod(int period);

public void overtimePeriod(int ot);

}

The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

Extending Multiple Interfaces

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.

The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

For example, if the Hockey interface extended both Sports and Event, it would be declared as −

Example

public interface Hockey extends Sports, Event

Multiple Inheritance Using Interface Example Java Program

Definition

Inheritance is when an object or class is based on another object or class, using the same implementation specifying implementation to maintain the same behavior. It is a mechanism for code reuse and to allow independent extensions of the original software via public classes and interfaces. The relationships of objects or classes through inheritance give rise to a hierarchy. Multiple Inheritance allows a class to have more than one super class and to inherit features from all parent class. it is achieved using interface.

Syntax

public interface **A**{

*//Do Something*

}

public interface **B** extends **A**{

*//Do Something*

}

public interface **C** extends **A**{

*//Do Something*

}

Multiple Inheritance Using Interface Example Program

interface **vehicleone**{

int speed=90;

public void **distance**();

}

interface **vehicletwo**{

int distance=100;

public void **speed**();

}

class **Vehicle** implements **vehicleone**,**vehicletwo**{

public void **distance**(){

int distance=speed\*100;

System.out.println("distance travelled is "+distance);

}

public void **speed**(){

int speed=distance/100;

}

}

class **MultipleInheritanceUsingInterface**{

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

System.out.println("Vehicle");

obj.distance();

obj.speed();

}

}

**What are default methods?**

Default methods enable you to add new functionality to the interfaces and ensure backward compatibility for existing classes which implement that interface.

As their name implies, default methods in interfaces are methods which will be invoked by default – if not overridden in implementing classes. Let’s understand with an example.

Moveable interface is some existing interface and wants to add a new method moveFast(). If it adds moveFast() method using old technique, then all classes implemeting Moveable will also be changed. So, let’s add moveFast() method as default method.

|  |
| --- |
| public interface Moveable  {      default void moveFast()      {          System.out.println("I am moving fast, buddy !!");      }  } |

If all classes implementing Moveable interface do not need change themselves (until some class specifically wants to override moveFast() method to add custom logic). All classes can directly call instance.moveFast() method.

|  |
| --- |
| public class Animal implements Moveable  {      public static void main(String[] args)      {          Animal tiger = new Animal();            //Call default method using instance reference          tiger.moveFast();      }  } |

# Multiple Inheritance in Java Example

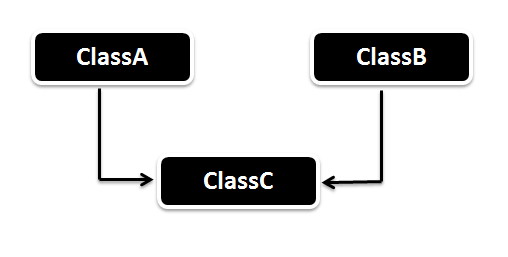
August 6, 2015 by [**javainterviewpoint**](http://www.javainterviewpoint.com/author/javainterviewpoint/) [**Leave a Comment**](http://www.javainterviewpoint.com/multiple-inheritance-in-java-with-example/#respond)

**Multiple**[**Inheritance**](http://www.javainterviewpoint.com/types-of-inheritance-in-java-singlemultiplemultilevelhierarchical-hybrid/) in Java is nothing but **one** class **extending** **more** **than** **one class**. [**Java**](https://www.javainterviewpoint.com/category/core-java/) does **not** have this capability. As the designers considered that multiple inheritance will to be too complex to manage, but indirectly you can achieve Multiple [**Inheritance in Java**](http://www.javainterviewpoint.com/inheritance-in-java/)using [**Interfaces**](http://www.javainterviewpoint.com/interface-java/).

As in Java we can implement more than one interface we achieve the same effect using interfaces.

### ****Flow Diagram****

Conceptually Multiple Inheritance has to be like the below diagram, **ClassA**and**ClassB** both inherited by **ClassC**. Since it is not supported we will changing the **ClassA to InterfaceA** and **ClassB to InterfaceB**.

**[](http://javainterviewpoint-7ac9.kxcdn.com/wp-content/uploads/2015/07/Multiple_Inheritance_in_Java.png)**

### ****Example of Multiple Inheritance****

Here we have two interfaces **Car** and **Bus.**

* **Car** interface has a attribute **speed** and a method defined **distanceTravelled()**
* **Bus** interface has a attribute **distance** and method **speed()**

The **Vehicle class** implements both interface **Car and Bus** and provides implementation.

package com.javainterviewpoint.inheritance;

interface Car

{

int speed=60;

public void distanceTravelled();

}

interface Bus

{

int distance=100;

public void speed();

}

public class Vehicle implements Car,Bus

{

int distanceTravelled;

int averageSpeed;

public void distanceTravelled()

{

distanceTravelled=speed\*distance;

System.out.println("Total Distance Travelled is : "+distanceTravelled);

}

public void speed()

{

int averageSpeed=distanceTravelled/speed;

System.out.println("Average Speed maintained is : "+averageSpeed);

}

public static void main(String args[])

{

Vehicle v1=new Vehicle();

v1.distanceTravelled();

v1.speed();

}

}

**Output :**

Total Distance Travelled is : 6000

Average Speed maintained is : 100

In the above code we doesn’t have ambiguity even when we **use classes instead of interfaces**, then there comes the question **why Java is not supporting ?**. The problem arises when both the classes has the **same method in it** **?** and the compiler will not know which method to call whereas the methods of the interfaces are by default [**abstract**](http://www.javainterviewpoint.com/abstract-class-java/) and implementations are not provided by the interface and hence we can avoid the ambiguity.

package com.javainterviewpoint.inheritance;

interface InterfaceA

{

public void disp();

}

interface InterfaceB

{

public void disp();

}

public class Client implements InterfaceA,InterfaceB

{

@Override

public void disp()

{

System.out.println("disp() method implementation");

}

public static void main(String args[])

{

Client c = new Client();

c.disp();

}

}

**Output :**

disp() method implementation

As we can see in the above code the **Client class** has implemented both the **interfaces InterfaceA and InterfaceB.** In this case we **didn’t** have **ambiguity** even though both the interfaces are having **same method**.

#### ****Other interesting articles****

**How multiple inheritance is achieved via default methods?**

Multiple inheritance is a feature of some object-oriented computer programming languages in which an object or class can inherit characteristics and behavior from more than one parent object or parent class.

We know that in java (until jdk 7), inheritence in java was supported by extendskeyword which is used to create a child class from a parent class. You cannot extend from two classes.

Until java 7, interfaces were only for declaring the contracts which implementing classes MUST implement (except the implementing class in not abstract itself). So there was no specific behavior attached with interfaces which a class can inherit. So, even after a class was capable of implementing as many interfaces as it want, it was not appropriate to term as multiple inheritance.

But since java 8’s default methods, interfaces have behavior as well. So now **if a class implement two interfaces and both defines default methods, then it is essentially inheriting behaviors from two parents which is multiple inheritance**.

For example, in below code Animal class does not define any of it’s own behavior; rather it is inheriting behavior from parent interfaces. That’s multiple inheritance.

|  |
| --- |
| package com.howtodoinjava.examples;    interface Moveable  {      default void moveFast(){          System.out.println("I am moving fast, buddy !!");      }  }    interface Crawlable  {      default void crawl(){          System.out.println("I am crawling !!");      }  }    public class Animal implements Moveable, Crawlable  {      public static void main(String[] args)      {          Animal self = new Animal();            self.moveFast();          self.crawl();      }  } |

**Possible conflicts and their resolutions**

In above example, we have two different interfaces and two different methods – so there is no conflict. What if **both interfaces decide to define a new method with same name**. Well they can define without problem. But which method will be invoked when Animal instance will call it’s name. That’s conflicting situation.

|  |
| --- |
| package com.howtodoinjava.examples;    interface Moveable  {      default void run(){          System.out.println("I am running, kid !!");      }  }    interface Crawlable  {      default void run(){          System.out.println("I am running, daddy !!");      }  }    public class Animal implements Moveable, Crawlable  {      public static void main(String[] args)      {          Animal self = new Animal();            //What will happen when below statement will execute          //self.run();      }  } |

So solve above conflict, caller class must decide which run() method it want to invoke and then call **using interface’s reference** like below.

|  |
| --- |
| Moveable.super.run();   //Call Moveable's run() method    //or    Crawlable.super.run();  //Call Crawlable's run() method |

**super** is a keyword.

* It is used inside a sub-class method definition to call a method defined in the super class. Private methods of the super-class cannot be called. Only public and protected methods can be called by the **super** keyword.
* It is also used by class constructors to invoke constructors of its [parent class](https://en.wikipedia.org/wiki/parent_class).

Syntax:

**super**.<method-name>([zero or more arguments]);

or:

**super**([zero or more arguments]);

For example:

|  |  |
| --- | --- |
| Computer code | 1 **public** **class** **SuperClass** {  2 **public** void printHello() {  3 System.out.println("Hello from SuperClass");  4 **return**;  5 }  6 } |

|  |  |
| --- | --- |
| Computer code | 1 **public** **class** **SubClass** **extends** SuperClass {  2 **public** void printHello() {  3 **super**.printHello();  4 System.out.println("Hello from SubClass");  5 **return**;  6 }  7 **public** **static** main(String[] args) {  8 SubClass obj = **new** SubClass();  9 obj.printHello();  10 }  11 } |

Running the above program:

|  |  |
| --- | --- |
| Computer code | **Command for Code listing 2**  $Java SubClass |

|  |  |
| --- | --- |
| Computer code | **Output of Code listing 2**  Hello from SuperClass  Hello from SubClass |

# 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();

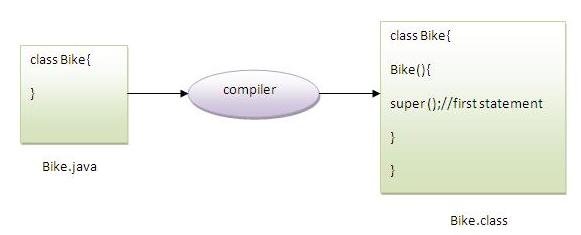
}}

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.**

1. **class** Animal{
2. Animal(){System.out.println("animal is created");}
3. }
4. **class** Dog **extends** Animal{
5. Dog(){
6. System.out.println("dog is created");
7. }
8. }
9. **class** TestSuper4{
10. **public** **static** **void** main(String args[]){
11. Dog d=**new** Dog();
12. }}

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

Abstract class in java is similar to interface except that it can contain default method implementation. An abstract class can have abstract method without body and it can have methods with implementation also.

abstract keyword is used to create a abstract class and method. Abstract class in java can’t be instantiated. Abstract class is mostly used to provide base for subclasses to extend and implement the abstract methods and override or use the implemented methods in abstract class.

**Abstract Class in Java**

[](https://cdn.journaldev.com/wp-content/uploads/2013/07/abstract-class-in-java.jpg)

Here is a simple example of Abstract Class in Java.

package com.journaldev.design;

//abstract class

public abstract class Person {

private String name;

private String gender;

public Person(String nm, String gen){

this.name=nm;

this.gender=gen;

}

//abstract method

public abstract void work();

@Override

public String toString(){

return "Name="+this.name+"::Gender="+this.gender;

}

public void changeName(String newName) {

this.name = newName;

}

}

Notice that work() is an abstract method and it has no body. Here is a concrete class example extending abstract class in java.

package com.journaldev.design;

public class Employee extends Person {

private int empId;

public Employee(String nm, String gen, int id) {

super(nm, gen);

this.empId=id;

}

@Override

public void work() {

if(empId == 0){

System.out.println("Not working");

}else{

System.out.println("Working as employee!!");

}

}

public static void main(String args[]){

//coding in terms of abstract classes

Person student = new Employee("Dove","Female",0);

Person employee = new Employee("Pankaj","Male",123);

student.work();

employee.work();

//using method implemented in abstract class - inheritance

employee.changeName("Pankaj Kumar");

System.out.println(employee.toString());

}

}

### Abstract class in Java Important Points

1. abstract keyword is used to create an abstract class in java.
2. Abstract class in java can’t be instantiated.
3. We can use abstract keyword to create an abstract method, an abstract method doesn’t have body.
4. If a class have abstract methods, then the class should also be abstract using abstract keyword, else it will not compile.
5. It’s not necessary to have abstract class to have abstract method.
6. If abstract class doesn’t have any method implementation, its better to use interface because java doesn’t support multiple class inheritance.
7. The subclass of abstract class in java must implement all the abstract methods unless the subclass is also an abstract class.
8. All the methods in an interface are implicitly abstract unless the interface methods are static or default. Static methods and default methods in interfaces are added in Java 8, for more details read [Java 8 interface changes](https://www.journaldev.com/2752/java-8-interface-changes-static-method-default-method).
9. Java Abstract class can implement interfaces without even providing the implementation of interface methods.
10. Java Abstract class is used to provide common method implementation to all the subclasses or to provide default implementation.
11. We can run abstract class in java like any other class if it has main() method.

# 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. It implementation is provided by the Honda class.

1. **abstract** **class** Bike{
2. **abstract** **void** run();
3. }
4. **class** Honda4 **extends** Bike{
5. **void** run(){System.out.println("running safely..");}
6. **public** **static** **void** main(String args[]){
7. Bike obj = **new** Honda4();  //derived class reference to the base class
8. obj.run();
9. }
10. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Honda4)

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*

1. **abstract** **class** Shape{
2. **abstract** **void** draw();
3. }
4. //In real scenario, implementation is provided by others i.e. unknown by end user
5. **class** Rectangle **extends** Shape{
6. **void** draw(){System.out.println("drawing rectangle");}
7. }
8. **class** Circle1 **extends** Shape{
9. **void** draw(){System.out.println("drawing circle");}
10. }
11. //In real scenario, method is called by programmer or user
12. **class** TestAbstraction1{
13. **public** **static** **void** main(String args[]){
14. Shape s=**new** Circle1();//In real scenario, object is provided through method e.g. getShape() method
15. s.draw();
16. }
17. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestAbstraction1)

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*

1. //example of abstract class that have method body
2. **abstract** **class** Bike{
3. Bike(){System.out.println("bike is created");}
4. **abstract** **void** run();
5. **void** changeGear(){System.out.println("gear changed");}
6. }
8. **class** Honda **extends** Bike{
9. **void** run(){System.out.println("running safely..");}
10. }
11. **class** TestAbstraction2{
12. **public** **static** **void** main(String args[]){
13. Bike obj = **new** Honda();
14. obj.run();
15. obj.changeGear();
16. }
17. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestAbstraction2)

bike is created

running safely..

gear changed

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

1. **class** Bike12{
2. **abstract** **void** run();
3. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Bike12)

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.

1. **interface** A{
2. **void** a();
3. **void** b();
4. **void** c();
5. **void** d();
6. }
8. **abstract** **class** B **implements** A{
9. **public** **void** c(){System.out.println("I am C");}
10. }
12. **class** M **extends** B{
13. **public** **void** a(){System.out.println("I am a");}
14. **public** **void** b(){System.out.println("I am b");}
15. **public** **void** d(){System.out.println("I am d");}
16. }
18. **class** Test5{
19. **public** **static** **void** main(String args[]){
20. A a=**new** M();
21. a.a();
22. a.b();
23. a.c();
24. a.d();
25. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Test5)

Output:I am a

I am b

I am c

I am d

# Final Keyword In Java

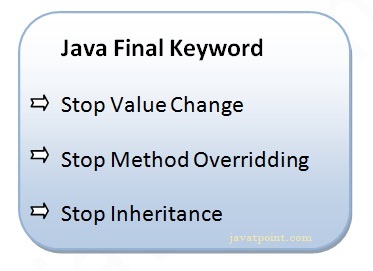
1. [Final variable](https://www.javatpoint.com/final-keyword#finalv)
2. [Final method](https://www.javatpoint.com/final-keyword#finalm)
3. [Final class](https://www.javatpoint.com/final-keyword#finalc)
4. [Is final method inherited ?](https://www.javatpoint.com/final-keyword#finalisinherited)
5. [Blank final variable](https://www.javatpoint.com/final-keyword#finalblank)
6. [Static blank final variable](https://www.javatpoint.com/final-keyword#finalstaticblank)
7. [Final parameter](https://www.javatpoint.com/final-keyword#finalpara)
8. [Can you declare a final constructor](https://www.javatpoint.com/final-keyword#finalcons)

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

**class** Bike9{

**final** **int** speedlimit=90;//final variable  // to remove final keyword

**void** run(){

  speedlimit=400;  // System.out.println(speedlimit);

// int max= speedlimit+10;

// System.out.println(max);

 }

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

 Bike9 obj=**new**  Bike9();

 obj.run();

 }

}//end of class

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Bike9)

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();

   }

}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Honda)

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** Honda();

  honda.run();

  }

}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Honda1)

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();

   }

}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Honda2)

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:

1. **class** Bike10{
2. **final** **int** speedlimit;//blank final variable
4. Bike10(){
5. speedlimit=70;
6. System.out.println(speedlimit);
7. }
9. **public** **static** **void** main(String args[]){
10. **new** Bike10();
11. }
12. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=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

1. **class** A{
2. **static** **final** **int** data;//static blank final variable
3. **static**{ data=50;}
4. **public** **static** **void** main(String args[]){
5. System.out.println(A.data);
6. }
7. }

### 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);

 }

}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Bike11)

Output:Compile Time Error

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

No, because constructor is never inherited.

# Java Package

1. [Java Package](https://www.javatpoint.com/package)
2. [Example of package](https://www.javatpoint.com/package#packageex)
3. [Accessing package](https://www.javatpoint.com/package#packageaccess)
   1. [By import packagename.\*](https://www.javatpoint.com/package#packageaccess1)
   2. [By import packagename.classname](https://www.javatpoint.com/package#packageaccess2)
   3. [By fully qualified name](https://www.javatpoint.com/package#packageaccess3)
4. [Subpackage](https://www.javatpoint.com/package#packagesub)
5. [Sending class file to another directory](https://www.javatpoint.com/package#packageanotherdirectory)
6. [-classpath switch](https://www.javatpoint.com/package#packageclasspathswitch)
7. [4 ways to load the class file or jar file](https://www.javatpoint.com/package#packagewaystoload)
8. [How to put two public class in a package](https://www.javatpoint.com/package#packagetwopublic)
9. [Static Import](https://www.javatpoint.com/package#packagestaticimport)
10. [Package class](https://www.javatpoint.com/package-class)

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.

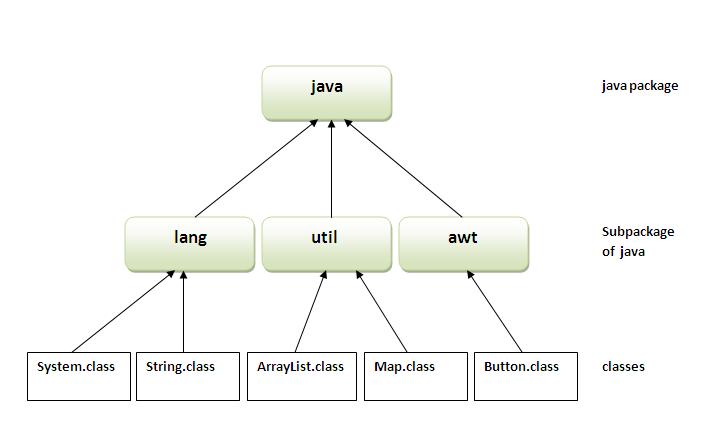
Here, we will have the detailed learning of creating and using user-defined packages.

## 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.



## Simple example of java package

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

1. //save as Simple.java
2. **package** mypack;
3. **public** **class** Simple{
4. **public** **static** **void** main(String args[]){
5. System.out.println("Welcome to package");
6. }
7. }

## 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; ex: **import** pack.A;
3. fully qualified name.

pack.A obj = **new** pack.A();//using 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.\*

1. //save by A.java
2. **package** pack;
3. **public** **class** A{
4. **public** **void** msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. **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

1. //save by A.java
3. **package** pack;
4. **public** **class** A{
5. **public** **void** msg(){System.out.println("Hello");}
6. }
7. //save by B.java
8. **package** mypack;
9. **import** pack.A;
11. **class** B{
12. **public** **static** **void** main(String args[]){
13. A obj = **new** A();
14. obj.msg();
15. }
16. }

Output:Hello

#### 3) Using fully qualified name

If you use fu

lly 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

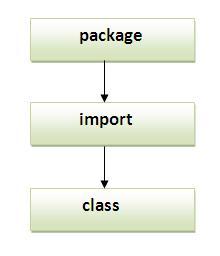
1. //save by A.java
2. **package** pack;
3. **public** **class** A{
4. **public** **void** msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. **package** mypack;
8. **class** B{
9. **public** **static** **void** main(String args[]){
10. pack.A obj = **new** pack.A();//using fully qualified name
11. obj.msg();
12. }
13. }

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 Microsystem has definded 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

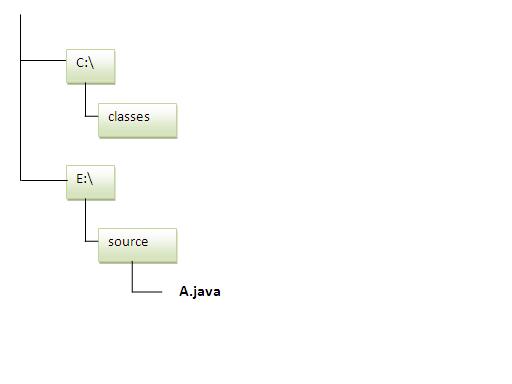
1. **package** com.javatpoint.core;
2. **class** Simple{
3. **public** **static** **void** main(String args[]){
4. System.out.println("Hello subpackage");
5. }
6. }

|  |
| --- |
| **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?

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



1. //save as Simple.java
2. **package** mypack;
3. **public** **class** Simple{
4. **public** **static** **void** main(String args[]){
5. System.out.println("Welcome to package");
6. }
7. }

### 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.

1. //save as C.java otherwise Compilte Time Error
3. **class** A{}
4. **class** B{}
5. **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: |

1. //save as A.java
3. **package** javatpoint;
4. **public** **class** A{}
5. //save as B.java
7. **package** javatpoint;
8. **public** **class** B{}

# Exception Handling in Java

1. [Exception Handling](https://www.javatpoint.com/exception-handling-in-java)
2. [Advantage of Exception Handling](https://www.javatpoint.com/exception-handling-in-java#exceptionad)
3. [Hierarchy of Exception classes](https://www.javatpoint.com/exception-handling-in-java#exceptionhierarchy)
4. [Types of Exception](https://www.javatpoint.com/exception-handling-in-java#exceptiontypes)
5. [Scenarios where exception may occur](https://www.javatpoint.com/exception-handling-in-java#exceptionscenarios)

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 ClassNotFound, 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:

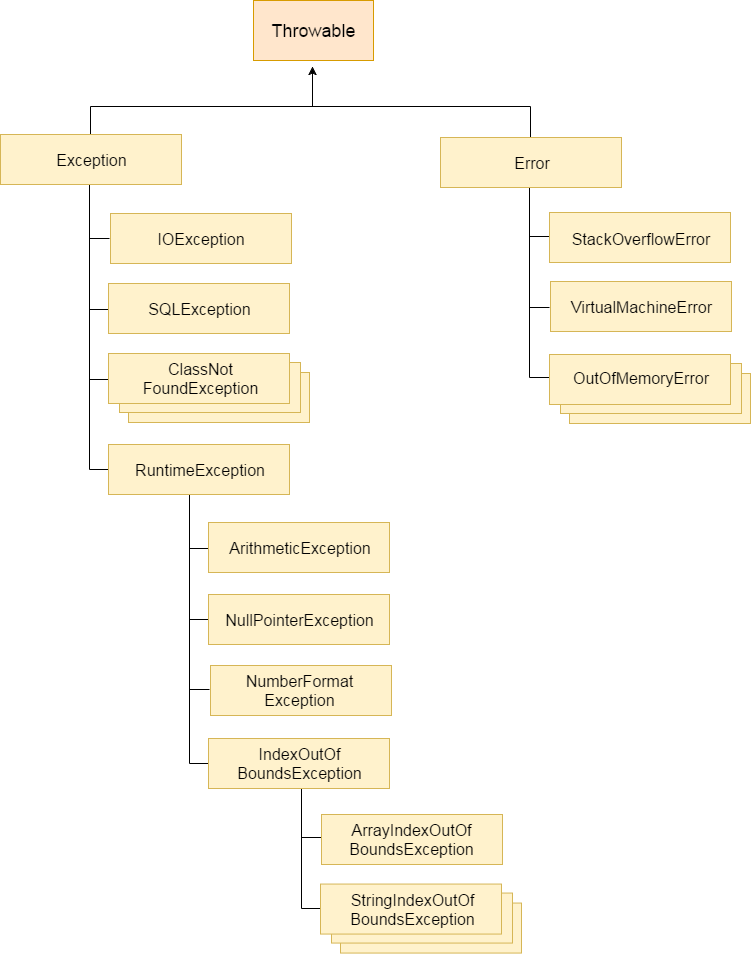
1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5;//exception occurs
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;

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 ? |

## Hierarchy of Java Exception classes



### 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.

1. **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.

1. String s=**null**;
2. 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.

1. String s="abc";
2. **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:

1. **int** a[]=**new** **int**[5];
2. a[10]=50; //ArrayIndexOutOfBoundsException

## 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

1. **try**{
2. //code that may throw exception
3. }**catch**(Exception\_class\_Name ref){}

#### Syntax of try-finally block

1. **try**{
2. //code that may throw exception
3. }**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.

1. **public** **class** Testtrycatch1{
2. **public** **static** **void** main(String args[]){
3. **int** data=50/0;//may throw exception
4. System.out.println("rest of the code...");
5. }
6. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testtrycatch1)

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.

1. **public** **class** Testtrycatch2{
2. **public** **static** **void** main(String args[]){
3. **try**{
4. **int** data=50/0;
5. }**catch**(ArithmeticException e)

{System.out.println(e);}

1. System.out.println("rest of the code...");
2. }
3. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testtrycatch2)

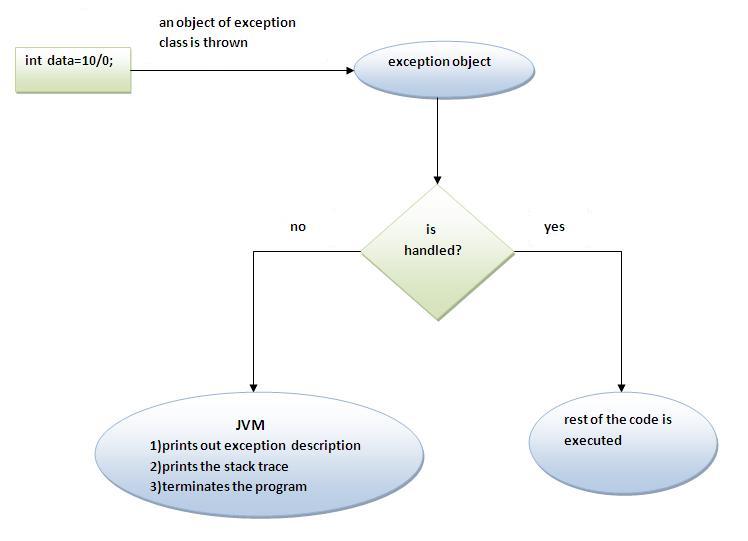
Output:

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 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 s1){

**super**(s1);

 }

}

**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[]){

Scanner s=new Scanner(System.in);

System.out.println(“Enter u r age”);

int age1=s.nextInt();

**try**{

      validate(age1);

      }**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...

# Java Applet

Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side.

### Advantage of Applet

There are many advantages of applet. They are as follows:

* It works at client side so less response time.
* Secured
* It can be executed by browsers running under many plateforms, including Linux, Windows, Mac Os etc.

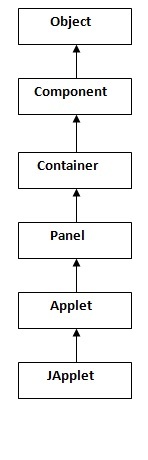
### Drawback of Applet

* Plugin is required at client browser to execute applet.

Do You Know

* Who is responsible to manage the life cycle of an applet ?
* How to perform animation in applet ?
* How to paint like paint brush in applet ?
* How to display digital clock in applet ?
* How to display analog clock in applet ?
* How to communicate two applets ?

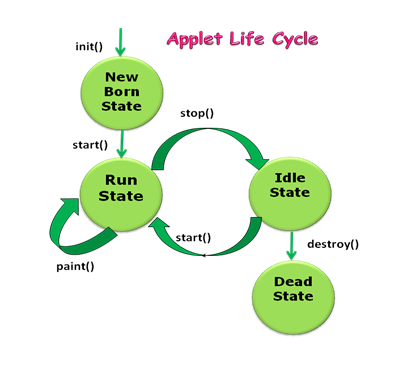
### Hierarchy of Applet

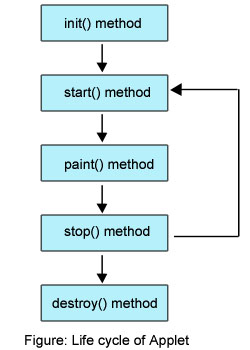


|  |
| --- |
| As displayed in the above diagram, Applet class extends Panel. Panel class extends Container which is the subclass of Component. |

### Lifecycle of Java Applet

1. Applet is initialized.
2. Applet is started.
3. Applet is painted.
4. Applet is stopped.
5. Applet is destroyed.





* **init()** method is called at the time of starting the execution. This is called only once in the life cycle.
* **start()** method is called by the init() method. This method is called a number of times in the life cycle; whenever the applet is **deiconifed**, to make the applet active.
* **paint()** method is called by the start() method. This is called number of times in the execution.
* **stop()** method is called whenever the applet window is **iconified** to inactivate the applet. This method is called number of times in the execution.
* **destroy()** method is called when the applet is closed. This method is called only once in the life cycle.

Observe, the init() and destroy() methods are called only once in the life cycle. But, start(), paint() and stop() methods are called a number of times.

### Lifecycle methods for Applet:

The java.applet.Applet class 4 life cycle methods and java.awt.Component class provides 1 life cycle methods for an applet.

### java.applet.Applet class

For creating any applet java.applet.Applet class must be inherited. It provides 4 life cycle methods of applet.

1. **public void init():** is used to initialized the Applet. It is invoked only once.
2. **public void start():** is invoked after the init() method or browser is maximized. It is used to start the Applet.
3. **public void stop():** is used to stop the Applet. It is invoked when Applet is stop or browser is minimized.
4. **public void destroy():** is used to destroy the Applet. It is invoked only once.

### java.awt.Component class

The Component class provides 1 life cycle method of applet.

1. **public void paint(Graphics g):** is used to paint the Applet. It provides Graphics class object that can be used for drawing oval, rectangle, arc etc.

### Who is responsible to manage the life cycle of an applet?

Java Plug-in software.

### How to run an Applet?

There are two ways to run an applet

1. By html file.
2. By appletViewer tool (for testing purpose).

### Simple example of Applet by html file:

To execute the applet by html file, create an applet and compile it. After that create an html file and place the applet code in html file. Now click the html file.

1. //First.java
2. **import** java.applet.Applet;
3. **import** java.awt.Graphics;
4. **public** **class** First **extends** Applet{
6. **public** **void** paint(Graphics g){
7. g.drawString("welcome",150,150);
8. }
10. }

#### Note: class must be public because its object is created by Java Plugin software that resides on the browser.

### myapplet.html

1. <html>
2. <body>
3. <applet code="First.class" width="300" height="300">
4. </applet>
5. </body>
6. </html>

### Simple example of Applet by appletviewer tool:

To execute the applet by appletviewer tool, create an applet that contains applet tag in comment and compile it. After that run it by: appletviewer First.java. Now Html file is not required but it is for testing purpose only.

1. //First.java
2. **import** java.applet.Applet;
3. **import** java.awt.Graphics;
4. **public** **class** First **extends** Applet{
6. **public** **void** paint(Graphics g){
7. g.drawString("welcome to applet",150,150);
8. }
10. }
11. /\*
12. <applet code="First.class" width="300" height="300">
13. </applet>
14. \*/

To execute the applet by appletviewer tool, write in command prompt:

**c:\>**javac First.java

**c:\>**appletviewer First.java

Displaying Graphics in Applet

java.awt.Graphics class provides many methods for graphics programming.

Commonly used methods of Graphics class:

1. **public abstract void drawString(String str, int x, int y):** is used to draw the specified string.
2. **public void drawRect(int x, int y, int width, int height):** draws a rectangle with the specified width and height.
3. **public abstract void fillRect(int x, int y, int width, int height):** is used to fill rectangle with the default color and specified width and height.
4. **public abstract void drawOval(int x, int y, int width, int height):** is used to draw oval with the specified width and height.
5. **public abstract void fillOval(int x, int y, int width, int height):** is used to fill oval with the default color and specified width and height.
6. **public abstract void drawLine(int x1, int y1, int x2, int y2):** is used to draw line between the points(x1, y1) and (x2, y2).
7. **public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer):** is used draw the specified image.
8. **public abstract void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle):** is used draw a circular or elliptical arc.
9. **public abstract void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle):** is used to fill a circular or elliptical arc.
10. **public abstract void setColor(Color c):** is used to set the graphics current color to the specified color.
11. **public abstract void setFont(Font font):** is used to set the graphics current font to the specified font.

Example of Graphics in applet:

1. **import** java.applet.Applet;
2. **import** java.awt.\*;
4. **public** **class** GraphicsDemo **extends** Applet{
6. **public** **void** paint(Graphics g){
7. g.setColor(Color.red);
8. g.drawString("Welcome",50, 50);
9. g.drawLine(20,30,20,300);
10. g.drawRect(70,100,30,30);
11. g.fillRect(170,100,30,30);
12. g.drawOval(70,200,30,30);
14. g.setColor(Color.pink);
15. g.fillOval(170,200,30,30);
16. g.drawArc(90,150,30,30,30,270);
17. g.fillArc(270,150,30,30,0,180);
19. }
20. }

myapplet.html

1. <html>
2. <body>
3. <applet code="GraphicsDemo.class" width="300" height="300">
4. </applet>
5. </body>
6. </html>

Displaying Image in Applet

Applet is mostly used in games and animation. For this purpose image is required to be displayed. The java.awt.Graphics class provide a method drawImage() to display the image.

Syntax of drawImage() method:

|  |
| --- |
| 1. **public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer):** is used draw the specified image. |

How to get the object of Image:

|  |
| --- |
| The java.applet.Applet class provides getImage() method that returns the object of Image. Syntax: |

1. **public** Image getImage(URL u, String image){}

Other required methods of Applet class to display image:

|  |
| --- |
| 1. **public URL getDocumentBase():** is used to return the URL of the document in which applet is embedded. 2. **public URL getCodeBase():** is used to return the base URL. |

Example of displaying image in applet:

1. **import** java.awt.\*;
2. **import** java.applet.\*;

5. **public** **class** DisplayImage **extends** Applet {
7. Image picture;
9. **public** **void** init() {
10. picture = getImage(getDocumentBase(),"sonoo.jpg");
11. }
13. **public** **void** paint(Graphics g) {
14. g.drawImage(picture, 30,30, **this**);
15. }
17. }

|  |
| --- |
| In the above example, drawImage() method of Graphics class is used to display the image. The 4th argument of drawImage() method of is ImageObserver object. The Component class implements ImageObserver interface. So current class object would also be treated as ImageObserver because Applet class indirectly extends the Component class. |

myapplet.html

1. <html>
2. <body>
3. <applet code="DisplayImage.class" width="300" height="300">
4. </applet>
5. </body>
6. </html>

Animation in Applet

|  |
| --- |
| Applet is mostly used in games and animation. For this purpose image is required to be moved. |

Example of animation in applet:

1. **import** java.awt.\*;
2. **import** java.applet.\*;
3. **public** **class** AnimationExample **extends** Applet {
5. Image picture;
7. **public** **void** init() {
8. picture =getImage(getDocumentBase(),"bike\_1.gif");
9. }
11. **public** **void** paint(Graphics g) {
12. **for**(**int** i=0;i<500;i++){
13. g.drawImage(picture, i,30, **this**);
15. **try**{Thread.sleep(100);}**catch**(Exception e){}
16. }
17. }
18. }

|  |
| --- |
| In the above example, drawImage() method of Graphics class is used to display the image. The 4th argument of drawImage() method of is ImageObserver object. The Component class implements ImageObserver interface. So current class object would also be treated as ImageObserver because Applet class indirectly extends the Component class. |

myapplet.html

1. <html>
2. <body>
3. <applet code="DisplayImage.class" width="300" height="300">
4. </applet>
5. </body>
6. </html>

EventHandling in Applet

|  |
| --- |
| As we perform event handling in AWT or Swing, we can perform it in applet also. Let's see the simple example of event handling in applet that prints a message by click on the button. |

Example of EventHandling in applet:

1. **import** java.applet.\*;
2. **import** java.awt.\*;
3. **import** java.awt.event.\*;
4. **public** **class** EventApplet **extends** Applet **implements** ActionListener{
5. Button b;
6. TextField tf;
8. **public** **void** init(){
9. tf=**new** TextField();
10. tf.setBounds(30,40,150,20);
12. b=**new** Button("Click");
13. b.setBounds(80,150,60,50);
15. add(b);add(tf);
16. b.addActionListener(**this**);
18. setLayout(**null**);
19. }
21. **public** **void** actionPerformed(ActionEvent e){
22. tf.setText("Welcome");
23. }
24. }

|  |
| --- |
| In the above example, we have created all the controls in init() method because it is invoked only once. |

myapplet.html

1. <html>
2. <body>
3. <applet code="EventApplet.class" width="300" height="300">
4. </applet>
5. </body>
6. </html>

JApplet class in Applet

|  |
| --- |
| As we prefer Swing to AWT. Now we can use JApplet that can have all the controls of swing. The JApplet class extends the Applet class. |

Example of EventHandling in JApplet:

1. **import** java.applet.\*;
2. **import** javax.swing.\*;
3. **import** java.awt.event.\*;
4. **public** **class** EventJApplet **extends** JApplet **implements** ActionListener{
5. JButton b;
6. JTextField tf;
7. **public** **void** init(){
9. tf=**new** JTextField();
10. tf.setBounds(30,40,150,20);
12. b=**new** JButton("Click");
13. b.setBounds(80,150,70,40);
15. add(b);add(tf);
16. b.addActionListener(**this**);
18. setLayout(**null**);
19. }
21. **public** **void** actionPerformed(ActionEvent e){
22. tf.setText("Welcome");
23. }
24. }

|  |
| --- |
| In the above example, we have created all the controls in init() method because it is invoked only once. |

myapplet.html

1. <html>
2. <body>
3. <applet code="EventJApplet.class" width="300" height="300">
4. </applet>
5. </body>
6. </html>

Painting in Applet

|  |
| --- |
| We can perform painting operation in applet by the mouseDragged() method of MouseMotionListener. |

Example of Painting in Applet:

1. **import** java.awt.\*;
2. **import** java.awt.event.\*;
3. **import** java.applet.\*;
4. **public** **class** MouseDrag **extends** Applet **implements** MouseMotionListener{
6. **public** **void** init(){
7. addMouseMotionListener(**this**);
8. setBackground(Color.red);
9. }
11. **public** **void** mouseDragged(MouseEvent me){
12. Graphics g=getGraphics();
13. g.setColor(Color.white);
14. g.fillOval(me.getX(),me.getY(),5,5);
15. }
16. **public** **void** mouseMoved(MouseEvent me){}
18. }

|  |
| --- |
| In the above example, getX() and getY() method of MouseEvent is used to get the current x-axis and y-axis. The getGraphics() method of Component class returns the object of Graphics. |

myapplet.html

1. <html>
2. <body>
3. <applet code="MouseDrag.class" width="300" height="300">
4. </applet>
5. </body>
6. </html>

Digital clock in Applet

Digital clock can be created by using the Calendar and SimpleDateFormat class. Let's see the simple example:

Example of Digital clock in Applet:

1. **import** java.applet.\*;
2. **import** java.awt.\*;
3. **import** java.util.\*;
4. **import** java.text.\*;
6. **public** **class** DigitalClock **extends** Applet **implements** Runnable {
8. Thread t = **null**;
9. **int** hours=0, minutes=0, seconds=0;
10. String timeString = "";
12. **public** **void** init() {
13. setBackground( Color.green);
14. }
16. **public** **void** start() {
17. t = **new** Thread( **this** );
18. t.start();
19. }

22. **public** **void** run() {
23. **try** {
24. **while** (**true**) {
26. Calendar cal = Calendar.getInstance();
27. hours = cal.get( Calendar.HOUR\_OF\_DAY );
28. **if** ( hours > 12 ) hours -= 12;
29. minutes = cal.get( Calendar.MINUTE );
30. seconds = cal.get( Calendar.SECOND );
32. SimpleDateFormat formatter = **new** SimpleDateFormat("hh:mm:ss");
33. Date date = cal.getTime();
34. timeString = formatter.format( date );
36. repaint();
37. t.sleep( 1000 );  // interval given in milliseconds
38. }
39. }
40. **catch** (Exception e) { }
41. }

44. **public** **void** paint( Graphics g ) {
45. g.setColor( Color.blue );
46. g.drawString( timeString, 50, 50 );
47. }
48. }

|  |
| --- |
| In the above example, getX() and getY() method of MouseEvent is used to get the current x-axis and y-axis. The getGraphics() method of Component class returns the object of Graphics. |

myapplet.html

1. <html>
2. <body>
3. <applet code="DigitalClock.class" width="300" height="300">
4. </applet>
5. </body>
6. </html>

Analog clock in Applet

Analog clock can be created by using the Math class. Let's see the simple example:

Example of Analog clock in Applet:

1. **import** java.applet.\*;
2. **import** java.awt.\*;
3. **import** java.util.\*;
4. **import** java.text.\*;
6. **public** **class** MyClock **extends** Applet **implements** Runnable {
8. **int** width, height;
9. Thread t = **null**;
10. **boolean** threadSuspended;
11. **int** hours=0, minutes=0, seconds=0;
12. String timeString = "";
14. **public** **void** init() {
15. width = getSize().width;
16. height = getSize().height;
17. setBackground( Color.black );
18. }
20. **public** **void** start() {
21. **if** ( t == **null** ) {
22. t = **new** Thread( **this** );
23. t.setPriority( Thread.MIN\_PRIORITY );
24. threadSuspended = **false**;
25. t.start();
26. }
27. **else** {
28. **if** ( threadSuspended ) {
29. threadSuspended = **false**;
30. **synchronized**( **this** ) {
31. notify();
32. }
33. }
34. }
35. }
37. **public** **void** stop() {
38. threadSuspended = **true**;
39. }
41. **public** **void** run() {
42. **try** {
43. **while** (**true**) {
45. Calendar cal = Calendar.getInstance();
46. hours = cal.get( Calendar.HOUR\_OF\_DAY );
47. **if** ( hours > 12 ) hours -= 12;
48. minutes = cal.get( Calendar.MINUTE );
49. seconds = cal.get( Calendar.SECOND );
51. SimpleDateFormat formatter
52. = **new** SimpleDateFormat( "hh:mm:ss", Locale.getDefault() );
53. Date date = cal.getTime();
54. timeString = formatter.format( date );
56. // Now the thread checks to see if it should suspend itself
57. **if** ( threadSuspended ) {
58. **synchronized**( **this** ) {
59. **while** ( threadSuspended ) {
60. wait();
61. }
62. }
63. }
64. repaint();
65. t.sleep( 1000 );  // interval specified in milliseconds
66. }
67. }
68. **catch** (Exception e) { }
69. }
71. **void** drawHand( **double** angle, **int** radius, Graphics g ) {
72. angle -= 0.5 \* Math.PI;
73. **int** x = (**int**)( radius\*Math.cos(angle) );
74. **int** y = (**int**)( radius\*Math.sin(angle) );
75. g.drawLine( width/2, height/2, width/2 + x, height/2 + y );
76. }
78. **void** drawWedge( **double** angle, **int** radius, Graphics g ) {
79. angle -= 0.5 \* Math.PI;
80. **int** x = (**int**)( radius\*Math.cos(angle) );
81. **int** y = (**int**)( radius\*Math.sin(angle) );
82. angle += 2\*Math.PI/3;
83. **int** x2 = (**int**)( 5\*Math.cos(angle) );
84. **int** y2 = (**int**)( 5\*Math.sin(angle) );
85. angle += 2\*Math.PI/3;
86. **int** x3 = (**int**)( 5\*Math.cos(angle) );
87. **int** y3 = (**int**)( 5\*Math.sin(angle) );
88. g.drawLine( width/2+x2, height/2+y2, width/2 + x, height/2 + y );
89. g.drawLine( width/2+x3, height/2+y3, width/2 + x, height/2 + y );
90. g.drawLine( width/2+x2, height/2+y2, width/2 + x3, height/2 + y3 );
91. }
93. **public** **void** paint( Graphics g ) {
94. g.setColor( Color.gray );
95. drawWedge( 2\*Math.PI \* hours / 12, width/5, g );
96. drawWedge( 2\*Math.PI \* minutes / 60, width/3, g );
97. drawHand( 2\*Math.PI \* seconds / 60, width/2, g );
98. g.setColor( Color.white );
99. g.drawString( timeString, 10, height-10 );
100. }
101. }

myapplet.html

1. <html>
2. <body>
3. <applet code="MyClock.class" width="300" height="300">
4. </applet>
5. </body>
6. </html>

Parameter in Applet

We can get any information from the HTML file as a parameter. For this purpose, Applet class provides a method named getParameter(). Syntax:

1. **public** String getParameter(String parameterName)

Example of using parameter in Applet:

1. **import** java.applet.Applet;
2. **import** java.awt.Graphics;
4. **public** **class** UseParam **extends** Applet{
6. **public** **void** paint(Graphics g){
7. String str=getParameter("msg");
8. g.drawString(str,50, 50);
9. }
11. }

myapplet.html

1. <html>
2. <body>
3. <applet code="UseParam.class" width="300" height="300">
4. <param name="msg" value="Welcome to applet">
5. </applet>
6. </body>
7. </html>

Applet Communication

java.applet.AppletContext class provides the facility of communication between applets. We provide the name of applet through the HTML file. It provides getApplet() method that returns the object of Applet. Syntax:

1. **public** Applet getApplet(String name){}

Example of Applet Communication

1. **import** java.applet.\*;
2. **import** java.awt.\*;
3. **import** java.awt.event.\*;
4. **public** **class** ContextApplet **extends** Applet **implements** ActionListener{
5. Button b;
7. **public** **void** init(){
8. b=**new** Button("Click");
9. b.setBounds(50,50,60,50);
11. add(b);
12. b.addActionListener(**this**);
13. }
15. **public** **void** actionPerformed(ActionEvent e){
17. AppletContext ctx=getAppletContext();
18. Applet a=ctx.getApplet("app2");
19. a.setBackground(Color.yellow);
20. }
21. }

myapplet.html

1. <html>
2. <body>
3. <applet code="ContextApplet.class" width="150" height="150" name="app1">
4. </applet>
6. <applet code="First.class" width="150" height="150" name="app2">
7. </applet>
8. </body>
9. </html>