Static

* A class can be made static only if it is an inner class. The nested static class can be accessed even without the object of the outer class.
* Static block is mostly used for changing the default values of static variables. This block gets executed when the class is loaded in the memory. A class can have multiple Static blocks, which will execute in the same sequence in which they have been written into the program.
  + Is used to initialize the static data member.
  + It is executed before main method (before the main method, this point is not very clear,have to read up more on this) at the time of class loading.
* A class can have any number of static initialization blocks, and they can appear anywhere in the class body. The runtime system guarantees that static initialization blocks are called in the order that they appear in the source code. And dont forget, this code will be executed when JVM loads the class. JVM combines all these blocks into one single static block and then executes. Here are a couple of points I like to mention: If you have executable statements in the static block, JVM will automatically execute these statements when the class is loaded into JVM.
* If you’re referring some static variables/methods from the static blocks, these statements will be executed after the class is loaded into JVM same as above i.e., now the static variables/methods referred and the static block both will be executed.
* *URL :* [*http://www.jusfortechies.com/java/core-java/static-blocks.php*](http://www.jusfortechies.com/java/core-java/static-blocks.php)
* Really nice url.

public class StaticExample{

static {

System.out.println("This is first static block");

}

public StaticExample(){

System.out.println("This is constructor");

}

public static String staticString = "Static Variable";

static {

System.out.println("This is second static block and "

+ staticString);

}

public static void main(String[] args){

StaticExample statEx = new StaticExample();

StaticExample.staticMethod2();

}

static {

staticMethod();

System.out.println("This is third static block");

}

public static void staticMethod() {

System.out.println("This is static method");

}

public static void staticMethod2() {

System.out.println("This is static method2");

}

}

What will happen when you execute the above code? You will see below output.

This is first static block

This is second static block and Static Variable

This is static method

This is third static block

This is constructor

This is static method2

Now lets the output. First all static blocks are positioned in the code and they are executed when the class is loaded into JVM. Since the static method staticMethod() is called inside third static block, its executed before calling the main method. But the staticMethod2() static method is executed after the class is instantiated because it is being called after the instantiation of the class.

Again if you miss to precede the block with "static" keyword, the block is called "constructor block" and will be executed when the class is instantiated. The constructor block will be copied into each constructor of the class. Say for example you have four parameterized constructors, then four copies of contructor blocks will be placed inside the constructor, one for each. Lets execute the below example and see the output.

When does a class get loaded in JVM :

A class is loaded only when you require information about that class.

public class SomethingCaller {

public static Something something = null; // (1) does not cause class loading

public static Class<?> somethingClass = Something.class; // (2) causes class loading

public void doSomething() {

new Something(); // (3) causes class loading

}

}

The lines (2) & (3) would cause the class to be loaded. The Something.class object contains information (line (2)) which could only come from the class definition, so you need to load the class. The call to the constructor (3) obviously requires the class definition. Similarly for any other method on the class.

However, line (1) doesn't cause the class to be loaded, because you don't actually need any information, it's just a reference to an object.

EDIT: In your changed question, you ask whether referring to Something.class loads the class. Yes it does. It does not load the class until main() is executed though. Using the following code:

public class SomethingTest {

public static void main(String[] args) {

new SomethingCaller();

}

}

public class SomethingCaller {

public void doSomething() {

Class<?> somethingClass = Something.class;

}

}

public class Something {}

This code does not cause the Something.class to be loaded. However, if I call doSomething(), the class is loaded. To test this, create the above classes, compile them and delete the Something.class file. The above code does not crash with a ClassNotFoundException.

* Static Methods can access class variables without using object of the class. It can access non-static methods and non-static variables by using objects. Static methods can be accessed directly in static and non-static methods. A static member method cannot call a non-static member method directly. You have to create an object of it and only then call.
  + Instance methods can access instance variables and instance methods directly.
  + Instance methods can access class variables and class methods directly.
  + Class methods can access class variables and class methods directly.
  + Class methods cannot access instance variables or instance methods directly—they must use an object reference. Also, class methods cannot use the “this” keyword as there is no instance for this to refer to.
  + The static method cannot use non static data member or call non-static method directly.
  + This and super cannot be used in static context.
* Static variables are initialized, local variables are not. They can be accessed in any other class w/o obj. They can be accessed in static and non-static functions. (I think they can be modified only in static member functions.)

This

-THIS() is used for constructor chaining.

class Student13{

    int id;

    String name;

    Student13(){System.out.println("default constructor is invoked");}

    Student13(int id,String name){

    this ();//it is used to invoked current class constructor.

    this.id = id;

    this.name = name;

    }

Super

If you are creating your own constructor and you don't have either this() or super() as the first statement, compiler will provide super() as the first statement of the constructor.

Covariant Return Type :

The covariant return type specifies that the return type may vary in the same direction as the subclass.

Before Java5, it was not possible to override any method by changing the return type. But now, since Java5, it is possible to override method by changing the return type if subclass overrides any method whose return type is Non-Primitive but it changes its return type to subclass type. Now, since java5, it is possible to override any method by changing the return type if the return type of the subclass overriding method is subclass type. It is known as covariant return type.(I think over riding by changing return type is possible only if the return type is the class type …confirm this)

Let's take a simple example:

Simple example of Covariant Return Type

1. class A{
2. A get(){return this;}
3. }
5. class B1 extends A{
6. B1 get(){return this;}
7. void message(){System.out.println("welcome to covariant return type");}
9. public static void main(String args[]){
10. new B1().get().message();
11. }
12. }

Output: welcome to covariant return typeAs you can see in the above example, the return type of the get() method of A class is A but the return type of the get() method of B class is B. Both methods have different return type but it is method overriding. This is known as covariant return type.

Final:

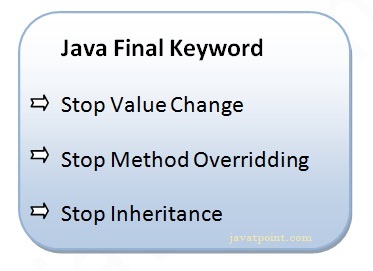
Final Keyword In Java

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

1. class Bike9{
2. final int speedlimit=90;//final variable
3. void run(){
4. speedlimit=400;
5. }
6. public static void main(String args[]){
7. Bike9 obj=new  Bike9();
8. obj.run();
9. }
10. }//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

1. class Bike{
2. final void run(){System.out.println("running");}
3. }
5. class Honda extends Bike{
6. void run(){System.out.println("running safely with 100kmph");}
8. public static void main(String args[]){
9. Honda honda= new Honda();
10. honda.run();
11. }
12. }

[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

1. final class Bike{}
3. class Honda1 extends Bike{
4. void run(){System.out.println("running safely with 100kmph");}
6. public static void main(String args[]){
7. Honda1 honda= new Honda();
8. honda.run();
9. }
10. }

[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:

1. class Bike{
2. final void run(){System.out.println("running...");}
3. }
4. class Honda2 extends Bike{
5. public static void main(String args[]){
6. new Honda2().run();
7. }
8. }

[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

1. class Student{
2. int id;
3. String name;
4. final String PAN\_CARD\_NUMBER;
5. ...
6. }

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.

1. class Bike11{
2. int cube(final int n){
3. n=n+2;//can't be changed as n is final
4. n\*n\*n;
5. }
6. public static void main(String args[]){
7. Bike11 b=new Bike11();
8. b.cube(5);
9. }
10. }

[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.

# Polymorphism in Java

**Polymorphism in java** is a concept by which we can perform a single action by different ways. Polymorphism is derived from 2 greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in java: compile time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

If you overload static method in java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.

## Runtime Polymorphism in Java

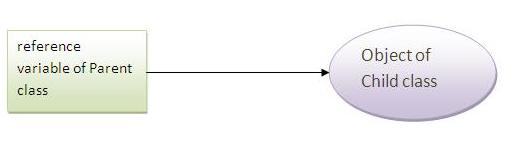
Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

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

Let's first understand the upcasting before Runtime Polymorphism.

### Upcasting

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



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

### Example of Java Runtime Polymorphism

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

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

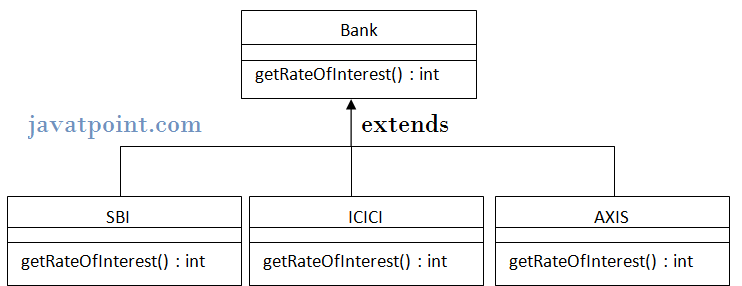
1. class Bike{
2. void run(){System.out.println("running");}
3. }
4. class Splender extends Bike{
5. void run(){System.out.println("running safely with 60km");}
7. public static void main(String args[]){
8. Bike b = new Splender();//upcasting
9. b.run();
10. }
11. }

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

Output:running safely with 60km.

### Real example of Java Runtime Polymorphism

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



Note: It is also given in method overriding but there was no upcasting.

1. class Bank{
2. int getRateOfInterest(){return 0;}
3. }
5. class SBI extends Bank{
6. int getRateOfInterest(){return 8;}
7. }
9. class ICICI extends Bank{
10. int getRateOfInterest(){return 7;}
11. }
12. class AXIS extends Bank{
13. int getRateOfInterest(){return 9;}
14. }
16. class Test3{
17. public static void main(String args[]){
18. Bank b1=new SBI();
19. Bank b2=new ICICI();
20. Bank b3=new AXIS();
21. System.out.println("SBI Rate of Interest: "+b1.getRateOfInterest());
22. System.out.println("ICICI Rate of Interest: "+b2.getRateOfInterest());
23. System.out.println("AXIS Rate of Interest: "+b3.getRateOfInterest());
24. }
25. }

[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=Test3)

Output:

SBI Rate of Interest: 8

ICICI Rate of Interest: 7

AXIS Rate of Interest: 9

### Java Runtime Polymorphism with data member

|  |
| --- |
| Method is overridden not the datamembers, so runtime polymorphism can't be achieved by data members. |
| In the example given below, both the classes have a datamember speedlimit, we are accessing the datamember by the reference variable of Parent class which refers to the subclass object. Since we are accessing the datamember which is not overridden, hence it will access the datamember of Parent class always. |

#### *Rule: Runtime polymorphism can't be achieved by data members.*

1. class Bike{
2. int speedlimit=90;
3. }
4. class Honda3 extends Bike{
5. int speedlimit=150;
7. public static void main(String args[]){
8. Bike obj=new Honda3();
9. System.out.println(obj.speedlimit);//90
10. }

[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=Honda3)

Output:90

## Java Runtime Polymorphism with Multilevel Inheritance

Let's see the simple example of Runtime Polymorphism with multilevel inheritance.

1. class Animal{
2. void eat(){System.out.println("eating");}
3. }
5. class Dog extends Animal{
6. void eat(){System.out.println("eating fruits");}
7. }
9. class BabyDog extends Dog{
10. void eat(){System.out.println("drinking milk");}
12. public static void main(String args[]){
13. Animal a1,a2,a3;
14. a1=new Animal();
15. a2=new Dog();
16. a3=new BabyDog();
18. a1.eat();
19. a2.eat();
20. a3.eat();
21. }
22. }

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

Output: eating

eating fruits

drinking Milk

### Try for Output

1. class Animal{
2. void eat(){System.out.println("animal is eating...");}
3. }
5. class Dog extends Animal{
6. void eat(){System.out.println("dog is eating...");}
7. }
9. class BabyDog1 extends Dog{
10. public static void main(String args[]){
11. Animal a=new BabyDog1();
12. a.eat();
13. }}

[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=BabyDog1)

Output: Dog is eating

Since, BabyDog is not overriding the eat() method, so eat() method of Dog class is invoked.

Static Binding and Dynamic Binding

Connecting a method call to the method body is known as binding.

There are two types of binding

1. static binding (also known as early binding).
2. dynamic binding (also known as late binding).

Understanding Type

Let's understand the type of instance.

1) variables have a type

Each variable has a type, it may be primitive and non-primitive.

1. int data=30;

Here data variable is a type of int.

2) References have a type

1. class Dog{
2. public static void main(String args[]){
3. Dog d1;//Here d1 is a type of Dog
4. }
5. }

3) Objects have a type

|  |
| --- |
| An object is an instance of particular java class,but it is also an instance of its superclass. |

1. class Animal{}
3. class Dog extends Animal{
4. public static void main(String args[]){
5. Dog d1=new Dog();
6. }
7. }

|  |
| --- |
| Here d1 is an instance of Dog class, but it is also an instance of Animal. |

static binding

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

If there is any private, final or static method in a class, there is static binding.??????(Did not understand this point.)

Example of static binding

1. class Dog{
2. private void eat(){System.out.println("dog is eating...");}
4. public static void main(String args[]){
5. Dog d1=new Dog();
6. d1.eat();
7. }
8. }

Dynamic binding

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

Example of dynamic binding

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

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

Output:dog is eating...

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

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 Abstaction

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. }
5. class Honda4 extends Bike{
6. void run(){System.out.println("running safely..");}
8. public static void main(String args[]){
9. Bike obj = new Honda4();
10. obj.run();
11. }
12. }

[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. }
9. class Circle1 extends Shape{
10. void draw(){System.out.println("drawing circle");}
11. }
13. //In real scenario, method is called by programmer or user
14. class TestAbstraction1{
15. public static void main(String args[]){
16. Shape s=new Circle1();//In real scenario, object is provided through method e.g. getShape() method
17. s.draw();
18. }
19. }

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

drawing circle

Another example of abstract class in java

*File: TestBank.java*

1. abstract class Bank{
2. abstract int getRateOfInterest();
3. }
5. class SBI extends Bank{
6. int getRateOfInterest(){return 7;}
7. }
8. class PNB extends Bank{
9. int getRateOfInterest(){return 7;}
10. }
12. class TestBank{
13. public static void main(String args[]){
14. Bank b=new SBI();//if object is PNB, method of PNB will be invoked
15. int interest=b.getRateOfInterest();
16. System.out.println("Rate of Interest is: "+interest+" %");
17. }}

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

Rate of Interest is: 7 %

Abstract class having constructor, data member, methods etc.

An abstract class can have data member, abstract method, method body, constructor and even main() method. *Abstract class can have a main method too !*

*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

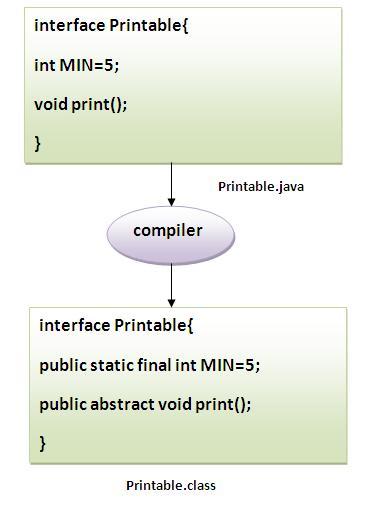
I am d

Interface :

-It has static constants and abstract methods only …. It cannot have static methods, and it cannot have non abstract methods. Interface represents full abstraction.

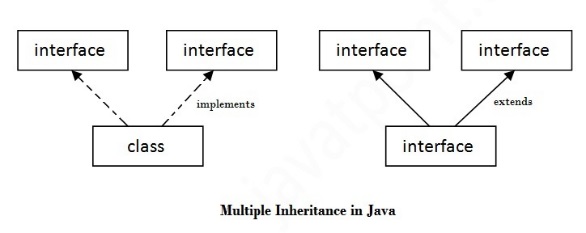
-The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.

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



Multiple Inheritance:

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



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

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

1. interface Printable{
2. void print();
3. }
5. interface Showable{
6. void print();
7. }
9. class testinterface1 implements Printable,Showable{
11. public void print(){System.out.println("Hello");}
13. public static void main(String args[]){
14. testinterface1 obj = new testinterface1();
15. obj.print();
16. }
17. }

[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=testinterface1)

Hello

Q) What is marker or tagged interface?

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

//How Serializable interface is written?

public interface Serializable{

}

Nested Interfaces are possible. ☺

Can we Overload or Override static methods in java ?

Let us first define Overloading and Overriding.

[Overriding](http://en.wikipedia.org/wiki/Method_overriding): Overriding is a feature of OOP languages like Java that is related to run-time polymorphism. A subclass (or derived class) provides a specific implementation of a method in superclass (or base class).  
The implementation to be executed is decided at run-time and decision is made according to the object used for call. Note that signatures of both methods must be same.

[Overloading](http://en.wikipedia.org/wiki/Function_overloading): Overloading is also a feature of OOP languages like Java that is related to compile time (or static) polymorphism. This feature allows different methods to have same name, but different signatures, especially number of input parameters and type of input paramaters. Note that in both C++ and Java, [methods cannot be overloaded according to return type.](http://www.geeksforgeeks.org/g-fact-75/)

Can we overload static methods?  
The answer is ‘Yes’. We can have two ore more static methods with same name, but differences in input parameters. For example, consider the following Java program.

|  |
| --- |
| // filename Test.java  public class Test {      public static void foo() {          System.out.println("Test.foo() called ");      }      public static void foo(int a) {          System.out.println("Test.foo(int) called ");      }      public static void main(String args[])      {          Test.foo();          Test.foo(10);      }  } |

Output:

Test.foo() called

Test.foo(int) called

Can we overload methods that differ only by static keyword?  
We cannot overload two methods in Java if they differ only by static keyword (number of parameters and types of parameters is same). See following Java program for example. This behaviour is same in C++ (See point 2 of [this](http://www.geeksforgeeks.org/function-overloading-in-c/)).

|  |
| --- |
| // filename Test.java  public class Test {      public static void foo() {          System.out.println("Test.foo() called ");      }      public void foo() { // Compiler Error: cannot redefine foo()          System.out.println("Test.foo(int) called ");      }      public static void main(String args[]) {          Test.foo();      }  } |

Output: Compiler Error, cannot redefine foo()

Can we Override static methods in java?  
We can declare static methods with same signature in subclass, but it is not considered overriding as there won’t be any run-time polymorphism. Hence the answer is ‘No’.  
If a derived class defines a static method with same signature as a static method in base class, the method in the derived class hides the method in the base class.

|  |
| --- |
| /\* Java program to show that if static method is redefined by     a derived class, then it is not overriding. \*/    // Superclass  class Base {        // Static method in base class which will be hidden in subclass      public static void display() {          System.out.println("Static or class method from Base");      }         // Non-static method which will be overridden in derived class       public void print()  {           System.out.println("Non-static or Instance method from Base");      }  }    // Subclass  class Derived extends Base {        // This method hides display() in Base      public static void display() {           System.out.println("Static or class method from Derived");      }        // This method overrides print() in Base      public void print() {           System.out.println("Non-static or Instance method from Derived");     }  }    // Driver class  public class Test {      public static void main(String args[ ])  {         Base obj1 = new Derived();           // As per overriding rules this should call to class Derive's static         // overridden method. Since static method can not be overridden, it         // calls Base's display()         obj1.display();           // Here overriding works and Derive's print() is called         obj1.print();      }  } |

Output:

Static or class method from Base

Non-static or Instance method from Derived

Following are some important points for method overriding and static methods in Java.  
1) For class (or static) methods, the method according to the type of reference is called, not according to the object being referred, which means method call is decided at compile time. So, runtime polymorphism is not possible.

2) For instance (or non-static) methods, the method is called according to the type of object being referred, not according to the type of reference, which means method calls is decided at run time. So runtime polymorphism is possible.

3) An instance method cannot override a static method, and a static method cannot hide an instance method. For example, the following program has two compiler errors.

|  |
| --- |
| /\* Java program to show that if static methods are redefined by     a derived class, then it is not overriding but hidding. \*/    // Superclass  class Base {        // Static method in base class which will be hidden in subclass      public static void display() {          System.out.println("Static or class method from Base");      }         // Non-static method which will be overridden in derived class       public void print()  {           System.out.println("Non-static or Instance method from Base");      }  }    // Subclass  class Derived extends Base {        // Static is removed here (Causes Compiler Error)      public void display() {          System.out.println("Non-static method from Derived");      }        // Static is added here (Causes Compiler Error)      public static void print() {          System.out.println("Static method from Derived");     }  } |

4) In a subclass (or Derived Class), we can overload the methods inherited from the superclass. Such overloaded methods neither hide nor override the superclass methods — they are new methods, unique to the subclass.

Abstract class Interface

1)An abstract class can have method body (non-abstract methods). Interface have only abstract methods.

2)An abstract class can have instance variables. An interface cannot have instance variables.

3)An abstract class can have constructor. Interface cannot have constructor.

4)An abstract class can have static methods. Interface cannot have static methods.

5)You can extends one abstract class. You can implement multiple interfaces.

\*\*\*Upcasting happens in case of interfaces also.

--java.lang.package is loaded by the jvm by default.

Static Import:

The static import feature of Java 5 facilitate the java programmer to access any static member of a class directly. There is no need to qualify it by the class name.

Advantage of static import:

* Less coding is required if you have access any static member of a class oftenly.

Disadvantage of static import:

* If you overuse the static import feature, it makes the program unreadable and unmaintainable.

Simple Example of static import

1. import static java.lang.System.\*;
2. class StaticImportExample{
3. public static void main(String args[]){
5. out.println("Hello");//Now no need of System.out
6. out.println("Java");
8. }
9. }

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

Output:Hello

Java

What is the difference between import and static import?

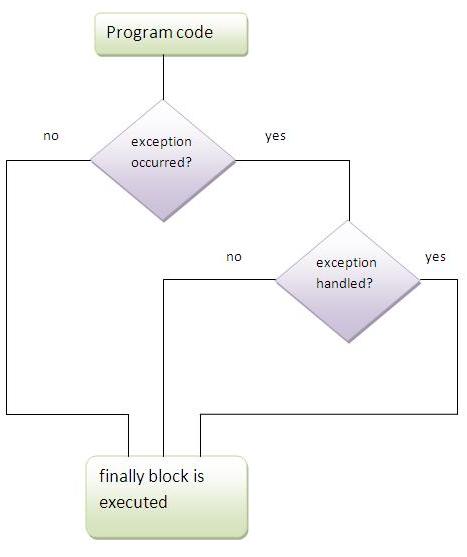
The import allows the java programmer to access classes of a package without package qualification whereas the static import feature allows to access the static members of a class without the class qualification. The import provides accessibility to classes and interface whereas static import provides accessibility to static members of the class.

Java finally block

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

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

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



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

Why use java finally

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

Usage of Java finally

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

Case 1

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

1. class TestFinallyBlock{
2. public static void main(String args[]){
3. try{
4. int data=25/5;
5. System.out.println(data);
6. }
7. catch(NullPointerException e){System.out.println(e);}
8. finally{System.out.println("finally block is always executed");}
9. System.out.println("rest of the code...");
10. }
11. }

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

Output:5

finally block is always executed

rest of the code...

Case 2

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

1. class TestFinallyBlock1{
2. public static void main(String args[]){
3. try{
4. int data=25/0;
5. System.out.println(data);
6. }
7. catch(NullPointerException e){System.out.println(e);}
8. finally{System.out.println("finally block is always executed");}
9. System.out.println("rest of the code...");
10. }
11. }

[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TestFinallyBlock1)

Output:finally block is always executed

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

Case 3

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

1. public class TestFinallyBlock2{
2. public static void main(String args[]){
3. try{
4. int data=25/0;
5. System.out.println(data);
6. }
7. catch(ArithmeticException e){System.out.println(e);}
8. finally{System.out.println("finally block is always executed");}
9. System.out.println("rest of the code...");
10. }
11. }

[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TestFinallyBlock2)

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

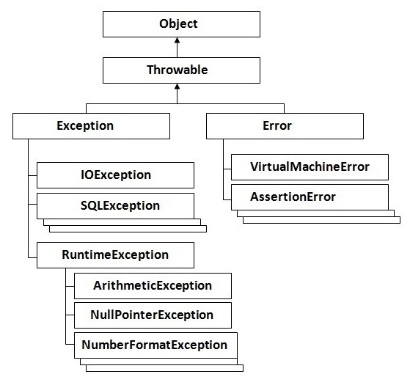
finally block is always executed

rest of the code...

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

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

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.

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

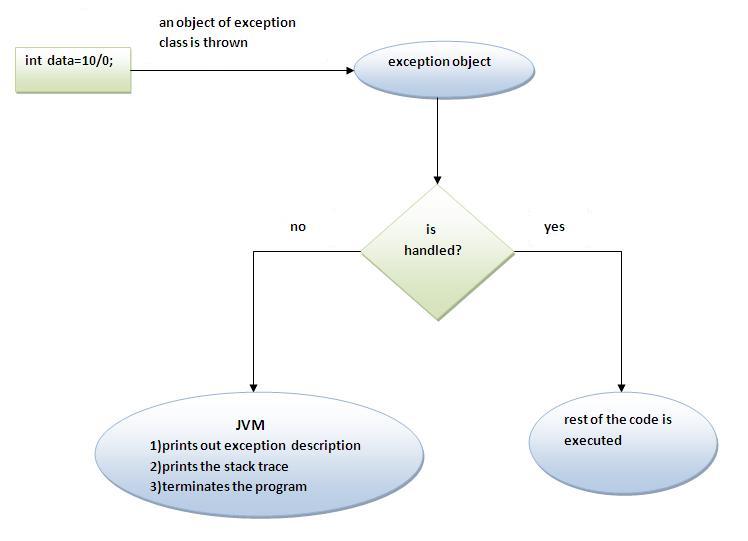
Syntax of java try-catch

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{}

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.

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

Why use nested try block

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

Java Exception propagation

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

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

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

***Checked Exception (compile time are not propagated) Unchecked Exception (runtime are propagated down the calling chain.)i.e Checked Exceptions need to be handled then and there I guess, Exception handling is more often used for checked exception ,if an unchecked exception like null pointer occurs more often than not it is the programmers wrong coding logic;***

Syntax of java throws

1. return\_type method\_name() **throws** exception\_class\_name{
2. ...
3. }

Which exception should be declared

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

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

Advantage of Java throws keyword

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

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

If you are calling a method that throws an exception, you should either handle that exception or throw it further.

Difference between throw and throws:

|  |  |
| --- | --- |
| **throw keyword** | **throws keyword** |
| 1)throw is used to explicitly throw an exception. | throws is used to declare an exception. |
| 2)checked exception can not be propagated without throws. | checked exception can be propagated with throws. |
| 3)throw is followed by an instance. | throws is followed by class. |
| 4)throw is used within the method. | throws is used with the method signature. |
| 5)You cannot throw multiple exception | You can declare multiple exception e.g. public void method()throws IOException,SQLException. |

Que) Can we rethrow an exception?

Yes by throwing same exception in catch block.

**Exception handling with method overriding : Java T point .**

**----What are different ways to create String Object?**

**ANS)We can create String object using new operator like any normal java class or we can use double quotes to create a String object. There are several constructors available in String class to get String from char array, byte array, StringBuffer and StringBuilder.**

|  |  |
| --- | --- |
| **1**  **2** | **String str = new String("abc");**  **String str1 = "abc";** |

**When we create a String using double quotes, JVM looks in the String pool to find if any other String is stored with same value. If found, it just returns the reference to that String object else it creates a new String object with given value and stores it in the String pool.  
When we use new operator, JVM creates the String object but don’t store it into the String Pool. We can useintern() method to store the String object into String pool or return the reference if there is already a String with equal value present in the pool.**