JAVA OOPS:

Java OOPs Concepts

1. [Object Oriented Programming](https://www.javatpoint.com/java-oops-concepts#oops)
2. [Advantage of OOPs over Procedure-oriented programming language](https://www.javatpoint.com/java-oops-concepts#oopsadvantage)
3. [Difference between Objcet-oriented and Objcet-based programming language.](https://www.javatpoint.com/java-oops-concepts#oopsdifference)

In this page, we will learn about basics of OOPs. Object Oriented Programming is a paradigm that provides many concepts such as **inheritance**, **data binding**, **polymorphism** etc.

**Simula** is considered as the first object-oriented programming language. The programming paradigm where everything is represented as an object, is known as truly object-oriented programming language.

**Smalltalk** is considered as the first truly object-oriented programming language.

OOPs (Object Oriented Programming System)

**Object** means a real word entity such as pen, chair, table etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

* Object
* Class
* Inheritance
* Polymorphism
* Abstraction
* Encapsulation

Object

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

Class

**Collection of objects** is called class. It is a logical entity.

Inheritance

**When one object acquires all the properties and behaviours of parent object** i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.



Polymorphism

When **one task is performed by different ways** i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.

In java, we use method overloading and method overriding to achieve polymorphism.

Another example can be to speak something e.g. cat speaks meaw, dog barks woof etc.

Abstraction

**Hiding internal details and showing functionality** is known as abstraction. For example: phone call, we don't know the internal processing.

In java, we use abstract class and interface to achieve abstraction.



Encapsulation

**Binding (or wrapping) code and data together into a single unit is known as encapsulation**. For example: capsule, it is wrapped with different medicines.

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Advantage of OOPs over Procedure-oriented programming language

|  |
| --- |
| 1)OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows. |
| 2)OOPs provides data hiding whereas in Procedure-oriented programming language a global data can be accessed from anywhere. |
| 3)OOPs provides ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language. |

|  |  |
| --- | --- |
| Global Data | Object Data |

What is difference between object-oriented programming language and object-based programming language?

Object based programming language follows all the features of OOPs except Inheritance. JavaScript and VBScript are examples of object based programming languages.

Do You Know ?

|  |
| --- |
| * Can we overload main method ? * Constructor returns a value but, what ? * Can we create a program without main method ? * What are the 6 ways to use this keyword ? * Why multiple inheritance is not supported in java ? * Why use aggregation ? * Can we override the static method ? * What is covariant return type ? * What are the three usage of super keyword? * Why use instance initializer block? * What is the usage of blank final variable ? * What is marker or tagged interface ? * What is runtime polymorphism or dynamic method dispatch ? * What is the difference between static and dynamic binding ? * How downcasting is possible in java ? * What is the purpose of private constructor? * What is object cloning ? |

What we will learn in OOPs Concepts ?

|  |
| --- |
| * Advantage of OOPs * Naming Convention * Object and class * Method overloading * Constructor * static keyword * this keyword with 6 usage * Inheritance * Aggregation * Method Overriding * Covariant Return Type * super keyword * Instance Initializer block * final keyword * Abstract class * Interface * Runtime Polymorphism * Static and Dynamic Binding * Downcasting with instanceof operator * Package * Access Modifiers * Encapsulation * Object Cloning |

# Java Naming conventions

Java **naming convention** is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method etc.

But, it is not forced to follow. So, it is known as convention not rule.

All the classes, interfaces, packages, methods and fields of java programming language are given according to java naming convention.

## Advantage of naming conventions in java

By using standard Java naming conventions, you make your code easier to read for yourself and for other programmers. Readability of Java program is very important. It indicates that **less time** is spent to figure out what the code does.

|  |  |
| --- | --- |
| **Name** | **Convention** |
| class name | should start with uppercase letter and be a noun e.g. String, Color, Button, System, Thread etc. |
| interface name | should start with uppercase letter and be an adjective e.g. Runnable, Remote, ActionListener etc. |
| method name | should start with lowercase letter and be a verb e.g. actionPerformed(), main(), print(), println() etc. |
| variable name | should start with lowercase letter e.g. firstName, orderNumber etc. |
| package name | should be in lowercase letter e.g. java, lang, sql, util etc. |
| constants name | should be in uppercase letter. e.g. RED, YELLOW, MAX\_PRIORITY etc. |

## CamelCase in java naming conventions

Java follows camelcase syntax for naming the class, interface, method and variable.

If name is combined with two words, second word will start with uppercase letter always e.g. actionPerformed(), firstName, ActionEvent, ActionListener etc.

# Object and Class in Java

1. [Object in Java](https://www.javatpoint.com/object-and-class-in-java#object)
2. [Class in Java](https://www.javatpoint.com/object-and-class-in-java#class)
3. [Instace Variable in Java](https://www.javatpoint.com/object-and-class-in-java#objectinstancevariable)
4. [Method in Java](https://www.javatpoint.com/object-and-class-in-java#objectmethod)
5. [Example of Object and class that maintains the records of student](https://www.javatpoint.com/object-and-class-in-java#objectex2)
6. [Annonymous Object](https://www.javatpoint.com/object-and-class-in-java#objectannonymous)

In this page, we will learn about java objects and classes. In object-oriented programming technique, we design a program using objects and classes.

Object is the physical as well as logical entity whereas class is the logical entity only.

### **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.

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.

### **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:**

1. **class** <class\_name>{
2. field;
3. method;
4. }

### **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 function 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.

### **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*

1. **class** Student{
2. **int** id;//field or data member or instance variable
3. String name;
5. **public** **static** **void** main(String args[]){
6. Student s1=**new** Student();//creating an object of Student
7. System.out.println(s1.id);//accessing member through reference variable
8. System.out.println(s1.name);
9. }
10. }

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

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*

1. **class** Student{
2. **int** id;
3. String name;
4. }
5. **class** TestStudent1{
6. **public** **static** **void** main(String args[]){
7. Student s1=**new** Student();
8. System.out.println(s1.id);
9. System.out.println(s1.name);
10. }
11. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestStudent1)

Output:

0

null

## 3 Ways to initialize object

There are 3 ways to initialize object in java.

1. By reference variable
2. By method
3. By constructor

### **1) 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*

1. **class** Student{
2. **int** id;
3. String name;
4. }
5. **class** TestStudent2{
6. **public** **static** **void** main(String args[]){
7. Student s1=**new** Student();
8. s1.id=101;
9. s1.name="Sonoo";
10. System.out.println(s1.id+" "+s1.name);//printing members with a white space
11. }
12. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestStudent2)

Output:

101 Sonoo

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

*File: TestStudent3.java*

1. **class** Student{
2. **int** id;
3. String name;
4. }
5. **class** TestStudent3{
6. **public** **static** **void** main(String args[]){
7. //Creating objects
8. Student s1=**new** Student();
9. Student s2=**new** Student();
10. //Initializing objects
11. s1.id=101;
12. s1.name="Sonoo";
13. s2.id=102;
14. s2.name="Amit";
15. //Printing data
16. System.out.println(s1.id+" "+s1.name);
17. System.out.println(s2.id+" "+s2.name);
18. }
19. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestStudent3)

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*

1. **class** Student{
2. **int** rollno;
3. String name;
4. **void** insertRecord(**int** r, String n){
5. rollno=r;
6. name=n;
7. }
8. **void** displayInformation(){System.out.println(rollno+" "+name);}
9. }
10. **class** TestStudent4{
11. **public** **static** **void** main(String args[]){
12. Student s1=**new** Student();
13. Student s2=**new** Student();
14. s1.insertRecord(111,"Karan");
15. s2.insertRecord(222,"Aryan");
16. s1.displayInformation();
17. s2.displayInformation();
18. }
19. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestStudent4)

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*

1. **class** Employee{
2. **int** id;
3. String name;
4. **float** salary;
5. **void** insert(**int** i, String n, **float** s) {
6. id=i;
7. name=n;
8. salary=s;
9. }
10. **void** display(){System.out.println(id+" "+name+" "+salary);}
11. }
12. **public** **class** TestEmployee {
13. **public** **static** **void** main(String[] args) {
14. Employee e1=**new** Employee();
15. Employee e2=**new** Employee();
16. Employee e3=**new** Employee();
17. e1.insert(101,"ajeet",45000);
18. e2.insert(102,"irfan",25000);
19. e3.insert(103,"nakul",55000);
20. e1.display();
21. e2.display();
22. e3.display();
23. }
24. }

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

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*

1. **class** Rectangle{
2. **int** length;
3. **int** width;
4. **void** insert(**int** l, **int** w){
5. length=l;
6. width=w;
7. }
8. **void** calculateArea(){System.out.println(length\*width);}
9. }
10. **class** TestRectangle1{
11. **public** **static** **void** main(String args[]){
12. Rectangle r1=**new** Rectangle();
13. Rectangle r2=**new** Rectangle();
14. r1.insert(11,5);
15. r2.insert(3,15);
16. r1.calculateArea();
17. r2.calculateArea();
18. }
19. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestRectangle1)

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:

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

Calling method through reference:

1. Calculation c=**new** Calculation();
2. c.fact(5);

Calling method through anonymous object

1. **new** Calculation().fact(5);

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

1. **class** Calculation{
2. **void** fact(**int**  n){
3. **int** fact=1;
4. **for**(**int** i=1;i<=n;i++){
5. fact=fact\*i;
6. }
7. System.out.println("factorial is "+fact);
8. }
9. **public** **static** **void** main(String args[]){
10. **new** Calculation().fact(5);//calling method with anonymous object
11. }
12. }

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:

1. **int** a=10, b=20;

Initialization of refernce variables:

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

Let's see the example:

1. **class** Rectangle{
2. **int** length;
3. **int** width;
4. **void** insert(**int** l,**int** w){
5. length=l;
6. width=w;
7. }
8. **void** calculateArea(){System.out.println(length\*width);}
9. }
10. **class** TestRectangle2{
11. **public** **static** **void** main(String args[]){
12. Rectangle r1=**new** Rectangle(),r2=**new** Rectangle();//creating two objects
13. r1.insert(11,5);
14. r2.insert(3,15);
15. r1.calculateArea();
16. r2.calculateArea();
17. }
18. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestRectangle2)

Output:

55

45

### **Real World Example: Account**

*File: TestAccount.java*

1. **class** Account{
2. **int** acc\_no;
3. String name;
4. **float** amount;
5. **void** insert(**int** a,String n,**float** amt){
6. acc\_no=a;
7. name=n;
8. amount=amt;
9. }
10. **void** deposit(**float** amt){
11. amount=amount+amt;
12. System.out.println(amt+" deposited");
13. }
14. **void** withdraw(**float** amt){
15. **if**(amount<amt){
16. System.out.println("Insufficient Balance");
17. }**else**{
18. amount=amount-amt;
19. System.out.println(amt+" withdrawn");
20. }
21. }
22. **void** checkBalance(){System.out.println("Balance is: "+amount);}
23. **void** display(){System.out.println(acc\_no+" "+name+" "+amount);}
24. }
26. **class** TestAccount{
27. **public** **static** **void** main(String[] args){
28. Account a1=**new** Account();
29. a1.insert(832345,"Ankit",1000);
30. a1.display();
31. a1.checkBalance();
32. a1.deposit(40000);
33. a1.checkBalance();
34. a1.withdraw(15000);
35. a1.checkBalance();
36. }}

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

Output:

832345 Ankit 1000.0

Balance is: 1000.0

40000.0 deposited

Balance is: 41000.0

15000.0 withdrawn

Balance is: 26000.0

|  |  |  |
| --- | --- | --- |
| Constructor in Java  1. [Types of constructors](https://www.javatpoint.com/java-constructor#constypes)    1. [Default Constructor](https://www.javatpoint.com/java-constructor#consdef)    2. [Parameterized Constructor](https://www.javatpoint.com/java-constructor#conspara) 2. [Constructor Overloading](https://www.javatpoint.com/java-constructor#consoverloading) 3. [Does constructor return any value](https://www.javatpoint.com/java-constructor#consdoesreturn) 4. [Copying the values of one object into another](https://www.javatpoint.com/java-constructor#conscopy) 5. [Does constructor perform other task instead initialization](https://www.javatpoint.com/java-constructor#consothertask)   In Java, constructor is a block of codes similar to method. It is called when an instance of object is created and memory is allocated for the object.  It is a special type of method which is used to initialize the object. **When a constructor is called** Everytime an object is created using new() keyword, atleast one constructor is called. It is called a default constructor.  **Note:** It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any. **Rules for creating java constructor** There are basically two rules defined for the constructor.   1. Constructor name must be same as its class name 2. Constructor must have no explicit return type  **Types of java constructors** There are two types of constructors in java:   1. Default constructor (no-arg constructor) 2. Parameterized constructor   java constructor Java Default Constructor A constructor is called "Default Constructor" when it doesn't have any parameter. **Syntax of default constructor:**  1. <class\_name>(){}  Example of default constructor  |  | | --- | | In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation. |  1. **class** Bike1{ 2. Bike1(){System.out.println("Bike is created");} 3. **public** **static** **void** main(String args[]){ 4. Bike1 b=**new** Bike1(); 5. } 6. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Bike1)  Output:  Bike is created Rule: If there is no constructor in a class, compiler automatically creates a default constructor. default constructor **Q) What is the purpose of default constructor?** Default constructor is used to provide the default values to the object like 0, null etc. depending on the type. **Example of default constructor that displays the default values**  1. **class** Student3{ 2. **int** id; 3. String name; 5. **void** display(){System.out.println(id+" "+name);} 7. **public** **static** **void** main(String args[]){ 8. Student3 s1=**new** Student3(); 9. Student3 s2=**new** Student3(); 10. s1.display(); 11. s2.display(); 12. } 13. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Student3)  Output:  0 null  0 null  **Explanation:**In the above class,you are not creating any constructor so compiler provides you a default constructor.Here 0 and null values are provided by default constructor. **Java parameterized constructor** A constructor which has a specific number of parameters is called parameterized constructor. **Why use parameterized constructor?**  |  | | --- | | Parameterized constructor is used to provide different values to the distinct objects. |  **Example of parameterized constructor** In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor. |

1. **class** Student4{
2. **int** id;
3. String name;
5. Student4(**int** i,String n){
6. id = i;
7. name = n;
8. }
9. **void** display(){System.out.println(id+" "+name);}
11. **public** **static** **void** main(String args[]){
12. Student4 s1 = **new** Student4(111,"Karan");
13. Student4 s2 = **new** Student4(222,"Aryan");
14. s1.display();
15. s2.display();
16. }
17. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Student4)

Output:

111 Karan

222 Aryan

## Constructor Overloading in Java

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

### **Example of Constructor Overloading**

1. **class** Student5{
2. **int** id;
3. String name;
4. **int** age;
5. Student5(**int** i,String n){
6. id = i;
7. name = n;
8. }
9. Student5(**int** i,String n,**int** a){
10. id = i;
11. name = n;
12. age=a;
13. }
14. **void** display(){System.out.println(id+" "+name+" "+age);}
16. **public** **static** **void** main(String args[]){
17. Student5 s1 = **new** Student5(111,"Karan");
18. Student5 s2 = **new** Student5(222,"Aryan",25);
19. s1.display();
20. s2.display();
21. }
22. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Student5)

Output:

111 Karan 0

222 Aryan 25

## Difference between constructor and method in java

There are many differences between constructors and methods. They are given below.

|  |  |
| --- | --- |
| **Java Constructor** | **Java Method** |
| Constructor is used to initialize the state of an object. | Method is used to expose behaviour of an object. |
| Constructor must not have return type. | Method must have return type. |
| Constructor is invoked implicitly. | Method is invoked explicitly. |
| The java compiler provides a default constructor if you don't have any constructor. | Method is not provided by compiler in any case. |
| Constructor name must be same as the class name. | Method name may or may not be same as class name. |

## Java Copy Constructor

There is no copy constructor in java. But, we can copy the values of one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in java. They are:

* By constructor
* By assigning the values of one object into another
* By clone() method of Object class

In this example, we are going to copy the values of one object into another using java constructor.

1. **class** Student6{
2. **int** id;
3. String name;
4. Student6(**int** i,String n){
5. id = i;
6. name = n;
7. }
9. Student6(Student6 s){
10. id = s.id;
11. name =s.name;
12. }
13. **void** display(){System.out.println(id+" "+name);}
15. **public** **static** **void** main(String args[]){
16. Student6 s1 = **new** Student6(111,"Karan");
17. Student6 s2 = **new** Student6(s1);
18. s1.display();
19. s2.display();
20. }
21. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Student6)

Output:

111 Karan

111 Karan

## Copying values without constructor

We can copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor.

1. **class** Student7{
2. **int** id;
3. String name;
4. Student7(**int** i,String n){
5. id = i;
6. name = n;
7. }
8. Student7(){}
9. **void** display(){System.out.println(id+" "+name);}
11. **public** **static** **void** main(String args[]){
12. Student7 s1 = **new** Student7(111,"Karan");
13. Student7 s2 = **new** Student7();
14. s2.id=s1.id;
15. s2.name=s1.name;
16. s1.display();
17. s2.display();
18. }
19. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Student7)

Output:

111 Karan

111 Karan

### **Q) Does constructor return any value?**

**Ans:**yes, that is current class instance (You cannot use return type yet it returns a value).

### **Can constructor perform other tasks instead of initialization?**

Yes, like object creation, starting a thread, calling method etc. You can perform any operation in the constructor as you perform in the method.

# Java static keyword

1. [Static variable](https://www.javatpoint.com/static-keyword-in-java#staticv)
2. [Program of counter without static variable](https://www.javatpoint.com/static-keyword-in-java#staticvcounter1)
3. [Program of counter with static variable](https://www.javatpoint.com/static-keyword-in-java#staticvcounter2)
4. [Static method](https://www.javatpoint.com/static-keyword-in-java#staticm)
5. [Restrictions for static method](https://www.javatpoint.com/static-keyword-in-java#staticmr)
6. [Why main method is static ?](https://www.javatpoint.com/static-keyword-in-java#staticwhymain)
7. [Static block](https://www.javatpoint.com/static-keyword-in-java#staticblock)
8. [Can we execute a program without main method ?](https://www.javatpoint.com/static-keyword-in-java#staticwithoutmain)

The **static keyword** in java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class.

The static can be:

1. variable (also known as class variable)
2. method (also known as class method)
3. block
4. nested class

## 1) Java static variable

If you declare any variable as static, it is known static variable.

* The static variable can be used to refer the common property of all objects (that is not unique for each object) e.g. company name of employees,college name of students etc.
* The static variable gets memory only once in class area at the time of class loading.

### **Advantage of static variable**

It makes your program **memory efficient** (i.e it saves memory).

#### Understanding problem without static variable

1. **class** Student{
2. **int** rollno;
3. String name;
4. String college="ITS";
5. }

Suppose there are 500 students in my college, now all instance data members will get memory each time when object is created.All student have its unique rollno and name so instance data member is good.Here, college refers to the common property of all objects.If we make it static,this field will get memory only once.

#### Java static property is shared to all objects.

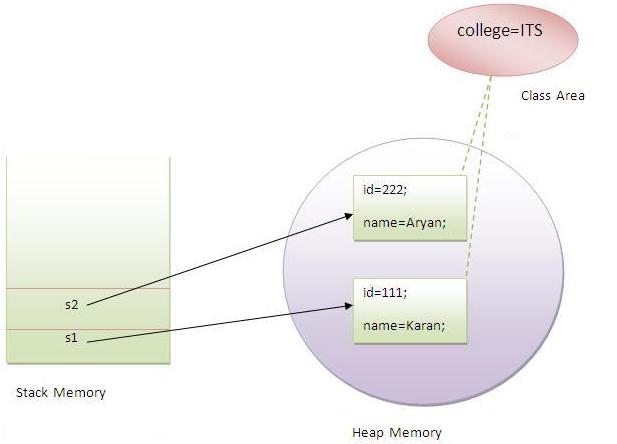
### **Example of static variable**

1. //Program of static variable
3. **class** Student8{
4. **int** rollno;
5. String name;
6. **static** String college ="ITS";
8. Student8(**int** r,String n){
9. rollno = r;
10. name = n;
11. }
12. **void** display (){System.out.println(rollno+" "+name+" "+college);}
14. **public** **static** **void** main(String args[]){
15. Student8 s1 = **new** Student8(111,"Karan");
16. Student8 s2 = **new** Student8(222,"Aryan");
18. s1.display();
19. s2.display();
20. }
21. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Student8)

Output:111 Karan ITS

222 Aryan ITS



### **Program of counter without static variable**

In this example, we have created an instance variable named count which is incremented in the constructor. Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable, if it is incremented, it won't reflect to other objects. So each objects will have the value 1 in the count variable.

1. **class** Counter{
2. **int** count=0;//will get memory when instance is created
4. Counter(){
5. count++;
6. System.out.println(count);
7. }
9. **public** **static** **void** main(String args[]){
11. Counter c1=**new** Counter();
12. Counter c2=**new** Counter();
13. Counter c3=**new** Counter();
15. }
16. }

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

Output:1

1

1

### **Program of counter by static variable**

|  |
| --- |
| As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value. |

1. **class** Counter2{
2. **static** **int** count=0;//will get memory only once and retain its value
4. Counter2(){
5. count++;
6. System.out.println(count);
7. }
9. **public** **static** **void** main(String args[]){
11. Counter2 c1=**new** Counter2();
12. Counter2 c2=**new** Counter2();
13. Counter2 c3=**new** Counter2();
15. }
16. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Counter2)

Output:1

2

3

## 2) Java static method

If you apply static keyword with any method, it is known as static method.

* A static method belongs to the class rather than object of a class.
* A static method can be invoked without the need for creating an instance of a class.
* static method can access static data member and can change the value of it.

### **Example of static method**

1. //Program of changing the common property of all objects(static field).
3. **class** Student9{
4. **int** rollno;
5. String name;
6. **static** String college = "ITS";
8. **static** **void** change(){
9. college = "BBDIT";
10. }
12. Student9(**int** r, String n){
13. rollno = r;
14. name = n;
15. }
17. **void** display (){System.out.println(rollno+" "+name+" "+college);}
19. **public** **static** **void** main(String args[]){
20. Student9.change();
22. Student9 s1 = **new** Student9 (111,"Karan");
23. Student9 s2 = **new** Student9 (222,"Aryan");
24. Student9 s3 = **new** Student9 (333,"Sonoo");
26. s1.display();
27. s2.display();
28. s3.display();
29. }
30. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Student9)

Output:111 Karan BBDIT

222 Aryan BBDIT

333 Sonoo BBDIT

### **Another example of static method that performs normal calculation**

1. //Program to get cube of a given number by static method
3. **class** Calculate{
4. **static** **int** cube(**int** x){
5. **return** x\*x\*x;
6. }
8. **public** **static** **void** main(String args[]){
9. **int** result=Calculate.cube(5);
10. System.out.println(result);
11. }
12. }

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

Output:125

### **Restrictions for static method**

|  |
| --- |
| There are two main restrictions for the static method. They are: |

|  |
| --- |
| 1. The static method can not use non static data member or call non-static method directly. 2. this and super cannot be used in static context. |

1. **class** A{
2. **int** a=40;//non static
4. **public** **static** **void** main(String args[]){
5. System.out.println(a);
6. }
7. }

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

Output:Compile Time Error

### **Q) why java main method is static?**

|  |
| --- |
| Ans) because object is not required to call static method if it were non-static method, jvm create object first then call main() method that will lead the problem of extra memory allocation. |

## 3) Java static block

* Is used to initialize the static data member.
* It is executed before main method at the time of classloading.

### **Example of static block**

1. **class** A2{
2. **static**{System.out.println("static block is invoked");}
3. **public** **static** **void** main(String args[]){
4. System.out.println("Hello main");
5. }
6. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=A2)

Output:static block is invoked

Hello main

### **Q) Can we execute a program without main() method?**

Ans) Yes, one of the way is static block but in previous version of JDK not in JDK 1.7.

1. **class** A3{
2. **static**{
3. System.out.println("static block is invoked");
4. System.exit(0);
5. }
6. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=A3)

Output:static block is invoked (if not JDK7)

In JDK7 and above, output will be:

Output:Error: Main method not found in class A3, please define the main method as:

public static void main(String[] args)

# this keyword in java

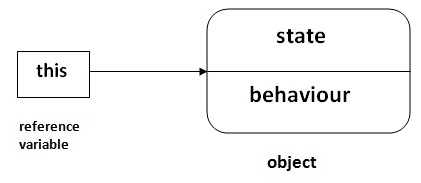
There can be a lot of usage of **java this keyword**. In java, this is a **reference variable** that refers to the current object.

## Usage of java this keyword

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

**Suggestion:** If you are beginner to java, lookup only three usage of this keyword.



### **1) this: to refer current class instance variable**

The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

#### Understanding the problem without this keyword

|  |
| --- |
| Let's understand the problem if we don't use this keyword by the example given below: |

1. **class** Student{
2. **int** rollno;
3. String name;
4. **float** fee;
5. Student(**int** rollno,String name,**float** fee){
6. rollno=rollno;
7. name=name;
8. fee=fee;
9. }
10. **void** display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
12. **class** TestThis1{
13. **public** **static** **void** main(String args[]){
14. Student s1=**new** Student(111,"ankit",5000f);
15. Student s2=**new** Student(112,"sumit",6000f);
16. s1.display();
17. s2.display();
18. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestThis1)

Output:

0 null 0.0

0 null 0.0

In the above example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

#### Solution of the above problem by this keyword

1. **class** Student{
2. **int** rollno;
3. String name;
4. **float** fee;
5. Student(**int** rollno,String name,**float** fee){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.fee=fee;
9. }
10. **void** display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
13. **class** TestThis2{
14. **public** **static** **void** main(String args[]){
15. Student s1=**new** Student(111,"ankit",5000f);
16. Student s2=**new** Student(112,"sumit",6000f);
17. s1.display();
18. s2.display();
19. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestThis2)

Output:

111 ankit 5000

112 sumit 6000

If local variables(formal arguments) and instance variables are different, there is no need to use this keyword like in the following program:

#### Program where this keyword is not required

1. **class** Student{
2. **int** rollno;
3. String name;
4. **float** fee;
5. Student(**int** r,String n,**float** f){
6. rollno=r;
7. name=n;
8. fee=f;
9. }
10. **void** display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
13. **class** TestThis3{
14. **public** **static** **void** main(String args[]){
15. Student s1=**new** Student(111,"ankit",5000f);
16. Student s2=**new** Student(112,"sumit",6000f);
17. s1.display();
18. s2.display();
19. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestThis3)

Output:

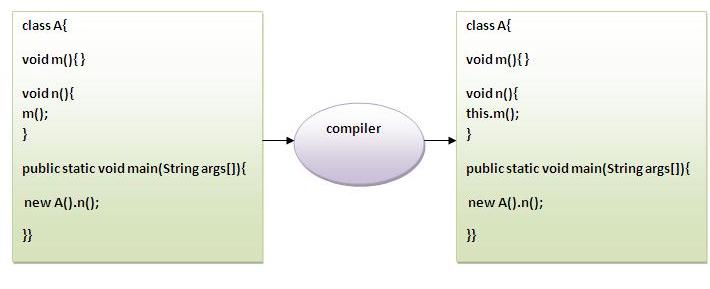
111 ankit 5000

112 sumit 6000

#### It is better approach to use meaningful names for variables. So we use same name for instance variables and parameters in real time, and always use this keyword.

### **2) this: to invoke current class method**

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method. Let's see the example



1. **class** A{
2. **void** m(){System.out.println("hello m");}
3. **void** n(){
4. System.out.println("hello n");
5. //m();//same as this.m()
6. **this**.m();
7. }
8. }
9. **class** TestThis4{
10. **public** **static** **void** main(String args[]){
11. A a=**new** A();
12. a.n();
13. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestThis4)

Output:

hello n

hello m

### **3) this() : to invoke current class constructor**

The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

**Calling default constructor from parameterized constructor:**

1. **class** A{
2. A(){System.out.println("hello a");}
3. A(**int** x){
4. **this**();
5. System.out.println(x);
6. }
7. }
8. **class** TestThis5{
9. **public** **static** **void** main(String args[]){
10. A a=**new** A(10);
11. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestThis5)

Output:

hello a

10

**Calling parameterized constructor from default constructor:**

1. **class** A{
2. A(){
3. **this**(5);
4. System.out.println("hello a");
5. }
6. A(**int** x){
7. System.out.println(x);
8. }
9. }
10. **class** TestThis6{
11. **public** **static** **void** main(String args[]){
12. A a=**new** A();
13. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestThis6)

Output:

5

hello a

### **Real usage of this() constructor call**

The this() constructor call should be used to reuse the constructor from the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining. Let's see the example given below that displays the actual use of this keyword.

1. **class** Student{
2. **int** rollno;
3. String name,course;
4. **float** fee;
5. Student(**int** rollno,String name,String course){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.course=course;
9. }
10. Student(**int** rollno,String name,String course,**float** fee){
11. **this**(rollno,name,course);//reusing constructor
12. **this**.fee=fee;
13. }
14. **void** display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
15. }
16. **class** TestThis7{
17. **public** **static** **void** main(String args[]){
18. Student s1=**new** Student(111,"ankit","java");
19. Student s2=**new** Student(112,"sumit","java",6000f);
20. s1.display();
21. s2.display();
22. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestThis7)

Output:

111 ankit java null

112 sumit java 6000

#### Rule: Call to this() must be the first statement in constructor.

1. **class** Student{
2. **int** rollno;
3. String name,course;
4. **float** fee;
5. Student(**int** rollno,String name,String course){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.course=course;
9. }
10. Student(**int** rollno,String name,String course,**float** fee){
11. **this**.fee=fee;
12. **this**(rollno,name,course);//C.T.Error
13. }
14. **void** display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
15. }
16. **class** TestThis8{
17. **public** **static** **void** main(String args[]){
18. Student s1=**new** Student(111,"ankit","java");
19. Student s2=**new** Student(112,"sumit","java",6000f);
20. s1.display();
21. s2.display();
22. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestThis8)

Compile Time Error: Call to this must be first statement in constructor

### **4) this: to pass as an argument in the method**

The this keyword can also be passed as an argument in the method. It is mainly used in the event handling. Let's see the example:

1. **class** S2{
2. **void** m(S2 obj){
3. System.out.println("method is invoked");
4. }
5. **void** p(){
6. m(**this**);
7. }
8. **public** **static** **void** main(String args[]){
9. S2 s1 = **new** S2();
10. s1.p();
11. }
12. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=S2)

Output:

method is invoked

### **Application of this that can be passed as an argument:**

In event handling (or) in a situation where we have to provide reference of a class to another one. It is used to reuse one object in many methods.

### **5) this: to pass as argument in the constructor call**

We can pass the this keyword in the constructor also. It is useful if we have to use one object in multiple classes. Let's see the example:

1. **class** B{
2. A4 obj;
3. B(A4 obj){
4. **this**.obj=obj;
5. }
6. **void** display(){
7. System.out.println(obj.data);//using data member of A4 class
8. }
9. }
11. **class** A4{
12. **int** data=10;
13. A4(){
14. B b=**new** B(**this**);
15. b.display();
16. }
17. **public** **static** **void** main(String args[]){
18. A4 a=**new** A4();
19. }
20. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=A4)

Output:10

### **6) this keyword can be used to return current class instance**

We can return this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive). Let's see the example:

### **Syntax of this that can be returned as a statement**

1. return\_type method\_name(){
2. **return** **this**;
3. }

### **Example of this keyword that you return as a statement from the method**

1. **class** A{
2. A getA(){
3. **return** **this**;
4. }
5. **void** msg(){System.out.println("Hello java");}
6. }
7. **class** Test1{
8. **public** **static** **void** main(String args[]){
9. **new** A().getA().msg();
10. }
11. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Test1)

Output:

Hello java

### **Proving this keyword**

|  |
| --- |
| Let's prove that this keyword refers to the current class instance variable. In this program, we are printing the reference variable and this, output of both variables are same. |

1. **class** A5{
2. **void** m(){
3. System.out.println(**this**);//prints same reference ID
4. }
5. **public** **static** **void** main(String args[]){
6. A5 obj=**new** A5();
7. System.out.println(obj);//prints the reference ID
8. obj.m();
9. }
10. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=A5)

Output:

A5@22b3ea59

A5@22b3ea59

INHERITANCE:

# 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. It is an important part of OPPs(Object Oriented programming system).

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

Inheritance represents the **IS-A relationship**, also known as parent-child relationship.

### **Why use inheritance in java**

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

### **Terms used in Inheritence**

* **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
* **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
* **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
* **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in previous class.

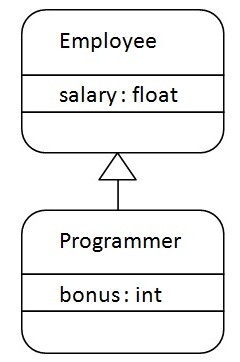
### **Syntax of Java Inheritance**

1. **class** Subclass-name **extends** Superclass-name
2. {
3. //methods and fields
4. }

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.

1. **class** Employee{
2. **float** salary=40000;
3. }
4. **class** Programmer **extends** Employee{
5. **int** bonus=10000;
6. **public** **static** **void** main(String args[]){
7. Programmer p=**new** Programmer();
8. System.out.println("Programmer salary is:"+p.salary);
9. System.out.println("Bonus of Programmer is:"+p.bonus);
10. }
11. }

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

Programmer salary is:40000.0

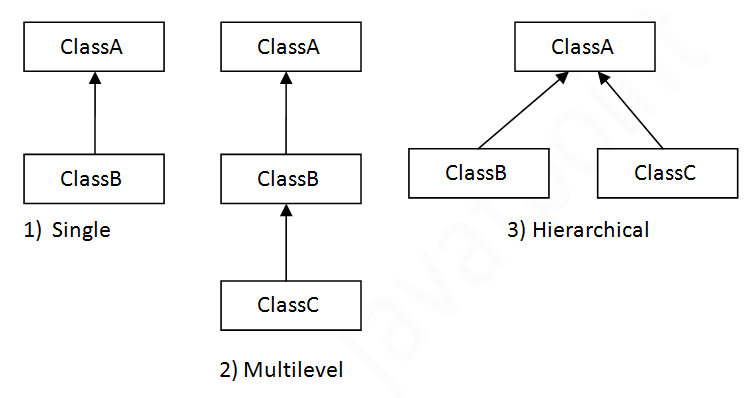
Bonus of programmer is:10000

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

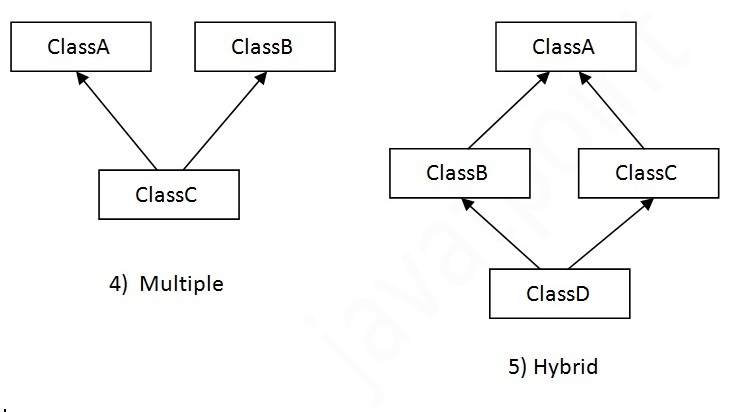
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

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*

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** bark(){System.out.println("barking...");}
6. }
7. **class** TestInheritance{
8. **public** **static** **void** main(String args[]){
9. Dog d=**new** Dog();
10. d.bark();
11. d.eat();
12. }}

Output:

barking...

eating...

## Multilevel Inheritance Example

*File: TestInheritance2.java*

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** bark(){System.out.println("barking...");}
6. }
7. **class** BabyDog **extends** Dog{
8. **void** weep(){System.out.println("weeping...");}
9. }
10. **class** TestInheritance2{
11. **public** **static** **void** main(String args[]){
12. BabyDog d=**new** BabyDog();
13. d.weep();
14. d.bark();
15. d.eat();
16. }}

Output:

weeping...

barking...

eating...

## Hierarchical Inheritance Example

*File: TestInheritance3.java*

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** bark(){System.out.println("barking...");}
6. }
7. **class** Cat **extends** Animal{
8. **void** meow(){System.out.println("meowing...");}
9. }
10. **class** TestInheritance3{
11. **public** **static** **void** main(String args[]){
12. Cat c=**new** Cat();
13. c.meow();
14. c.eat();
15. //c.bark();//C.T.Error
16. }}

Output:

meowing...

eating...

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

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

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

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

1. **class** A{
2. **void** msg(){System.out.println("Hello");}
3. }
4. **class** B{
5. **void** msg(){System.out.println("Welcome");}
6. }
7. **class** C **extends** A,B{//suppose if it were
9. Public Static **void** main(String args[]){
10. C obj=**new** C();
11. obj.msg();//Now which msg() method would be invoked?
12. }
13. }

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

Compile Time Error

# Aggregation in Java

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

Consider a situation, Employee object contains many informations such as id, name, emailId etc. It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc. as given below.

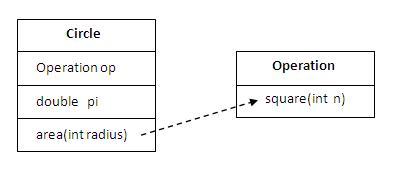
1. **class** Employee{
2. **int** id;
3. String name;
4. Address address;//Address is a class
5. ...
6. }

In such case, Employee has an entity reference address, so relationship is Employee HAS-A address.

### **Why use Aggregation?**

* For Code Reusability.

### **Simple Example of Aggregation**



In this example, we have created the reference of Operation class in the Circle class.

1. **class** Operation{
2. **int** square(**int** n){
3. **return** n\*n;
4. }
5. }
7. **class** Circle{
8. Operation op;//aggregation
9. **double** pi=3.14;
11. **double** area(**int** radius){
12. op=**new** Operation();
13. **int** rsquare=op.square(radius);//code reusability (i.e. delegates the method call).
14. **return** pi\*rsquare;
15. }


19. **public** **static** **void** main(String args[]){
20. Circle c=**new** Circle();
21. **double** result=c.area(5);
22. System.out.println(result);
23. }
24. }

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

Output:78.5

### **When use Aggregation?**

* Code reuse is also best achieved by aggregation when there is no is-a relationship.
* Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

### **Understanding meaningful example of Aggregation**

In this example, Employee has an object of Address, address object contains its own informations such as city, state, country etc. In such case relationship is Employee HAS-A address.

#### Address.java

1. **public** **class** Address {
2. String city,state,country;
4. **public** Address(String city, String state, String country) {
5. **this**.city = city;
6. **this**.state = state;
7. **this**.country = country;
8. }
10. }

#### Emp.java

1. **public** **class** Emp {
2. **int** id;
3. String name;
4. Address address;
6. **public** Emp(**int** id, String name,Address address) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.address=address;
10. }
12. **void** display(){
13. System.out.println(id+" "+name);
14. System.out.println(address.city+" "+address.state+" "+address.country);
15. }
17. **public** **static** **void** main(String[] args) {
18. Address address1=**new** Address("gzb","UP","india");
19. Address address2=**new** Address("gno","UP","india");
21. Emp e=**new** Emp(111,"varun",address1);
22. Emp e2=**new** Emp(112,"arun",address2);
24. e.display();
25. e2.display();
27. }
28. }

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

Output:111 varun

gzb UP india

112 arun

gno UP india

# Method Overloading in Java

1. [Different ways to overload the method](https://www.javatpoint.com/method-overloading-in-java#monumberofways)
2. [By changing the no. of arguments](https://www.javatpoint.com/method-overloading-in-java#mobynumber)
3. [By changing the datatype](https://www.javatpoint.com/method-overloading-in-java#mobydatatype)
4. [Why method overloading is not possible by changing the return type](https://www.javatpoint.com/method-overloading-in-java#moreturntype)
5. [Can we overload the main method](https://www.javatpoint.com/method-overloading-in-java#momainmethod)
6. [method overloading with Type Promotion](https://www.javatpoint.com/method-overloading-in-java#motypepromotion)

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

So, we perform method overloading to figure out the program quickly.



## Advantage of method overloading

Method overloading increases the readability of the program.

### **Different ways to overload the method**

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

#### In java, Method Overloading is not possible by changing the return type of the method only.

### **1) Method Overloading: changing no. of arguments**

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

In this example, we are creating static methods so that we don't need to create instance for calling methods.

1. **class** Adder{
2. **static** **int** add(**int** a,**int** b){**return** a+b;}
3. **static** **int** add(**int** a,**int** b,**int** c){**return** a+b+c;}
4. }
5. **class** TestOverloading1{
6. **public** **static** **void** main(String[] args){
7. System.out.println(Adder.add(11,11));
8. System.out.println(Adder.add(11,11,11));
9. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestOverloading1)

Output:

22

33

### **2) Method Overloading: changing data type of arguments**

In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

1. **class** Adder{
2. **static** **int** add(**int** a, **int** b){**return** a+b;}
3. **static** **double** add(**double** a, **double** b){**return** a+b;}
4. }
5. **class** TestOverloading2{
6. **public** **static** **void** main(String[] args){
7. System.out.println(Adder.add(11,11));
8. System.out.println(Adder.add(12.3,12.6));
9. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestOverloading2)

Output:

22

24.9

### **Q) Why Method Overloading is not possible by changing the return type of method only?**

In java, method overloading is not possible by changing the return type of the method only because of ambiguity. Let's see how ambiguity may occur:

1. **class** Adder{
2. **static** **int** add(**int** a,**int** b){**return** a+b;}
3. **static** **double** add(**int** a,**int** b){**return** a+b;}
4. }
5. **class** TestOverloading3{
6. **public** **static** **void** main(String[] args){
7. System.out.println(Adder.add(11,11));//ambiguity
8. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestOverloading3)

Output:

Compile Time Error: method add(int,int) is already defined in class Adder

System.out.println(Adder.add(11,11)); //Here, how can java determine which sum() method should be called?

#### Note: Compile Time Error is better than Run Time Error. So, java compiler renders compiler time error if you declare the same method having same parameters.

### **Can we overload java main() method?**

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But JVM calls main() method which receives string array as arguments only. Let's see the simple example:

1. **class** TestOverloading4{
2. **public** **static** **void** main(String[] args){System.out.println("main with String[]");}
3. **public** **static** **void** main(String args){System.out.println("main with String");}
4. **public** **static** **void** main(){System.out.println("main without args");}
5. }

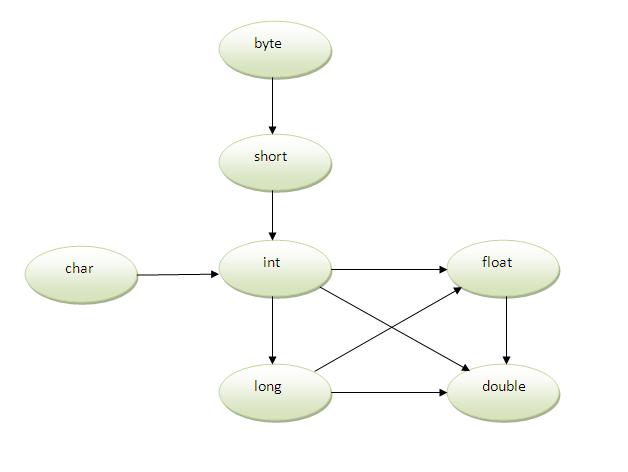
[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestOverloading4)

Output:

main with String[]

## Method Overloading and Type Promotion

One type is promoted to another implicitly if no matching datatype is found. Let's understand the concept by the figure given below:



As displayed in the above diagram, byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int,long,float or double. The char datatype can be promoted to int,long,float or double and so on.

### **Example of Method Overloading with TypePromotion**

1. **class** OverloadingCalculation1{
2. **void** sum(**int** a,**long** b){System.out.println(a+b);}
3. **void** sum(**int** a,**int** b,**int** c){System.out.println(a+b+c);}
5. **public** **static** **void** main(String args[]){
6. OverloadingCalculation1 obj=**new** OverloadingCalculation1();
7. obj.sum(20,20);//now second int literal will be promoted to long
8. obj.sum(20,20,20);
10. }
11. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=OverloadingCalculation1)

Output:40

60

### **Example of Method Overloading with Type Promotion if matching found**

If there are matching type arguments in the method, type promotion is not performed.

1. **class** OverloadingCalculation2{
2. **void** sum(**int** a,**int** b){System.out.println("int arg method invoked");}
3. **void** sum(**long** a,**long** b){System.out.println("long arg method invoked");}
5. **public** **static** **void** main(String args[]){
6. OverloadingCalculation2 obj=**new** OverloadingCalculation2();
7. obj.sum(20,20);//now int arg sum() method gets invoked
8. }
9. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=OverloadingCalculation2)

Output:int arg method invoked

### **Example of Method Overloading with Type Promotion in case of ambiguity**

If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be ambiguity.

1. **class** OverloadingCalculation3{
2. **void** sum(**int** a,**long** b){System.out.println("a method invoked");}
3. **void** sum(**long** a,**int** b){System.out.println("b method invoked");}
5. **public** **static** **void** main(String args[]){
6. OverloadingCalculation3 obj=**new** OverloadingCalculation3();
7. obj.sum(20,20);//now ambiguity
8. }
9. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=OverloadingCalculation3)

Output:Compile Time Error

#### One type is not de-promoted implicitly for example double cannot be depromoted to any type implicitly.

# Method Overriding in Java

1. [Understanding problem without method overriding](https://www.javatpoint.com/method-overriding-in-java#moverproblem)
2. [Can we override the static method](https://www.javatpoint.com/method-overriding-in-java#movercanstatic)
3. [method overloading vs method overriding](https://www.javatpoint.com/method-overriding-in-java#moverdiff)

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in java**.

In other words, If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

### **Usage of Java Method Overriding**

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

#### Rules for Java Method Overriding

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

### **Understanding the problem without method overriding**

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

1. **class** Vehicle{
2. **void** run(){System.out.println("Vehicle is running");}
3. }
4. **class** Bike **extends** Vehicle{
6. **public** **static** **void** main(String args[]){
7. Bike obj = **new** Bike();
8. obj.run();
9. }
10. }

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

Output:Vehicle is running

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

### **Example of method overriding**

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

1. **class** Vehicle{
2. **void** run(){System.out.println("Vehicle is running");}
3. }
4. **class** Bike2 **extends** Vehicle{
5. **void** run(){System.out.println("Bike is running safely");}
7. **public** **static** **void** main(String args[]){
8. Bike2 obj = **new** Bike2();
9. obj.run();
10. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Bike2)

Output:Bike is running safely

### **Real example of Java Method Overriding**

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

Java method overriding example of bank

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** Test2{
17. **public** **static** **void** main(String args[]){
18. SBI s=**new** SBI();
19. ICICI i=**new** ICICI();
20. AXIS a=**new** AXIS();
21. System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
22. System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
23. System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
24. }
25. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Test2)

Output:

SBI Rate of Interest: 8

ICICI Rate of Interest: 7

AXIS Rate of Interest: 9

### **Can we override static method?**

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

### **Why we cannot override static method?**

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

### **Can we override java main method?**

No, because main is a static method.

## Difference between method Overloading and Method Overriding in java

[Click me for difference between method overloading and overriding](https://www.javatpoint.com/method-overloading-vs-method-overriding-in-java)

### **More topics on Method Overriding (Not For Beginners)**

[Method Overriding with Access Modifier](https://www.javatpoint.com/access-modifiers#accessoverriding)

Let's see the concept of method overriding with access modifier.

[Exception Handling with Method Overriding](https://www.javatpoint.com/exception-handling-with-method-overriding)

Let's see the concept of method overriding with exception handling.

# 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. Let's take a simple example:

#### Note: If you are beginner to java, skip this topic and return to it after OOPs concepts.

### **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. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=B1)

Output:welcome to covariant return type

As 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.

### **How is Covariant return types implemented?**

Java doesn't allow the return type based overloading but JVM always allows return type based overloading. JVM uses full signature of a method for lookup/resolution. Full signature means it includes return type in addition to argument types. i.e., a class can have two or more methods differing only by return type. javac uses this fact to implement covariant return types.

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.

1. **class** Animal{
2. String color="white";
3. }
4. **class** Dog **extends** Animal{
5. String color="black";
6. **void** printColor(){
7. System.out.println(color);//prints color of Dog class
8. System.out.println(**super**.color);//prints color of Animal class
9. }
10. }
11. **class** TestSuper1{
12. **public** **static** **void** main(String args[]){
13. Dog d=**new** Dog();
14. d.printColor();
15. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestSuper1)

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.

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** eat(){System.out.println("eating bread...");}
6. **void** bark(){System.out.println("barking...");}
7. **void** work(){
8. **super**.eat();
9. bark();
10. }
11. }
12. **class** TestSuper2{
13. **public** **static** **void** main(String args[]){
14. Dog d=**new** Dog();
15. d.work();
16. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestSuper2)

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:

1. **class** Animal{
2. Animal(){System.out.println("animal is created");}
3. }
4. **class** Dog **extends** Animal{
5. Dog(){
6. **super**();
7. System.out.println("dog is created");
8. }
9. }
10. **class** TestSuper3{
11. **public** **static** **void** main(String args[]){
12. Dog d=**new** Dog();
13. }}

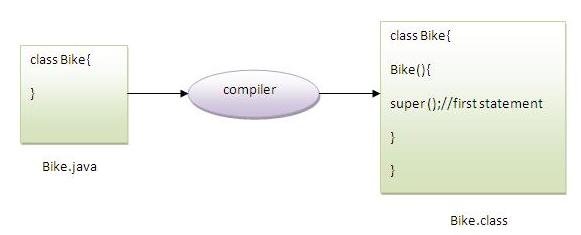
[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestSuper3)

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. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestSuper4)

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.

1. **class** Person{
2. **int** id;
3. String name;
4. Person(**int** id,String name){
5. **this**.id=id;
6. **this**.name=name;
7. }
8. }
9. **class** Emp **extends** Person{
10. **float** salary;
11. Emp(**int** id,String name,**float** salary){
12. **super**(id,name);//reusing parent constructor
13. **this**.salary=salary;
14. }
15. **void** display(){System.out.println(id+" "+name+" "+salary);}
16. }
17. **class** TestSuper5{
18. **public** **static** **void** main(String[] args){
19. Emp e1=**new** Emp(1,"ankit",45000f);
20. e1.display();
21. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestSuper5)

Output:

1 ankit 45000

# Instance initializer block

1. [Instance initializer block](https://www.javatpoint.com/instance-initializer-block)
2. [Example of Instance initializer block](https://www.javatpoint.com/instance-initializer-block#instanceinitializerex)
3. [What is invoked firstly instance initializer block or constructor?](https://www.javatpoint.com/instance-initializer-block#instanceinitializerfirstly)
4. [Rules for instance initializer block](https://www.javatpoint.com/instance-initializer-block#instanceinitializerrules)
5. [Program of instance initializer block that is invoked after super()](https://www.javatpoint.com/instance-initializer-block#instanceinitializersuper)

|  |
| --- |
| **Instance Initializer block** is used to initialize the instance data member. It run each time when object of the class is created. |
| The initialization of the instance variable can be done directly but there can be performed extra operations while initializing the instance variable in the instance initializer block. |

#### Que) What is the use of instance initializer block while we can directly assign a value in instance data member? For example:

1. **class** Bike{
2. **int** speed=100;
3. }

## Why use instance initializer block?

|  |
| --- |
| Suppose I have to perform some operations while assigning value to instance data member e.g. a for loop to fill a complex array or error handling etc. |

### **Example of instance initializer block**

|  |
| --- |
| Let's see the simple example of instance initializer block that performs initialization. |

1. **class** Bike7{
2. **int** speed;
4. Bike7(){System.out.println("speed is "+speed);}
6. {speed=100;}
8. **public** **static** **void** main(String args[]){
9. Bike7 b1=**new** Bike7();
10. Bike7 b2=**new** Bike7();
11. }
12. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Bike7)

Output:speed is 100

speed is 100

|  |
| --- |
| There are three places in java where you can perform operations:   1. method 2. constructor 3. block |

## What is invoked first, instance initializer block or constructor?

1. **class** Bike8{
2. **int** speed;
4. Bike8(){System.out.println("constructor is invoked");}
6. {System.out.println("instance initializer block invoked");}
8. **public** **static** **void** main(String args[]){
9. Bike8 b1=**new** Bike8();
10. Bike8 b2=**new** Bike8();
11. }
12. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Bike8)

Output:instance initializer block invoked

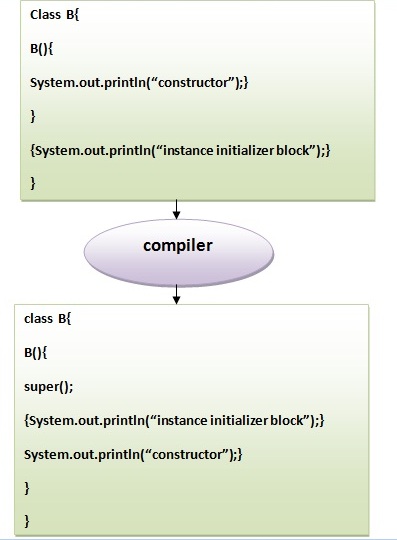
constructor is invoked

instance initializer block invoked

constructor is invoked

|  |
| --- |
| In the above example, it seems that instance initializer block is firstly invoked but NO. Instance intializer block is invoked at the time of object creation. The java compiler copies the instance initializer block in the constructor after the first statement super(). So firstly, constructor is invoked. Let's understand it by the figure given below: |

#### Note: The java compiler copies the code of instance initializer block in every constructor.



## Rules for instance initializer block :

|  |
| --- |
| There are mainly three rules for the instance initializer block. They are as follows: |

1. The instance initializer block is created when instance of the class is created.
2. The instance initializer block is invoked after the parent class constructor is invoked (i.e. after super() constructor call).
3. The instance initializer block comes in the order in which they appear.

## Program of instance initializer block that is invoked after super()

1. **class** A{
2. A(){
3. System.out.println("parent class constructor invoked");
4. }
5. }
6. **class** B2 **extends** A{
7. B2(){
8. **super**();
9. System.out.println("child class constructor invoked");
10. }
12. {System.out.println("instance initializer block is invoked");}
14. **public** **static** **void** main(String args[]){
15. B2 b=**new** B2();
16. }
17. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=B2)

Output:parent class constructor invoked

instance initializer block is invoked

child class constructor invoked

## Another example of instance block

1. **class** A{
2. A(){
3. System.out.println("parent class constructor invoked");
4. }
5. }
7. **class** B3 **extends** A{
8. B3(){
9. **super**();
10. System.out.println("child class constructor invoked");
11. }
13. B3(**int** a){
14. **super**();
15. System.out.println("child class constructor invoked "+a);
16. }
18. {System.out.println("instance initializer block is invoked");}
20. **public** **static** **void** main(String args[]){
21. B3 b1=**new** B3();
22. B3 b2=**new** B3(10);
23. }
24. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=B3)

Output:parent class constructor invoked

instance initializer block is invoked

child class constructor invoked

parent class constructor invoked

instance initializer block is invoked

child class constructor invoked 10

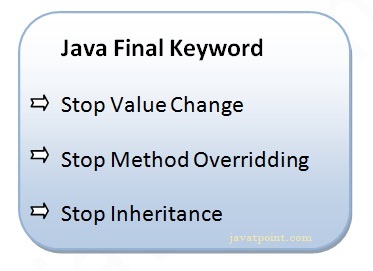
# Final Keyword In Java

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.

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** Honda1();
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

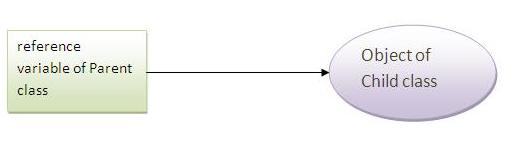
**Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

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.

## Java Runtime Polymorphism Example: Bank

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 are providing 8.4%, 7.3% and 9.7% rate of interest.

Java Runtime Polymorphism example of bank

#### Note: This example is also given in method overriding but there was no upcasting.

1. **class** Bank{
2. **float** getRateOfInterest(){**return** 0;}
3. }
4. **class** SBI **extends** Bank{
5. **float** getRateOfInterest(){**return** 8.4f;}
6. }
7. **class** ICICI **extends** Bank{
8. **float** getRateOfInterest(){**return** 7.3f;}
9. }
10. **class** AXIS **extends** Bank{
11. **float** getRateOfInterest(){**return** 9.7f;}
12. }
13. **class** TestPolymorphism{
14. **public** **static** **void** main(String args[]){
15. Bank b;
16. b=**new** SBI();
17. System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());
18. b=**new** ICICI();
19. System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());
20. b=**new** AXIS();
21. System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());
22. }
23. }

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

Output:

SBI Rate of Interest: 8.4

ICICI Rate of Interest: 7.3

AXIS Rate of Interest: 9.7

## Java Runtime Polymorphism Example: Shape

1. **class** Shape{
2. **void** draw(){System.out.println("drawing...");}
3. }
4. **class** Rectangle **extends** Shape{
5. **void** draw(){System.out.println("drawing rectangle...");}
6. }
7. **class** Circle **extends** Shape{
8. **void** draw(){System.out.println("drawing circle...");}
9. }
10. **class** Triangle **extends** Shape{
11. **void** draw(){System.out.println("drawing triangle...");}
12. }
13. **class** TestPolymorphism2{
14. **public** **static** **void** main(String args[]){
15. Shape s;
16. s=**new** Rectangle();
17. s.draw();
18. s=**new** Circle();
19. s.draw();
20. s=**new** Triangle();
21. s.draw();
22. }
23. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestPolymorphism2)

Output:

drawing rectangle...

drawing circle...

drawing triangle...

## Java Runtime Polymorphism Example: Animal

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** eat(){System.out.println("eating bread...");}
6. }
7. **class** Cat **extends** Animal{
8. **void** eat(){System.out.println("eating rat...");}
9. }
10. **class** Lion **extends** Animal{
11. **void** eat(){System.out.println("eating meat...");}
12. }
13. **class** TestPolymorphism3{
14. **public** **static** **void** main(String[] args){
15. Animal a;
16. a=**new** Dog();
17. a.eat();
18. a=**new** Cat();
19. a.eat();
20. a=**new** Lion();
21. a.eat();
22. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestPolymorphism3)

Output:

eating bread...

eating rat...

eating meat...

## 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. }
4. **class** Dog **extends** Animal{
5. **void** eat(){System.out.println("eating fruits");}
6. }
7. **class** BabyDog **extends** Dog{
8. **void** eat(){System.out.println("drinking milk");}
9. **public** **static** **void** main(String args[]){
10. Animal a1,a2,a3;
11. a1=**new** Animal();
12. a2=**new** Dog();
13. a3=**new** BabyDog();
14. a1.eat();
15. a2.eat();
16. a3.eat();
17. }
18. }

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

[**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.

### **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. |

# Java instanceof

1. [java instanceof](https://www.javatpoint.com/downcasting-with-instanceof-operator#instanceof)
2. [Example of instanceof operator](https://www.javatpoint.com/downcasting-with-instanceof-operator#instanceofex)
3. [Applying the instanceof operator with a variable the have null value](https://www.javatpoint.com/downcasting-with-instanceof-operator#instanceofnull)
4. [Downcasting with instanceof operator](https://www.javatpoint.com/downcasting-with-instanceof-operator#instanceofdowncasting)
5. [Downcasting without instanceof operator](https://www.javatpoint.com/downcasting-with-instanceof-operator#instanceofdowncastingwithout)

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

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

### **Simple example of java instanceof**

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

1. **class** Simple1{
2. **public** **static** **void** main(String args[]){
3. Simple1 s=**new** Simple1();
4. System.out.println(s **instanceof** Simple1);//true
5. }
6. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Simple1)

Output:true

An object of subclass type is also a type of parent class. For example, if Dog extends Animal then object of Dog can be referred by either Dog or Animal class.

## Another example of java instanceof operator

1. **class** Animal{}
2. **class** Dog1 **extends** Animal{//Dog inherits Animal
4. **public** **static** **void** main(String args[]){
5. Dog1 d=**new** Dog1();
6. System.out.println(d **instanceof** Animal);//true
7. }
8. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Dog1)

Output:true

## instanceof in java with a variable that have null value

If we apply instanceof operator with a variable that have null value, it returns false. Let's see the example given below where we apply instanceof operator with the variable that have null value.

1. **class** Dog2{
2. **public** **static** **void** main(String args[]){
3. Dog2 d=**null**;
4. System.out.println(d **instanceof** Dog2);//false
5. }
6. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Dog2)

Output:false

## Downcasting with java instanceof operator

When Subclass type refers to the object of Parent class, it is known as downcasting. If we perform it directly, compiler gives Compilation error. If you perform it by typecasting, ClassCastException is thrown at runtime. But if we use instanceof operator, downcasting is possible.

1. Dog d=**new** Animal();//Compilation error

If we perform downcasting by typecasting, ClassCastException is thrown at runtime.

1. Dog d=(Dog)**new** Animal();
2. //Compiles successfully but ClassCastException is thrown at runtime

### **Possibility of downcasting with instanceof**

Let's see the example, where downcasting is possible by instanceof operator.

1. **class** Animal { }
3. **class** Dog3 **extends** Animal {
4. **static** **void** method(Animal a) {
5. **if**(a **instanceof** Dog3){
6. Dog3 d=(Dog3)a;//downcasting
7. System.out.println("ok downcasting performed");
8. }
9. }
11. **public** **static** **void** main (String [] args) {
12. Animal a=**new** Dog3();
13. Dog3.method(a);
14. }
16. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Dog3)

Output:ok downcasting performed

### **Downcasting without the use of java instanceof**

Downcasting can also be performed without the use of instanceof operator as displayed in the following example:

1. **class** Animal { }
2. **class** Dog4 **extends** Animal {
3. **static** **void** method(Animal a) {
4. Dog4 d=(Dog4)a;//downcasting
5. System.out.println("ok downcasting performed");
6. }
7. **public** **static** **void** main (String [] args) {
8. Animal a=**new** Dog4();
9. Dog4.method(a);
10. }
11. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Dog4)

Output:ok downcasting performed

Let's take closer look at this, actual object that is referred by a, is an object of Dog class. So if we downcast it, it is fine. But what will happen if we write:

1. Animal a=**new** Animal();
2. Dog.method(a);
3. //Now ClassCastException but not in case of instanceof operator

### **Understanding Real use of instanceof in java**

Let's see the real use of instanceof keyword by the example given below.

1. **interface** Printable{}
2. **class** A **implements** Printable{
3. **public** **void** a(){System.out.println("a method");}
4. }
5. **class** B **implements** Printable{
6. **public** **void** b(){System.out.println("b method");}
7. }
9. **class** Call{
10. **void** invoke(Printable p){//upcasting
11. **if**(p **instanceof** A){
12. A a=(A)p;//Downcasting
13. a.a();
14. }
15. **if**(p **instanceof** B){
16. B b=(B)p;//Downcasting
17. b.b();
18. }
20. }
21. }//end of Call class
23. **class** Test4{
24. **public** **static** **void** main(String args[]){
25. Printable p=**new** B();
26. Call c=**new** Call();
27. c.invoke(p);
28. }
29. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Test4)

Output: b 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();
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*

1. **abstract** **class** Bank{
2. **abstract** **int** getRateOfInterest();
3. }
4. **class** SBI **extends** Bank{
5. **int** getRateOfInterest(){**return** 7;}
6. }
7. **class** PNB **extends** Bank{
8. **int** getRateOfInterest(){**return** 8;}
9. }
11. **class** TestBank{
12. **public** **static** **void** main(String args[]){
13. Bank b;
14. b=**new** SBI();
15. System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
16. b=**new** PNB();
17. System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
18. }}

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

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

# Interface in Java

1. [Interface](https://www.javatpoint.com/interface-in-java)
2. [Example of Interface](https://www.javatpoint.com/interface-in-java#interfaceex)
3. [Multiple inheritance by Interface](https://www.javatpoint.com/interface-in-java#interfacemultiple)
4. [Why multiple inheritance is supported in Interface while it is not supported in case of class.](https://www.javatpoint.com/interface-in-java#interfacewhynot)
5. [Marker Interface](https://www.javatpoint.com/interface-in-java#interfacemarker)
6. [Nested Interface](https://www.javatpoint.com/nested-interface)

An **interface in java** is a blueprint of a class. It has static constants and abstract methods.

The interface in java is **a mechanism to achieve abstraction**. There can be only abstract methods in the java interface not method body. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have methods and variables but the methods declared in interface contain only method signature, not body.

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

It cannot be instantiated just like abstract class.

## Why use Java interface?

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

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

## How to declare interface?

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

### **Syntax:**

1. **interface** <interface\_name>{
3. // declare constant fields
4. // declare methods that abstract
5. // by default.
6. }

## Java 8 Interface Improvement

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

## Internal addition by compiler

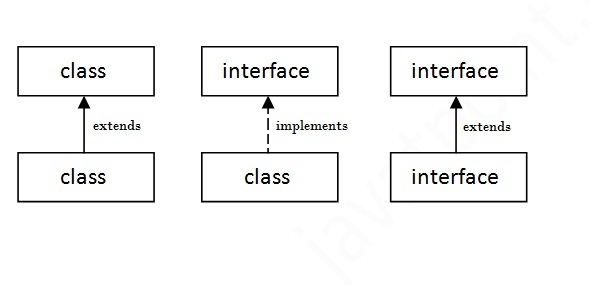
#### The java compiler adds public and abstract keywords before the interface method. More, it adds public, static and final keywords before data members.

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

interface in java

#### Understanding relationship between classes and interfaces

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



## Java Interface Example

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

1. **interface** printable{
2. **void** print();
3. }
4. **class** A6 **implements** printable{
5. **public** **void** print(){System.out.println("Hello");}
7. **public** **static** **void** main(String args[]){
8. A6 obj = **new** A6();
9. obj.print();
10. }
11. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=A6)

Output:

Hello

## Java Interface Example: Drawable

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

*File: TestInterface1.java*

1. //Interface declaration: by first user
2. **interface** Drawable{
3. **void** draw();
4. }
5. //Implementation: by second user
6. **class** Rectangle **implements** Drawable{
7. **public** **void** draw(){System.out.println("drawing rectangle");}
8. }
9. **class** Circle **implements** Drawable{
10. **public** **void** draw(){System.out.println("drawing circle");}
11. }
12. //Using interface: by third user
13. **class** TestInterface1{
14. **public** **static** **void** main(String args[]){
15. Drawable d=**new** Circle();//In real scenario, object is provided by method e.g. getDrawable()
16. d.draw();
17. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestInterface1)

Output:

drawing circle

## Java Interface Example: Bank

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

*File: TestInterface2.java*

1. **interface** Bank{
2. **float** rateOfInterest();
3. }
4. **class** SBI **implements** Bank{
5. **public** **float** rateOfInterest(){**return** 9.15f;}
6. }
7. **class** PNB **implements** Bank{
8. **public** **float** rateOfInterest(){**return** 9.7f;}
9. }
10. **class** TestInterface2{
11. **public** **static** **void** main(String[] args){
12. Bank b=**new** SBI();
13. System.out.println("ROI: "+b.rateOfInterest());
14. }}

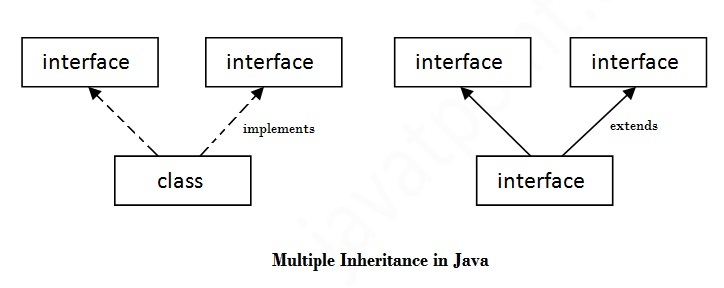
[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestInterface2)

Output:

ROI: 9.15

## Multiple inheritance in Java by interface

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



1. **interface** Printable{
2. **void** print();
3. }
4. **interface** Showable{
5. **void** show();
6. }
7. **class** A7 **implements** Printable,Showable{
8. **public** **void** print(){System.out.println("Hello");}
9. **public** **void** show(){System.out.println("Welcome");}
11. **public** **static** **void** main(String args[]){
12. A7 obj = **new** A7();
13. obj.print();
14. obj.show();
15. }
16. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=A7)

Output:Hello

Welcome

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

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

1. **interface** Printable{
2. **void** print();
3. }
4. **interface** Showable{
5. **void** print();
6. }
8. **class** TestInterface3 **implements** Printable, Showable{
9. **public** **void** print(){System.out.println("Hello");}
10. **public** **static** **void** main(String args[]){
11. TestInterface3 obj = **new** TestInterface3();
12. obj.print();
13. }
14. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestInterface3)

Output:

Hello

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

## Interface inheritance

A class implements interface but one interface extends another interface .

1. **interface** Printable{
2. **void** print();
3. }
4. **interface** Showable **extends** Printable{
5. **void** show();
6. }
7. **class** TestInterface4 **implements** Showable{
8. **public** **void** print(){System.out.println("Hello");}
9. **public** **void** show(){System.out.println("Welcome");}
11. **public** **static** **void** main(String args[]){
12. TestInterface4 obj = **new** TestInterface4();
13. obj.print();
14. obj.show();
15. }
16. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestInterface4)

Output:

Hello

Welcome

## Java 8 Default Method in Interface

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

*File: TestInterfaceDefault.java*

1. **interface** Drawable{
2. **void** draw();
3. **default** **void** msg(){System.out.println("default method");}
4. }
5. **class** Rectangle **implements** Drawable{
6. **public** **void** draw(){System.out.println("drawing rectangle");}
7. }
8. **class** TestInterfaceDefault{
9. **public** **static** **void** main(String args[]){
10. Drawable d=**new** Rectangle();
11. d.draw();
12. d.msg();
13. }}

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

Output:

drawing rectangle

default method

## Java 8 Static Method in Interface

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

*File: TestInterfaceStatic.java*

1. **interface** Drawable{
2. **void** draw();
3. **static** **int** cube(**int** x){**return** x\*x\*x;}
4. }
5. **class** Rectangle **implements** Drawable{
6. **public** **void** draw(){System.out.println("drawing rectangle");}
7. }
9. **class** TestInterfaceStatic{
10. **public** **static** **void** main(String args[]){
11. Drawable d=**new** Rectangle();
12. d.draw();
13. System.out.println(Drawable.cube(3));
14. }}

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

Output:

drawing rectangle

27

## Q) What is marker or tagged interface?

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

1. //How Serializable interface is written?
2. **public** **interface** Serializable{
3. }

#### Nested Interface in Java

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

1. **interface** printable{
2. **void** print();
3. **interface** MessagePrintable{
4. **void** msg();
5. }
6. }

# [**More about Nested Interface**](https://www.javatpoint.com/nested-interface)Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 1) Abstract class can **have abstract and non-abstract**methods. | Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4) Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 5) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 6) An **abstract class**can extend another Java class and implement multiple Java interfaces. | An **interface** can extend another Java interface only. |
| 7) An **abstract class**can be extended using keyword ?extends?. | An **interface class**can be implemented using keyword ?implements?. |
| 8) A Java**abstract class**can have class members like private, protected, etc. | Members of a Java interface are public by default. |
| 9)**Example:** public abstract class Shape{ public abstract void draw(); } | **Example:** public interface Drawable{ void draw(); } |

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

### **Example of abstract class and interface in Java**

Let's see a simple example where we are using interface and abstract class both.

1. //Creating interface that has 4 methods
2. **interface** A{
3. **void** a();//bydefault, public and abstract
4. **void** b();
5. **void** c();
6. **void** d();
7. }
9. //Creating abstract class that provides the implementation of one method of A interface
10. **abstract** **class** B **implements** A{
11. **public** **void** c(){System.out.println("I am C");}
12. }
14. //Creating subclass of abstract class, now we need to provide the implementation of rest of the methods
15. **class** M **extends** B{
16. **public** **void** a(){System.out.println("I am a");}
17. **public** **void** b(){System.out.println("I am b");}
18. **public** **void** d(){System.out.println("I am d");}
19. }
21. //Creating a test class that calls the methods of A interface
22. **class** Test5{
23. **public** **static** **void** main(String args[]){
24. A a=**new** M();
25. a.a();
26. a.b();
27. a.c();
28. a.d();
29. }}

[**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

# 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.

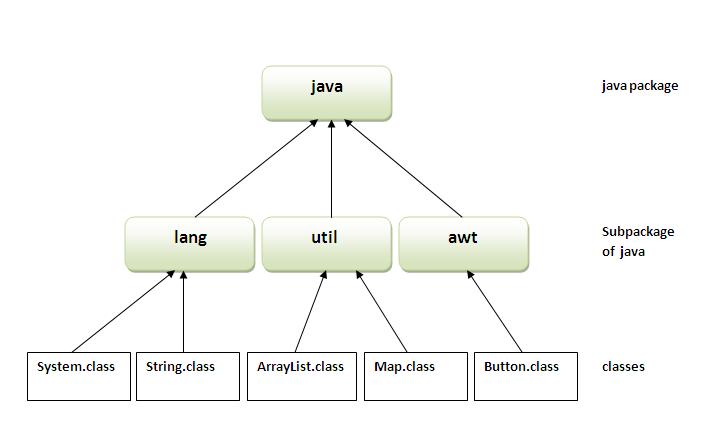
Here, we will have the detailed learning of creating and using user-defined packages.

## 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;
3. fully qualified name.

#### 1) Using packagename.\*

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

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

## Example of package that import the packagename.\*

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. **import** pack.\*;
10. **class** B{
11. **public** **static** **void** main(String args[]){
12. A obj = **new** A();
13. obj.msg();
14. }
15. }

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

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

## Example of package by import fully qualified name

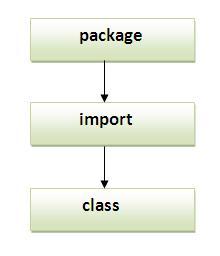
1. //save by A.java
2. **package** pack;
3. **public** **class** A{
4. **public** **void** msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. **package** mypack;
8. **class** B{
9. **public** **static** **void** main(String args[]){
10. pack.A obj = **new** pack.A();//using fully qualified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

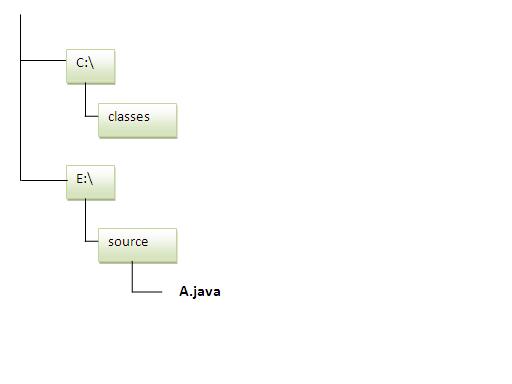
1. **package** com.javatpoint.core;
2. **class** Simple{
3. **public** **static** **void** main(String args[]){
4. System.out.println("Hello subpackage");
5. }
6. }

|  |
| --- |
| **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{}

### **What is static import feature of Java5?**

|  |
| --- |
| Click [Static Import](https://www.javatpoint.com/static-import-in-java) feature of Java5. |

### **What about package class?**

|  |
| --- |
| Click for [Package class](https://www.javatpoint.com/package-class) |

# Access Modifiers in java

1. [private access modifier](https://www.javatpoint.com/access-modifiers#accessprivate)
2. [Role of private constructor](https://www.javatpoint.com/access-modifiers#accessprivatecons)
3. [default access modifier](https://www.javatpoint.com/access-modifiers#accessdefault)
4. [protected access modifier](https://www.javatpoint.com/access-modifiers#accessprotected)
5. [public access modifier](https://www.javatpoint.com/access-modifiers#accesspublic)
6. [Applying access modifier with method overriding](https://www.javatpoint.com/access-modifiers#accessoverriding)

There are two types of modifiers in java: **access modifiers** and **non-access modifiers**.

The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.

There are 4 types of java access modifiers:

1. private
2. default
3. protected
4. public

There are many non-access modifiers such as static, abstract, synchronized, native, volatile, transient etc. Here, we will learn access modifiers.

### **1) private access modifier**

|  |
| --- |
| The private access modifier is accessible only within class. |

### **Simple example of private access modifier**

|  |
| --- |
| In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is compile time error. |

1. **class** A{
2. **private** **int** data=40;
3. **private** **void** msg(){System.out.println("Hello java");}
4. }
6. **public** **class** Simple{
7. **public** **static** **void** main(String args[]){
8. A obj=**new** A();
9. System.out.println(obj.data);//Compile Time Error
10. obj.msg();//Compile Time Error
11. }
12. }

### **Role of Private Constructor**

|  |
| --- |
| If you make any class constructor private, you cannot create the instance of that class from outside the class. For example: |

1. **class** A{
2. **private** A(){}//private constructor
3. **void** msg(){System.out.println("Hello java");}
4. }
5. **public** **class** Simple{
6. **public** **static** **void** main(String args[]){
7. A obj=**new** A();//Compile Time Error
8. }
9. }

#### Note: A class cannot be private or protected except nested class.

### **2) default access modifier**

|  |
| --- |
| If you don't use any modifier, it is treated as **default** bydefault. The default modifier is accessible only within package. |

### **Example of default access modifier**

|  |
| --- |
| In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package. |

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

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

### **3) protected access modifier**

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

### **Example of protected access modifier**

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

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

Output:Hello

### **4) public access modifier**

|  |
| --- |
| The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers. |

### **Example of public access modifier**

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
9. **package** mypack;
10. **import** pack.\*;
12. **class** B{
13. **public** **static** **void** main(String args[]){
14. A obj = **new** A();
15. obj.msg();
16. }
17. }

Output:Hello

### **Understanding all java access modifiers**

Let's understand the access modifiers by a simple table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Access Modifier** | **within class** | **within package** | **outside package by subclass only** | **outside package** |
| **Private** | Y | N | N | N |
| **Default** | Y | Y | N | N |
| **Protected** | Y | Y | Y | N |
| **Public** | Y | Y | Y | Y |

### **Java access modifiers with method overriding**

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

1. **class** A{
2. **protected** **void** msg(){System.out.println("Hello java");}
3. }
5. **public** **class** Simple **extends** A{
6. **void** msg(){System.out.println("Hello java");}//C.T.Error
7. **public** **static** **void** main(String args[]){
8. Simple obj=**new** Simple();
9. obj.msg();
10. }
11. }

|  |
| --- |
| The default modifier is more restrictive than protected. That is why there is compile time error. |

# Encapsulation in Java

**Encapsulation in java** is a process of wrapping code and data together into a single unit, for example capsule i.e. mixed of several medicines.



We can create a fully encapsulated class in java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

The **Java Bean** class is the example of fully encapsulated class.

### **Advantage of Encapsulation in java**

By providing only setter or getter method, you can make the class **read-only or write-only**.

It provides you the **control over the data**. Suppose you want to set the value of id i.e. greater than 100 only, you can write the logic inside the setter method.

### **Simple example of encapsulation in java**

Let's see the simple example of encapsulation that has only one field with its setter and getter methods.

1. //save as Student.java
2. **package** com.javatpoint;
3. **public** **class** Student{
4. **private** String name;
6. **public** String getName(){
7. **return** name;
8. }
9. **public** **void** setName(String name){
10. **this**.name=name
11. }
12. }
13. //save as Test.java
14. **package** com.javatpoint;
15. **class** Test{
16. **public** **static** **void** main(String[] args){
17. Student s=**new** Student();
18. s.setName("vijay");
19. System.out.println(s.getName());
20. }
21. }

Compile By: javac -d . Test.java

Run By: java com.javatpoint.Test

Output: vijay

# Object class in Java

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

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

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

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

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



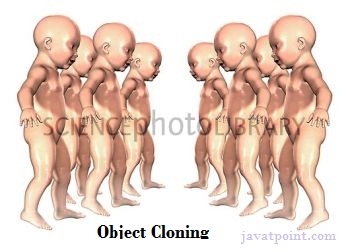
### **Methods of Object class**

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

|  |  |
| --- | --- |
| **Method** | **Description** |
| public final Class getClass() | returns the Class class object of this object. The Class class can further be used to get the metadata of this class. |
| public int hashCode() | returns the hashcode number for this object. |
| public boolean equals(Object obj) | compares the given object to this object. |
| protected Object clone() throws CloneNotSupportedException | creates and returns the exact copy (clone) of this object. |
| public String toString() | returns the string representation of this object. |
| public final void notify() | wakes up single thread, waiting on this object's monitor. |
| public final void notifyAll() | wakes up all the threads, waiting on this object's monitor. |
| public final void wait(long timeout)throws InterruptedException | causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method). |
| public final void wait(long timeout,int nanos)throws InterruptedException | causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method). |
| public final void wait()throws InterruptedException | causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method). |
| protected void finalize()throws Throwable | is invoked by the garbage collector before object is being garbage collected. |

We will have the detailed learning of these methods in next chapters.

# Object Cloning in Java

The **object cloning** is a way to create exact copy of an object. The clone() method of Object class is used to clone an object.

The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates **CloneNotSupportedException**.

The **clone() method** is defined in the Object class. Syntax of the clone() method is as follows:

1. **protected** Object clone() **throws** CloneNotSupportedException

### **Why use clone() method ?**

The **clone() method** saves the extra processing task for creating the exact copy of an object. If we perform it by using the new keyword, it will take a lot of processing time to be performed that is why we use object cloning.

### **Advantage of Object cloning**

Although Object.clone() has some design issues but it is still a popular and easy way of copying objects. Following is a list of advantages of using clone() method:

* You don't need to write lengthy and repetitive codes. Just use an abstract class with a 4- or 5-line long clone() method.
* It is the easiest and most efficient way for copying objects, especially if we are applying it to an already developed or an old project. Just define a parent class, implement Cloneable in it, provide the definition of the clone() method and the task will be done.
* Clone() is the fastest way to copy array.

### **Disadvantage of Object cloning**

Following is a list of some disadvantages of clone() method:

* To use the Object.clone() method, we have to change a lot of syntaxes to our code, like implementing a Cloneable interface, defining the clone() method and handling CloneNotSupportedException, and finally, calling Object.clone() etc.
* We have to implement cloneable interface while it doesn?t have any methods in it. We just have to use it to tell the JVM that we can perform clone() on our object.
* Object.clone() is protected, so we have to provide our own clone() and indirectly call Object.clone() from it.
* Object.clone() doesn?t invoke any constructor so we don?t have any control over object construction.
* If you want to write a clone method in a child class then all of its superclasses should define the clone() method in them or inherit it from another parent class. Otherwise, the super.clone() chain will fail.
* Object.clone() supports only shallow copying but we will need to override it if we need deep cloning.

### **Example of clone() method (Object cloning)**

Let's see the simple example of object cloning

1. **class** Student18 **implements** Cloneable{
2. **int** rollno;
3. String name;
5. Student18(**int** rollno,String name){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. }
10. **public** Object clone()**throws** CloneNotSupportedException{
11. **return** **super**.clone();
12. }
14. **public** **static** **void** main(String args[]){
15. **try**{
16. Student18 s1=**new** Student18(101,"amit");
18. Student18 s2=(Student18)s1.clone();
20. System.out.println(s1.rollno+" "+s1.name);
21. System.out.println(s2.rollno+" "+s2.name);
23. }**catch**(CloneNotSupportedException c){}
25. }
26. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Student18)

Output:101 amit

101 amit

[download the example of object cloning](https://www.javatpoint.com/src/oops/clone.zip)

As you can see in the above example, both reference variables have the same value. Thus, the clone() copies the values of an object to another. So we don't need to write explicit code to copy the value of an object to another.

If we create another object by new keyword and assign the values of another object to this one, it will require a lot of processing on this object. So to save the extra processing task we use clone() method.

# Java Array

Normally, array is a collection of similar type of elements that have contiguous memory location.

**Java array** is an object the contains elements of similar data type. It is a data structure where we store similar elements. We can store only fixed set of elements in a java array.

Array in java is index based, first element of the array is stored at 0 index.



### **Advantage of Java Array**

* **Code Optimization:** It makes the code optimized, we can retrieve or sort the data easily.
* **Random access:** We can get any data located at any index position.

### **Disadvantage of Java Array**

* **Size Limit:** We can store only fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in java.

### **Types of Array in java**

There are two types of array.

* Single Dimensional Array
* Multidimensional Array

### **Single Dimensional Array in java**

### **Syntax to Declare an Array in java**

1. dataType[] arr; (or)
2. dataType []arr; (or)
3. dataType arr[];

### **Instantiation of an Array in java**

1. arrayRefVar=**new** datatype[size];

### **Example of single dimensional java array**

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

1. **class** Testarray{
2. **public** **static** **void** main(String args[]){
4. **int** a[]=**new** **int**[5];//declaration and instantiation
5. a[0]=10;//initialization
6. a[1]=20;
7. a[2]=70;
8. a[3]=40;
9. a[4]=50;
11. //printing array
12. **for**(**int** i=0;i<a.length;i++)//length is the property of array
13. System.out.println(a[i]);
15. }}

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

Output: 10

20

70

40

50

## Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

1. **int** a[]={33,3,4,5};//declaration, instantiation and initialization

Let's see the simple example to print this array.

1. **class** Testarray1{
2. **public** **static** **void** main(String args[]){
4. **int** a[]={33,3,4,5};//declaration, instantiation and initialization
6. //printing array
7. **for**(**int** i=0;i<a.length;i++)//length is the property of array
8. System.out.println(a[i]);
10. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testarray1)

Output:33

3

4

5

### **Passing Array to method in java**

We can pass the java array to method so that we can reuse the same logic on any array.

Let's see the simple example to get minimum number of an array using method.

1. **class** Testarray2{
2. **static** **void** min(**int** arr[]){
3. **int** min=arr[0];
4. **for**(**int** i=1;i<arr.length;i++)
5. **if**(min>arr[i])
6. min=arr[i];
8. System.out.println(min);
9. }
11. **public** **static** **void** main(String args[]){
13. **int** a[]={33,3,4,5};
14. min(a);//passing array to method
16. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testarray2)

Output:3

### **Multidimensional array in java**

In such case, data is stored in row and column based index (also known as matrix form).

### **Syntax to Declare Multidimensional Array in java**

1. dataType[][] arrayRefVar; (or)
2. dataType [][]arrayRefVar; (or)
3. dataType arrayRefVar[][]; (or)
4. dataType []arrayRefVar[];

### **Example to instantiate Multidimensional Array in java**

1. **int**[][] arr=**new** **int**[3][3];//3 row and 3 column

### **Example to initialize Multidimensional Array in java**

1. arr[0][0]=1;
2. arr[0][1]=2;
3. arr[0][2]=3;
4. arr[1][0]=4;
5. arr[1][1]=5;
6. arr[1][2]=6;
7. arr[2][0]=7;
8. arr[2][1]=8;
9. arr[2][2]=9;

### **Example of Multidimensional java array**

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

1. **class** Testarray3{
2. **public** **static** **void** main(String args[]){
4. //declaring and initializing 2D array
5. **int** arr[][]={{1,2,3},{2,4,5},{4,4,5}};
7. //printing 2D array
8. **for**(**int** i=0;i<3;i++){
9. **for**(**int** j=0;j<3;j++){
10. System.out.print(arr[i][j]+" ");
11. }
12. System.out.println();
13. }
15. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testarray3)

Output:1 2 3

2 4 5

4 4 5

### **What is the class name of java array?**

In java, array is an object. For array object, an proxy class is created whose name can be obtained by getClass().getName() method on the object.

1. **class** Testarray4{
2. **public** **static** **void** main(String args[]){
4. **int** arr[]={4,4,5};
6. Class c=arr.getClass();
7. String name=c.getName();
9. System.out.println(name);
11. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testarray4)

Output:I

### **Copying a java array**

We can copy an array to another by the arraycopy method of System class.

### **Syntax of arraycopy method**

1. **public** **static** **void** arraycopy(
2. Object src, **int** srcPos,Object dest, **int** destPos, **int** length
3. )

### **Example of arraycopy method**

1. **class** TestArrayCopyDemo {
2. **public** **static** **void** main(String[] args) {
3. **char**[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
4. 'i', 'n', 'a', 't', 'e', 'd' };
5. **char**[] copyTo = **new** **char**[7];
7. System.arraycopy(copyFrom, 2, copyTo, 0, 7);
8. System.out.println(**new** String(copyTo));
9. }
10. }

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

Output:caffein

### **Addition of 2 matrices in java**

Let's see a simple example that adds two matrices.

1. **class** Testarray5{
2. **public** **static** **void** main(String args[]){
3. //creating two matrices
4. **int** a[][]={{1,3,4},{3,4,5}};
5. **int** b[][]={{1,3,4},{3,4,5}};
7. //creating another matrix to store the sum of two matrices
8. **int** c[][]=**new** **int**[2][3];
10. //adding and printing addition of 2 matrices
11. **for**(**int** i=0;i<2;i++){
12. **for**(**int** j=0;j<3;j++){
13. c[i][j]=a[i][j]+b[i][j];
14. System.out.print(c[i][j]+" ");
15. }
16. System.out.println();//new line
17. }
19. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testarray5)

Output:2 6 8

6 8 10

Wrapper class in Java

**Wrapper class in java** provides the mechanism *to convert primitive into object and object into primitive*.

Since J2SE 5.0, **autoboxing** and **unboxing** feature converts primitive into object and object into primitive automatically. The automatic conversion of primitive into object is known as autoboxing and vice-versa unboxing.

The eight classes of *java.lang* package are known as wrapper classes in java. The list of eight wrapper classes are given below:

|  |  |
| --- | --- |
| **Primitive Type** | **Wrapper class** |
| boolean | Boolean |
| char | Character |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |

Wrapper class Example: Primitive to Wrapper

1. **public** **class** WrapperExample1{
2. **public** **static** **void** main(String args[]){
3. //Converting int into Integer
4. **int** a=20;
5. Integer i=Integer.valueOf(a);//converting int into Integer
6. Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
8. System.out.println(a+" "+i+" "+j);
9. }}

Output:

20 20 20

Wrapper class Example: Wrapper to Primitive

1. **public** **class** WrapperExample2{
2. **public** **static** **void** main(String args[]){
3. //Converting Integer to int
4. Integer a=**new** Integer(3);
5. **int** i=a.intValue();//converting Integer to int
6. **int** j=a;//unboxing, now compiler will write a.intValue() internally
8. System.out.println(a+" "+i+" "+j);
9. }}

Output:

3 3 3

# Call by Value and Call by Reference in Java

|  |
| --- |
| There is only call by value in java, not call by reference. If we call a method passing a value, it is known as call by value. The changes being done in the called method, is not affected in the calling method. |
|  |

### **Example of call by value in java**

|  |
| --- |
| In case of call by value original value is not changed. Let's take a simple example: |

1. **class** Operation{
2. **int** data=50;
4. **void** change(**int** data){
5. data=data+100;//changes will be in the local variable only
6. }
8. **public** **static** **void** main(String args[]){
9. Operation op=**new** Operation();
11. System.out.println("before change "+op.data);
12. op.change(500);
13. System.out.println("after change "+op.data);
15. }
16. }

[download this example](https://www.javatpoint.com/src/oops/callbyvalue1.zip)

Output:before change 50

after change 50

### **Another Example of call by value in java**

In case of call by reference original value is changed if we made changes in the called method. If we pass object in place of any primitive value, original value will be changed. In this example we are passing object as a value. Let's take a simple example:

1. **class** Operation2{
2. **int** data=50;
4. **void** change(Operation2 op){
5. op.data=op.data+100;//changes will be in the instance variable
6. }

9. **public** **static** **void** main(String args[]){
10. Operation2 op=**new** Operation2();
12. System.out.println("before change "+op.data);
13. op.change(op);//passing object
14. System.out.println("after change "+op.data);
16. }
17. }

[download this example](https://www.javatpoint.com/src/oops/callbyvalue2.zip)

Output:before change 50

after change 150

# Java Strictfp Keyword

Java strictfp keyword ensures that you will get the same result on every platform if you perform operations in the floating-point variable. The precision may differ from platform to platform that is why java programming language have provided the strictfp keyword, so that you get same result on every platform. So, now you have better control over the floating-point arithmetic.

### **Legal code for strictfp keyword**

The strictfp keyword can be applied on methods, classes and interfaces.

1. **strictfp** **class** A{}//strictfp applied on class
2. **strictfp** **interface** M{}//strictfp applied on interface
3. **class** A{
4. **strictfp** **void** m(){}//strictfp applied on method
5. }

### **Illegal code for strictfp keyword**

The strictfp keyword **cannot** be applied on abstract methods, variables or constructors.

1. **class** B{
2. **strictfp** **abstract** **void** m();//Illegal combination of modifiers
3. }
4. **class** B{
5. **strictfp** **int** data=10;//modifier strictfp not allowed here
6. }
7. **class** B{
8. **strictfp** B(){}//modifier strictfp not allowed here
9. }

Creating API Document | javadoc tool

We can create document api in java by the help of **javadoc** tool. In the java file, we must use the documentation comment /\*\*... \*/ to post information for the class, method, constructor, fields etc.

Let's see the simple class that contains documentation comment.

1. **package** com.abc;
2. /\*\* This class is a user-defined class that contains one methods cube.\*/
3. **public** **class** M{
5. /\*\* The cube method prints cube of the given number \*/
6. **public** **static** **void**  cube(**int** n){System.out.println(n\*n\*n);}
7. }

To create the document API, you need to use the javadoc tool followed by java file name. There is no need to compile the javafile.

On the command prompt, you need to write:

javadoc M.java

to generate the document api. Now, there will be created a lot of html files. Open the index.html file to get the information about the classes.

# Java Command Line Arguments

1. [Command Line Argument](https://www.javatpoint.com/command-line-argument)
2. [Simple example of command-line argument](https://www.javatpoint.com/command-line-argument#cmdex1)
3. [Example of command-line argument that prints all the values](https://www.javatpoint.com/command-line-argument#cmdex2)

The java command-line argument is an argument i.e. passed at the time of running the java program.

The arguments passed from the console can be received in the java program and it can be used as an input.

So, it provides a convenient way to check the behavior of the program for the different values. You can pass **N** (1,2,3 and so on) numbers of arguments from the command prompt.

### **Simple example of command-line argument in java**

|  |
| --- |
| In this example, we are receiving only one argument and printing it. To run this java program, you must pass at least one argument from the command prompt. |

1. **class** CommandLineExample{
2. **public** **static** **void** main(String args[]){
3. System.out.println("Your first argument is: "+args[0]);
4. }
5. }
6. compile by > javac CommandLineExample.java
7. run by > java CommandLineExample sonoo

Output: Your first argument is: sonoo

### **Example of command-line argument that prints all the values**

|  |
| --- |
| In this example, we are printing all the arguments passed from the command-line. For this purpose, we have traversed the array using for loop. |

1. **class** A{
2. **public** **static** **void** main(String args[]){
4. **for**(**int** i=0;i<args.length;i++)
5. System.out.println(args[i]);
7. }
8. }
9. compile by > javac A.java
10. run by > java A sonoo jaiswal 1 3 abc

Output: sonoo

jaiswal

1

3

abc

# Difference between object and class

There are many differences between object and class. A list of differences between object and class are given below:

|  |  |  |
| --- | --- | --- |
| **No.** | **Object** | **Class** |
| 1) | Object is an **instance** of a class. | Class is a **blueprint or template**from which objects are created. |
| 2) | Object is a **real world entity** such as pen, laptop, mobile, bed, keyboard, mouse, chair etc. | Class is a **group of similar objects**. |
| 3) | Object is a **physical** entity. | Class is a **logical** entity. |
| 4) | Object is created through **new keyword** mainly e.g. Student s1=new Student(); | Class is declared using **class keyword** e.g. class Student{} |
| 5) | Object is created **many times** as per requirement. | Class is declared **once**. |
| 6) | Object **allocates memory when it is created**. | Class **doesn't allocated memory when it is created**. |
| 7) | There are **many ways to create object** in java such as new keyword, newInstance() method, clone() method, factory method and deserialization. | There is only **one way to define class** in java using class keyword. |

|  |  |  |
| --- | --- | --- |
| **No.** | **Method Overloading** | **Method Overriding** |
| 1) | Method overloading is used *to increase the readability* of the program. | Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class. |
| 2) | Method overloading is performed *within class*. | Method overriding occurs *in two classes* that have IS-A (inheritance) relationship. |
| 3) | In case of method overloading, *parameter must be different*. | In case of method overriding, *parameter must be same*. |
| 4) | Method overloading is the example of *compile time polymorphism*. | Method overriding is the example of *run time polymorphism*. |
| 5) | In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter. | *Return type must be same or covariant* in method overriding. |

Let's see some real life example of class and object in java to understand the difference well:

**Class:** Human **Object:** Man, Woman

**Class:** Fruit **Object:** Apple, Banana, Mango, Guava wtc.

**Class:** Mobile phone **Object:** iPhone, Samsung, Moto

**Class:** Food **Object:** Pizza, Burger, Samosa

Difference between method overloading and method overriding in java

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

Java Method Overloading example

1. **class** OverloadingExample{
2. **static** **int** add(**int** a,**int** b){**return** a+b;}
3. **static** **int** add(**int** a,**int** b,**int** c){**return** a+b+c;}
4. }

Java Method Overriding example

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** eat(){System.out.println("eating bread...");}
6. }

String:

# Java String

In java, string is basically an object that represents sequence of char values. An array of characters works same as java string. For example:

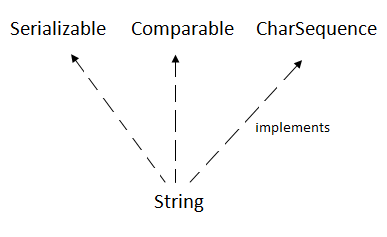
1. **char**[] ch={'j','a','v','a','t','p','o','i','n','t'};
2. String s=**new** String(ch);

is same as:

1. String s="javatpoint";

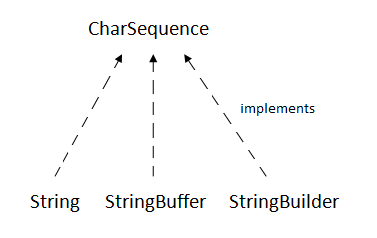
**Java String** class provides a lot of methods to perform operations on string such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* interfaces.



## CharSequence Interface

The CharSequence interface is used to represent sequence of characters. It is implemented by String, StringBuffer and StringBuilder classes. It means, we can create string in java by using these 3 classes.



The java String is immutable i.e. it cannot be changed. Whenever we change any string, a new instance is created. For mutable string, you can use StringBuffer and StringBuilder classes.

We will discuss about immutable string later. Let's first understand what is string in java and how to create the string object.

### **What is String in java**

Generally, string is a sequence of characters. But in java, string is an object that represents a sequence of characters. The java.lang.String class is used to create string object.

### **How to create String object?**

|  |
| --- |
| There are two ways to create String object:   1. By string literal 2. By new keyword |

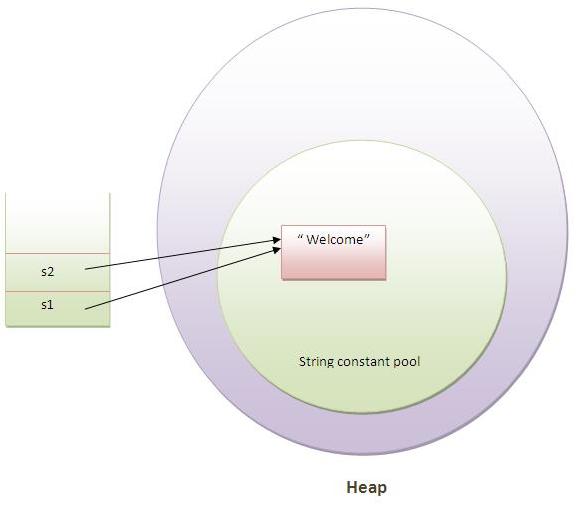
### **1) String Literal**

Java String literal is created by using double quotes. For Example:

1. String s="welcome";

Each time you create a string literal, the JVM checks the string constant pool first. If the string already exists in the pool, a reference to the pooled instance is returned. If string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

1. String s1="Welcome";
2. String s2="Welcome";//will not create new instance



In the above example only one object will be created. Firstly JVM will not find any string object with the value "Welcome" in string constant pool, so it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create new object but will return the reference to the same instance.

#### Note: String objects are stored in a special memory area known as string constant pool.

### **Why java uses concept of string literal?**

To make Java more memory efficient (because no new objects are created if it exists already in string constant pool).

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | [char charAt(int index)](https://www.javatpoint.com/java-string-charat) | returns char value for the particular index |
| 2 | [int length()](https://www.javatpoint.com/java-string-length) | returns string length |
| 3 | [static String format(String format, Object... args)](https://www.javatpoint.com/java-string-format) | returns formatted string |
| 4 | [static String format(Locale l, String format, Object... args)](https://www.javatpoint.com/java-string-format) | returns formatted string with given locale |
| 5 | [String substring(int beginIndex)](https://www.javatpoint.com/java-string-substring) | returns substring for given begin index |
| 6 | [String substring(int beginIndex, int endIndex)](https://www.javatpoint.com/java-string-substring) | returns substring for given begin index and end index |
| 7 | [boolean contains(CharSequence s)](https://www.javatpoint.com/java-string-contains) | returns true or false after matching the sequence of char value |
| 8 | [static String join(CharSequence delimiter, CharSequence... elements)](https://www.javatpoint.com/java-string-join) | returns a joined string |
| 9 | [static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)](https://www.javatpoint.com/java-string-join) | returns a joined string |
| 10 | [boolean equals(Object another)](https://www.javatpoint.com/java-string-equals) | checks the equality of string with object |
| 11 | [boolean isEmpty()](https://www.javatpoint.com/java-string-isempty) | checks if string is empty |
| 12 | [String concat(String str)](https://www.javatpoint.com/java-string-concat) | concatinates specified string |
| 13 | [String replace(char old, char new)](https://www.javatpoint.com/java-string-replace) | replaces all occurrences of specified char value |
| 14 | [String replace(CharSequence old, CharSequence new)](https://www.javatpoint.com/java-string-replace) | replaces all occurrences of specified CharSequence |
| 15 | [static String equalsIgnoreCase(String another)](https://www.javatpoint.com/java-string-equalsignorecase) | compares another string. It doesn't check case. |
| 16 | [String[] split(String regex)](https://www.javatpoint.com/java-string-split) | returns splitted string matching regex |
| 17 | [String[] split(String regex, int limit)](https://www.javatpoint.com/java-string-split) | returns splitted string matching regex and limit |
| 18 | [String intern()](https://www.javatpoint.com/java-string-intern) | returns interned string |
| 19 | [int indexOf(int ch)](https://www.javatpoint.com/java-string-indexof) | returns specified char value index |
| 20 | [int indexOf(int ch, int fromIndex)](https://www.javatpoint.com/java-string-indexof) | returns specified char value index starting with given index |
| 21 | [int indexOf(String substring)](https://www.javatpoint.com/java-string-indexof) | returns specified substring index |
| 22 | [int indexOf(String substring, int fromIndex)](https://www.javatpoint.com/java-string-indexof) | returns specified substring index starting with given index |
| 23 | [String toLowerCase()](https://www.javatpoint.com/java-string-tolowercase) | returns string in lowercase. |
| 24 | [String toLowerCase(Locale l)](https://www.javatpoint.com/java-string-tolowercase) | returns string in lowercase using specified locale. |
| 25 | [String toUpperCase()](https://www.javatpoint.com/java-string-touppercase) | returns string in uppercase. |
| 26 | [String toUpperCase(Locale l)](https://www.javatpoint.com/java-string-touppercase) | returns string in uppercase using specified locale. |
| 27 | [String trim()](https://www.javatpoint.com/java-string-trim) | removes beginning and ending spaces of this string. |
| 28 | [static String valueOf(int value)](https://www.javatpoint.com/java-string-valueof) | converts given type into string. It is overloaded. |

### **2) By new keyword**

1. String s=**new** String("Welcome");//creates two objects and one reference variable

In such case, JVM will create a new string object in normal(non pool) heap memory and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in heap(non pool).

### **Java String Example**

1. **public** **class** StringExample{
2. **public** **static** **void** main(String args[]){
3. String s1="java";//creating string by java string literal
4. **char** ch[]={'s','t','r','i','n','g','s'};
5. String s2=**new** String(ch);//converting char array to string
6. String s3=**new** String("example");//creating java string by new keyword
7. System.out.println(s1);
8. System.out.println(s2);
9. System.out.println(s3);
10. }}

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

java

strings

example

### **Java String class methods**

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

Do You Know ?

* Why String objects are immutable?
* How to create an immutable class?
* What is string constant pool?
* What code is written by the compiler if you concat any string by + (string concatenation operator)?
* What is the difference between StringBuffer and StringBuilder class?

What we will learn in String Handling ?

* Concept of String
* Immutable String
* String Comparison
* String Concatenation
* Concept of Substring
* String class methods and its usage
* StringBuffer class
* StringBuilder class
* Creating Immutable class
* toString() method
* StringTokenizer class

# Immutable String in Java

In java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable.

Once string object is created its data or state can't be changed but a new string object is created.

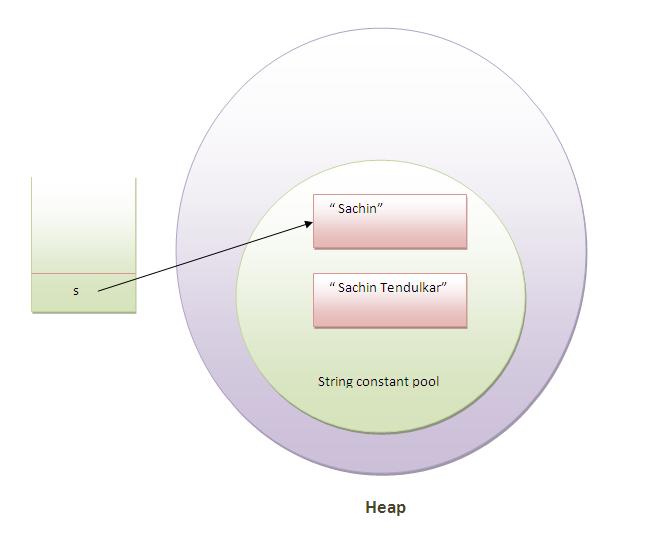
Let's try to understand the immutability concept by the example given below:

1. **class** Testimmutablestring{
2. **public** **static** **void** main(String args[]){
3. String s="Sachin";
4. s.concat(" Tendulkar");//concat() method appends the string at the end
5. System.out.println(s);//will print Sachin because strings are immutable objects
6. }
7. }

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

Output:Sachin

Now it can be understood by the diagram given below. Here Sachin is not changed but a new object is created with sachintendulkar. That is why string is known as immutable.



As you can see in the above figure that two objects are created but s reference variable still refers to "Sachin" not to "Sachin Tendulkar".

But if we explicitely assign it to the reference variable, it will refer to "Sachin Tendulkar" object.For example:

1. **class** Testimmutablestring1{
2. **public** **static** **void** main(String args[]){
3. String s="Sachin";
4. s=s.concat(" Tendulkar");
5. System.out.println(s);
6. }
7. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testimmutablestring1)

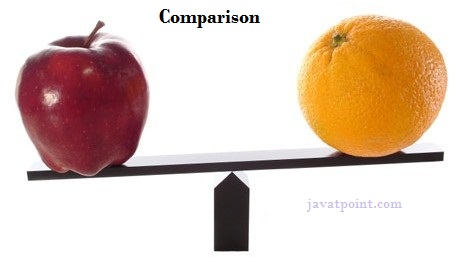
Output:Sachin Tendulkar

In such case, s points to the "Sachin Tendulkar". Please notice that still sachin object is not modified.

### **Why string objects are immutable in java?**

|  |
| --- |
| Because java uses the concept of string literal.Suppose there are 5 reference variables,all referes to one object "sachin".If one reference variable changes the value of the object, it will be affected to all the reference variables. That is why string objects are immutable in java. |

Java String compare



We can compare string in java on the basis of content and reference.

It is used in **authentication** (by equals() method), **sorting** (by compareTo() method), **reference matching** (by == operator) etc.

There are three ways to compare string in java:

1. By equals() method
2. By = = operator
3. By compareTo() method

1) String compare by equals() method

The String equals() method compares the original content of the string. It compares values of string for equality. String class provides two methods:

* **public boolean equals(Object another)** compares this string to the specified object.
* **public boolean equalsIgnoreCase(String another)** compares this String to another string, ignoring case.

1. **class** Teststringcomparison1{
2. **public** **static** **void** main(String args[]){
3. String s1="Sachin";
4. String s2="Sachin";
5. String s3=**new** String("Sachin");
6. String s4="Saurav";
7. System.out.println(s1.equals(s2));//true
8. System.out.println(s1.equals(s3));//true
9. System.out.println(s1.equals(s4));//false
10. }
11. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Teststringcomparison1)

Output:true

true

false

1. **class** Teststringcomparison2{
2. **public** **static** **void** main(String args[]){
3. String s1="Sachin";
4. String s2="SACHIN";
6. System.out.println(s1.equals(s2));//false
7. System.out.println(s1.equalsIgnoreCase(s2));//true
8. }
9. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Teststringcomparison2)

Output:

false

true

[Click here for more about equals() method](https://www.javatpoint.com/java-string-equals)

2) String compare by == operator

The = = operator compares references not values.

1. **class** Teststringcomparison3{
2. **public** **static** **void** main(String args[]){
3. String s1="Sachin";
4. String s2="Sachin";
5. String s3=**new** String("Sachin");
6. System.out.println(s1==s2);//true (because both refer to same instance)
7. System.out.println(s1==s3);//false(because s3 refers to instance created in nonpool)
8. }
9. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Teststringcomparison3)

Output:true

false

3) String compare by compareTo() method

The String compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

Suppose s1 and s2 are two string variables. If:

* **s1 == s2** :0
* **s1 > s2**  :positive value
* **s1 < s2**  :negative value

1. **class** Teststringcomparison4{
2. **public** **static** **void** main(String args[]){
3. String s1="Sachin";
4. String s2="Sachin";
5. String s3="Ratan";
6. System.out.println(s1.compareTo(s2));//0
7. System.out.println(s1.compareTo(s3));//1(because s1>s3)
8. System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )
9. }
10. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Teststringcomparison4)

Output:0

1

-1

# [Click me for more about compareTo() method](https://www.javatpoint.com/java-string-compareto)

# String Concatenation in Java

In java, string concatenation forms a new string that is the combination of multiple strings. There are two ways to concat string in java:

1. By + (string concatenation) operator
2. By concat() method

## 1) String Concatenation by + (string concatenation) operator

Java string concatenation operator (+) is used to add strings. For Example:

1. **class** TestStringConcatenation1{
2. **public** **static** **void** main(String args[]){
3. String s="Sachin"+" Tendulkar";
4. System.out.println(s);//Sachin Tendulkar
5. }
6. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestStringConcatenation1)

Output:Sachin Tendulkar

The **Java compiler transforms** above code to this:

1. String s=(**new** StringBuilder()).append("Sachin").append(" Tendulkar).toString();

In java, String concatenation is implemented through the StringBuilder (or StringBuffer) class and its append method. String concatenation operator produces a new string by appending the second operand onto the end of the first operand. The string concatenation operator can concat not only string but primitive values also. For Example:

1. **class** TestStringConcatenation2{
2. **public** **static** **void** main(String args[]){
3. String s=50+30+"Sachin"+40+40;
4. System.out.println(s);//80Sachin4040
5. }
6. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestStringConcatenation2)

80Sachin4040

#### Note: After a string literal, all the + will be treated as string concatenation operator.

### **2) String Concatenation by concat() method**

The String concat() method concatenates the specified string to the end of current string. Syntax:

1. **public** String concat(String another)

Let's see the example of String concat() method.

1. **class** TestStringConcatenation3{
2. **public** **static** **void** main(String args[]){
3. String s1="Sachin ";
4. String s2="Tendulkar";
5. String s3=s1.concat(s2);
6. System.out.println(s3);//Sachin Tendulkar
7. }
8. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestStringConcatenation3)

Sachin Tendulkar

Substring in Java

A part of string is called **substring**. In other words, substring is a subset of another string. In case of substring startIndex is inclusive and endIndex is exclusive.

**Note: Index starts from 0.**

You can get substring from the given string object by one of the two methods:

1. **public String substring(int startIndex):** This method returns new String object containing the substring of the given string from specified startIndex (inclusive).
2. **public String substring(int startIndex, int endIndex):**This method returns new String object containing the substring of the given string from specified startIndex to endIndex.

In case of string:

* **startIndex:** inclusive
* **endIndex:** exclusive

Let's understand the startIndex and endIndex by the code given below.

1. String s="hello";
2. System.out.println(s.substring(0,2));//he

In the above substring, 0 points to h but 2 points to e (because end index is exclusive).

Example of java substring

1. **public** **class** TestSubstring{
2. **public** **static** **void** main(String args[]){
3. String s="SachinTendulkar";
4. System.out.println(s.substring(6));//Tendulkar
5. System.out.println(s.substring(0,6));//Sachin
6. }
7. }

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

Tendulkar

Sachin

# Java String class methods

The java.lang.String class provides a lot of methods to work on string. By the help of these methods, we can perform operations on string such as trimming, concatenating, converting, comparing, replacing strings etc.

Java String is a powerful concept because everything is treated as a string if you submit any form in window based, web based or mobile application.

Let's see the important methods of String class.

### **Java String toUpperCase() and toLowerCase() method**

The java string toUpperCase() method converts this string into uppercase letter and string toLowerCase() method into lowercase letter.

1. String s="Sachin";
2. System.out.println(s.toUpperCase());//SACHIN
3. System.out.println(s.toLowerCase());//sachin
4. System.out.println(s);//Sachin(no change in original)

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

SACHIN

sachin

Sachin

### **Java String trim() method**

The string trim() method eliminates white spaces before and after string.

1. String s="  Sachin  ";
2. System.out.println(s);//  Sachin
3. System.out.println(s.trim());//Sachin

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testmethodofstringclass1)

Sachin

Sachin

### **Java String startsWith() and endsWith() method**

1. String s="Sachin";
2. System.out.println(s.startsWith("Sa"));//true
3. System.out.println(s.endsWith("n"));//true

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testmethodofstringclass2)

true

true

### **Java String charAt() method**

The string charAt() method returns a character at specified index.

1. String s="Sachin";
2. System.out.println(s.charAt(0));//S
3. System.out.println(s.charAt(3));//h

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testmethodofstringclass3)

S

h

### **Java String length() method**

The string length() method returns length of the string.

1. String s="Sachin";
2. System.out.println(s.length());//6

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testmethodofstringclass4)

6

### **Java String intern() method**

A pool of strings, initially empty, is maintained privately by the class String.

When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.

1. String s=**new** String("Sachin");
2. String s2=s.intern();
3. System.out.println(s2);//Sachin

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testmethodofstringclass5)

Sachin

### **Java String valueOf() method**

The string valueOf() method coverts given type such as int, long, float, double, boolean, char and char array into string.

1. **int** a=10;
2. String s=String.valueOf(a);
3. System.out.println(s+10);

Output:

1010

### **Java String replace() method**

The string replace() method replaces all occurrence of first sequence of character with second sequence of character.

1. String s1="Java is a programming language. Java is a platform. Java is an Island.";
2. String replaceString=s1.replace("Java","Kava");//replaces all occurrences of "Java" to "Kava"
3. System.out.println(replaceString);

Output:

Kava is a programming language. Kava is a platform. Kava is an Island.

# Java StringBuffer class

Java StringBuffer class is used to create mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

#### Note: Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.

### **Important Constructors of StringBuffer class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| StringBuffer() | creates an empty string buffer with the initial capacity of 16. |
| StringBuffer(String str) | creates a string buffer with the specified string. |
| StringBuffer(int capacity) | creates an empty string buffer with the specified capacity as length. |

### **Important methods of StringBuffer class**

|  |  |  |
| --- | --- | --- |
| **Modifier and Type** | **Method** | **Description** |
| public synchronized StringBuffer | append(String s) | is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc. |
| public synchronized StringBuffer | insert(int offset, String s) | is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc. |
| public synchronized StringBuffer | replace(int startIndex, int endIndex, String str) | is used to replace the string from specified startIndex and endIndex. |
| public synchronized StringBuffer | delete(int startIndex, int endIndex) | is used to delete the string from specified startIndex and endIndex. |
| public synchronized StringBuffer | reverse() | is used to reverse the string. |
| public int | capacity() | is used to return the current capacity. |
| public void | ensureCapacity(int minimumCapacity) | is used to ensure the capacity at least equal to the given minimum. |
| public char | charAt(int index) | is used to return the character at the specified position. |
| public int | length() | is used to return the length of the string i.e. total number of characters. |
| public String | substring(int beginIndex) | is used to return the substring from the specified beginIndex. |
| public String | substring(int beginIndex, int endIndex) | is used to return the substring from the specified beginIndex and endIndex. |

### **What is mutable string**

A string that can be modified or changed is known as mutable string. StringBuffer and StringBuilder classes are used for creating mutable string.

### **1) StringBuffer append() method**

The append() method concatenates the given argument with this string.

1. **class** StringBufferExample{
2. **public** **static** **void** main(String args[]){
3. StringBuffer sb=**new** StringBuffer("Hello ");
4. sb.append("Java");//now original string is changed
5. System.out.println(sb);//prints Hello Java
6. }
7. }

### **2) StringBuffer insert() method**

The insert() method inserts the given string with this string at the given position.

1. **class** StringBufferExample2{
2. **public** **static** **void** main(String args[]){
3. StringBuffer sb=**new** StringBuffer("Hello ");
4. sb.insert(1,"Java");//now original string is changed
5. System.out.println(sb);//prints HJavaello
6. }
7. }

### **3) StringBuffer replace() method**

The replace() method replaces the given string from the specified beginIndex and endIndex.

1. **class** StringBufferExample3{
2. **public** **static** **void** main(String args[]){
3. StringBuffer sb=**new** StringBuffer("Hello");
4. sb.replace(1,3,"Java");
5. System.out.println(sb);//prints HJavalo
6. }
7. }

### **4) StringBuffer delete() method**

The delete() method of StringBuffer class deletes the string from the specified beginIndex to endIndex.

1. **class** StringBufferExample4{
2. **public** **static** **void** main(String args[]){
3. StringBuffer sb=**new** StringBuffer("Hello");
4. sb.delete(1,3);
5. System.out.println(sb);//prints Hlo
6. }
7. }

### **5) StringBuffer reverse() method**

The reverse() method of StringBuilder class reverses the current string.

1. **class** StringBufferExample5{
2. **public** **static** **void** main(String args[]){
3. StringBuffer sb=**new** StringBuffer("Hello");
4. sb.reverse();
5. System.out.println(sb);//prints olleH
6. }
7. }

### **6) StringBuffer capacity() method**

The capacity() method of StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity\*2)+2. For example if your current capacity is 16, it will be (16\*2)+2=34.

1. **class** StringBufferExample6{
2. **public** **static** **void** main(String args[]){
3. StringBuffer sb=**new** StringBuffer();
4. System.out.println(sb.capacity());//default 16
5. sb.append("Hello");
6. System.out.println(sb.capacity());//now 16
7. sb.append("java is my favourite language");
8. System.out.println(sb.capacity());//now (16\*2)+2=34 i.e (oldcapacity\*2)+2
9. }
10. }

### **7) StringBuffer ensureCapacity() method**

The ensureCapacity() method of StringBuffer class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity\*2)+2. For example if your current capacity is 16, it will be (16\*2)+2=34.

1. **class** StringBufferExample7{
2. **public** **static** **void** main(String args[]){
3. StringBuffer sb=**new** StringBuffer();
4. System.out.println(sb.capacity());//default 16
5. sb.append("Hello");
6. System.out.println(sb.capacity());//now 16
7. sb.append("java is my favourite language");
8. System.out.println(sb.capacity());//now (16\*2)+2=34 i.e (oldcapacity\*2)+2
9. sb.ensureCapacity(10);//now no change
10. System.out.println(sb.capacity());//now 34
11. sb.ensureCapacity(50);//now (34\*2)+2
12. System.out.println(sb.capacity());//now 70
13. }
14. }

# Java StringBuilder class

Java StringBuilder class is used to create mutable (modifiable) string. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK 1.5.

## Important Constructors of StringBuilder class

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| StringBuilder() | creates an empty string Builder with the initial capacity of 16. |
| StringBuilder(String str) | creates a string Builder with the specified string. |
| StringBuilder(int length) | creates an empty string Builder with the specified capacity as length. |

## Important methods of StringBuilder class

|  |  |
| --- | --- |
| **Method** | **Description** |
| public StringBuilder append(String s) | is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc. |
| public StringBuilder insert(int offset, String s) | is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc. |
| public StringBuilder replace(int startIndex, int endIndex, String str) | is used to replace the string from specified startIndex and endIndex. |
| public StringBuilder delete(int startIndex, int endIndex) | is used to delete the string from specified startIndex and endIndex. |
| public StringBuilder reverse() | is used to reverse the string. |
| public int capacity() | is used to return the current capacity. |
| public void ensureCapacity(int minimumCapacity) | is used to ensure the capacity at least equal to the given minimum. |
| public char charAt(int index) | is used to return the character at the specified position. |
| public int length() | is used to return the length of the string i.e. total number of characters. |
| public String substring(int beginIndex) | is used to return the substring from the specified beginIndex. |
| public String substring(int beginIndex, int endIndex) | is used to return the substring from the specified beginIndex and endIndex. |

## Java StringBuilder Examples

Let's see the examples of different methods of StringBuilder class.

### **1) StringBuilder append() method**

The StringBuilder append() method concatenates the given argument with this string.

1. **class** StringBuilderExample{
2. **public** **static** **void** main(String args[]){
3. StringBuilder sb=**new** StringBuilder("Hello ");
4. sb.append("Java");//now original string is changed
5. System.out.println(sb);//prints Hello Java
6. }
7. }

### **2) StringBuilder insert() method**

The StringBuilder insert() method inserts the given string with this string at the given position.

1. **class** StringBuilderExample2{
2. **public** **static** **void** main(String args[]){
3. StringBuilder sb=**new** StringBuilder("Hello ");
4. sb.insert(1,"Java");//now original string is changed
5. System.out.println(sb);//prints HJavaello
6. }
7. }

### **3) StringBuilder replace() method**

The StringBuilder replace() method replaces the given string from the specified beginIndex and endIndex.

1. **class** StringBuilderExample3{
2. **public** **static** **void** main(String args[]){
3. StringBuilder sb=**new** StringBuilder("Hello");
4. sb.replace(1,3,"Java");
5. System.out.println(sb);//prints HJavalo
6. }
7. }

### **4) StringBuilder delete() method**

The delete() method of StringBuilder class deletes the string from the specified beginIndex to endIndex.

1. **class** StringBuilderExample4{
2. **public** **static** **void** main(String args[]){
3. StringBuilder sb=**new** StringBuilder("Hello");
4. sb.delete(1,3);
5. System.out.println(sb);//prints Hlo
6. }
7. }

### **5) StringBuilder reverse() method**

The reverse() method of StringBuilder class reverses the current string.

1. **class** StringBuilderExample5{
2. **public** **static** **void** main(String args[]){
3. StringBuilder sb=**new** StringBuilder("Hello");
4. sb.reverse();
5. System.out.println(sb);//prints olleH
6. }
7. }

### **6) StringBuilder capacity() method**

The capacity() method of StringBuilder class returns the current capacity of the Builder. The default capacity of the Builder is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity\*2)+2. For example if your current capacity is 16, it will be (16\*2)+2=34.

1. **class** StringBuilderExample6{
2. **public** **static** **void** main(String args[]){
3. StringBuilder sb=**new** StringBuilder();
4. System.out.println(sb.capacity());//default 16
5. sb.append("Hello");
6. System.out.println(sb.capacity());//now 16
7. sb.append("java is my favourite language");
8. System.out.println(sb.capacity());//now (16\*2)+2=34 i.e (oldcapacity\*2)+2
9. }
10. }

### **7) StringBuilder ensureCapacity() method**

The ensureCapacity() method of StringBuilder class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity\*2)+2. For example if your current capacity is 16, it will be (16\*2)+2=34.

1. **class** StringBuilderExample7{
2. **public** **static** **void** main(String args[]){
3. StringBuilder sb=**new** StringBuilder();
4. System.out.println(sb.capacity());//default 16
5. sb.append("Hello");
6. System.out.println(sb.capacity());//now 16
7. sb.append("java is my favourite language");
8. System.out.println(sb.capacity());//now (16\*2)+2=34 i.e (oldcapacity\*2)+2
9. sb.ensureCapacity(10);//now no change
10. System.out.println(sb.capacity());//now 34
11. sb.ensureCapacity(50);//now (34\*2)+2
12. System.out.println(sb.capacity());//now 70
13. }
14. }

Difference between String and StringBuffer

There are many differences between String and StringBuffer. A list of differences between String and StringBuffer are given below:

|  |  |  |
| --- | --- | --- |
| **No.** | **String** | **StringBuffer** |
| 1) | String class is immutable. | StringBuffer class is mutable. |
| 2) | String is slow and consumes more memory when you concat too many strings because every time it creates new instance. | StringBuffer is fast and consumes less memory when you cancat strings. |
| 3) | String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method. | StringBuffer class doesn't override the equals() method of Object class. |

Performance Test of String and StringBuffer

1. **public** **class** ConcatTest{
2. **public** **static** String concatWithString()    {
3. String t = "Java";
4. **for** (**int** i=0; i<10000; i++){
5. t = t + "Tpoint";
6. }
7. **return** t;
8. }
9. **public** **static** String concatWithStringBuffer(){
10. StringBuffer sb = **new** StringBuffer("Java");
11. **for** (**int** i=0; i<10000; i++){
12. sb.append("Tpoint");
13. }
14. **return** sb.toString();
15. }
16. **public** **static** **void** main(String[] args){
17. **long** startTime = System.currentTimeMillis();
18. concatWithString();
19. System.out.println("Time taken by Concating with String: "+(System.currentTimeMillis()-startTime)+"ms");
20. startTime = System.currentTimeMillis();
21. concatWithStringBuffer();
22. System.out.println("Time taken by Concating with  StringBuffer: "+(System.currentTimeMillis()-startTime)+"ms");
23. }
24. }

Time taken by Concating with String: 578ms

Time taken by Concating with StringBuffer: 0ms

String and StringBuffer HashCode Test

As you can see in the program given below, String returns new hashcode value when you concat string but StringBuffer returns same.

1. **public** **class** InstanceTest{
2. **public** **static** **void** main(String args[]){
3. System.out.println("Hashcode test of String:");
4. String str="java";
5. System.out.println(str.hashCode());
6. str=str+"tpoint";
7. System.out.println(str.hashCode());
9. System.out.println("Hashcode test of StringBuffer:");
10. StringBuffer sb=**new** StringBuffer("java");
11. System.out.println(sb.hashCode());
12. sb.append("tpoint");
13. System.out.println(sb.hashCode());
14. }
15. }

Hashcode test of String:

3254818

229541438

Hashcode test of StringBuffer:

118352462

118352462

Difference between StringBuffer and StringBuilder

There are many differences between StringBuffer and StringBuilder. A list of differences between StringBuffer and StringBuilder are given below:

|  |  |  |
| --- | --- | --- |
| **No.** | **StringBuffer** | **StringBuilder** |
| 1) | StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously. | StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously. |
| 2) | StringBuffer is *less efficient* than StringBuilder. | StringBuilder is *more efficient* than StringBuffer. |

StringBuffer Example

1. **public** **class** BufferTest{
2. **public** **static** **void** main(String[] args){
3. StringBuffer buffer=**new** StringBuffer("hello");
4. buffer.append("java");
5. System.out.println(buffer);
6. }
7. }

hellojava

StringBuilder Example

1. **public** **class** BuilderTest{
2. **public** **static** **void** main(String[] args){
3. StringBuilder builder=**new** StringBuilder("hello");
4. builder.append("java");
5. System.out.println(builder);
6. }
7. }

hellojava

Performance Test of StringBuffer and StringBuilder

Let's see the code to check the performance of StringBuffer and StringBuilder classes.

1. **public** **class** ConcatTest{
2. **public** **static** **void** main(String[] args){
3. **long** startTime = System.currentTimeMillis();
4. StringBuffer sb = **new** StringBuffer("Java");
5. **for** (**int** i=0; i<10000; i++){
6. sb.append("Tpoint");
7. }
8. System.out.println("Time taken by StringBuffer: " + (System.currentTimeMillis() - startTime) + "ms");
9. startTime = System.currentTimeMillis();
10. StringBuilder sb2 = **new** StringBuilder("Java");
11. **for** (**int** i=0; i<10000; i++){
12. sb2.append("Tpoint");
13. }
14. System.out.println("Time taken by StringBuilder: " + (System.currentTimeMillis() - startTime) + "ms");
15. }
16. }

Time taken by StringBuffer: 16ms

Time taken by StringBuilder: 0ms

# How to create Immutable class?

There are many immutable classes like String, Boolean, Byte, Short, Integer, Long, Float, Double etc. In short, all the wrapper classes and String class is immutable. We can also create immutable class by creating final class that have final data members as the example given below:

### **Example to create Immutable class**

|  |
| --- |
| In this example, we have created a final class named Employee. It have one final datamember, a parameterized constructor and getter method. |

1. **public** **final** **class** Employee{
2. **final** String pancardNumber;
4. **public** Employee(String pancardNumber){
5. **this**.pancardNumber=pancardNumber;
6. }
8. **public** String getPancardNumber(){
9. **return** pancardNumber;
10. }
12. }

The above class is immutable because:

* The instance variable of the class is final i.e. we cannot change the value of it after creating an object.
* The class is final so we cannot create the subclass.
* There is no setter methods i.e. we have no option to change the value of the instance variable.

These points makes this class as immutable.

# Java toString() method

If you want to represent any object as a string, **toString() method** comes into existence.

The toString() method returns the string representation of the object.

If you print any object, java compiler internally invokes the toString() method on the object. So overriding the toString() method, returns the desired output, it can be the state of an object etc. depends on your implementation.

## Advantage of Java toString() method

By overriding the toString() method of the Object class, we can return values of the object, so we don't need to write much code.

### **Understanding problem without toString() method**

Let's see the simple code that prints reference.

1. **class** Student{
2. **int** rollno;
3. String name;
4. String city;
6. Student(**int** rollno, String name, String city){
7. **this**.rollno=rollno;
8. **this**.name=name;
9. **this**.city=city;
10. }
12. **public** **static** **void** main(String args[]){
13. Student s1=**new** Student(101,"Raj","lucknow");
14. Student s2=**new** Student(102,"Vijay","ghaziabad");
16. System.out.println(s1);//compiler writes here s1.toString()
17. System.out.println(s2);//compiler writes here s2.toString()
18. }
19. }

Output:Student@1fee6fc

Student@1eed786

|  |  |
| --- | --- |
| As you can see in the above example, printing s1 and s2 prints the hashcode values of the objects but I want to print the values of these objects. Since java compiler internally calls toString() method, overriding this method will return the specified values. Let's understand it with the example given below: |  |

## Example of Java toString() method

Now let's see the real example of toString() method.

1. **class** Student{
2. **int** rollno;
3. String name;
4. String city;
6. Student(**int** rollno, String name, String city){
7. **this**.rollno=rollno;
8. **this**.name=name;
9. **this**.city=city;
10. }
12. **public** String toString(){//overriding the toString() method
13. **return** rollno+" "+name+" "+city;
14. }
15. **public** **static** **void** main(String args[]){
16. Student s1=**new** Student(101,"Raj","lucknow");
17. Student s2=**new** Student(102,"Vijay","ghaziabad");
19. System.out.println(s1);//compiler writes here s1.toString()
20. System.out.println(s2);//compiler writes here s2.toString()
21. }
22. }

[download this example of toString method](https://www.javatpoint.com/src/string/tostring.zip)

Output:101 Raj lucknow

102 Vijay ghaziabad

# StringTokenizer in Java

1. [StringTokenizer](https://www.javatpoint.com/string-tokenizer-in-java)
2. [Methods of StringTokenizer](https://www.javatpoint.com/string-tokenizer-in-java)
3. [Example of StringTokenizer](https://www.javatpoint.com/string-tokenizer-in-java)

The **java.util.StringTokenizer** class allows you to break a string into tokens. It is simple way to break string.

It doesn't provide the facility to differentiate numbers, quoted strings, identifiers etc. like StreamTokenizer class. We will discuss about the StreamTokenizer class in I/O chapter.

#### Constructors of StringTokenizer class

There are 3 constructors defined in the StringTokenizer class.

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| StringTokenizer(String str) | creates StringTokenizer with specified string. |
| StringTokenizer(String str, String delim) | creates StringTokenizer with specified string and delimeter. |
| StringTokenizer(String str, String delim, boolean returnValue) | creates StringTokenizer with specified string, delimeter and returnValue. If return value is true, delimiter characters are considered to be tokens. If it is false, delimiter characters serve to separate tokens. |

#### Methods of StringTokenizer class

The 6 useful methods of StringTokenizer class are as follows:

|  |  |
| --- | --- |
| **Public method** | **Description** |
| boolean hasMoreTokens() | checks if there is more tokens available. |
| String nextToken() | returns the next token from the StringTokenizer object. |
| String nextToken(String delim) | returns the next token based on the delimeter. |
| boolean hasMoreElements() | same as hasMoreTokens() method. |
| Object nextElement() | same as nextToken() but its return type is Object. |
| int countTokens() | returns the total number of tokens. |

### **Simple example of StringTokenizer class**

Let's see the simple example of StringTokenizer class that tokenizes a string "my name is khan" on the basis of whitespace.

1. **import** java.util.StringTokenizer;
2. **public** **class** Simple{
3. **public** **static** **void** main(String args[]){
4. StringTokenizer st = **new** StringTokenizer("my name is khan"," ");
5. **while** (st.hasMoreTokens()) {
6. System.out.println(st.nextToken());
7. }
8. }
9. }

Output:my

name

is

khan

### **Example of nextToken(String delim) method of StringTokenizer class**

1. **import** java.util.\*;
3. **public** **class** Test {
4. **public** **static** **void** main(String[] args) {
5. StringTokenizer st = **new** StringTokenizer("my,name,is,khan");
7. // printing next token
8. System.out.println("Next token is : " + st.nextToken(","));
9. }
10. }

Output:Next token is : my

#### StringTokenizer class is deprecated now. It is recommended to use split() method of String class or regex (Regular Expression).

# Java String FAQs or Interview Questions

A list of top Java String FAQs (Frequently Asked Questions) or interview questions are given below. These questions can be asked by the interviewer.

### **1) How many objects will be created in the following code?**

String s1="javatpoint";

String s2="javatpoint";

**Answer:** Only one.

### **2) What is the difference between equals() method and == operator?**

The equals() method matches content of the strings whereas == operator matches object or reference of the strings.

### **3) Is String class final?**

**Answer:** Yes.

### [**4) How to reverse String in java?**](https://www.javatpoint.com/how-to-reverse-string-in-java)

Input:

this is javatpoint

Output:

tnioptavaj si siht

### [**5) How to check Palindrome String in java?**](https://www.javatpoint.com/how-to-check-palindrome-string-in-java)

Input:

nitin

Output:

true

Input:

jatin

Output:

false

### [**6) Write a java program to capitalize each word in string?**](https://www.javatpoint.com/java-program-to-capitalize-each-word-in-string)

Input:

this is javatpoint

Output:

This Is Javatpoint

### [**7) Write a java program to reverse each word in string?**](https://www.javatpoint.com/java-program-to-reverse-each-word-in-string)

Input:

this is javatpoint

Output:

siht si tnioptavaj

### [**8) Write a java program to tOGGLE each word in string?**](https://www.javatpoint.com/java-program-to-toggle-each-word-in-string)

Input:

this is javatpoint

Output:

tHIS iS jAVATPOINT

### [**9) Write a java program reverse tOGGLE each word in string?**](https://www.javatpoint.com/java-program-to-reverse-toggle-each-word-in-string)

Input:

this is javatpoint

Output:

sIHT sI tNIOPTAVAJ

### [**10) What is the difference between String and StringBuffer in java?**](https://www.javatpoint.com/difference-between-string-and-stringbuffer)

### [**11) What is the difference between StringBuffer and StringBuilder in java?**](https://www.javatpoint.com/difference-between-stringbuffer-and-stringbuilder)

### [**12) What does intern() method in java?**](https://www.javatpoint.com/java-string-intern)

### [**13) How to convert String to int in java?**](https://www.javatpoint.com/java-string-to-int)

### [**14) How to convert int to String in java?**](https://www.javatpoint.com/java-int-to-string)

### [**15) How to convert String to Date in java?**](https://www.javatpoint.com/java-string-to-date)

### [**16) How to Optimize Java String Creation?**](https://www.javatpoint.com/how-to-optimize-java-string-creation)

### [**17) Java Program to check whether two Strings are anagram or not**](https://www.javatpoint.com/java-program-to-check-whether-two-strings-are-anagram-or-not)

### [**18) Java program to find the percentage of uppercase, lowercase, digits and special characters in a String**](https://www.javatpoint.com/java-program-to-find-percentage-of-uppercase-lowercase-digits-and-special-characters)

### [**19) How to convert String to Integer and Integer to String in Java**](https://www.javatpoint.com/how-to-convert-string-to-integer-and-integer-to-string-in-java)

### [**20) Java Program to find duplicate characters in a String**](https://www.javatpoint.com/java-program-to-find-duplicate-characters-in-a-string)

### [**21) Java Program to prove that strings are immutable in java**](https://www.javatpoint.com/java-program-to-prove-that-strings-are-immutable-in-java)

### [**22) Java Program to remove all white spaces from a String**](https://www.javatpoint.com/java-program-to-remove-all-white-spaces-from-a-string)

### [**23) Java Program to check whether one String is a rotation of another**](https://www.javatpoint.com/java-program-to-check-whether-one-string-is-a-rotation-of-another)

### [**24) Java Program to count the number of words in a String**](https://www.javatpoint.com/java-program-to-count-the-number-of-words-in-a-string)

### [**25) Java Program to reverse a given String with preserving the position of space**](https://www.javatpoint.com/java-program-to-reverse-a-string-preserving-the-position-of-space)

### [**26) How to swap two String variables without third variable**](https://www.javatpoint.com/how-to-swap-two-string-variables-without-using-third-variable-in-java)

### [**27) How to remove a particular character from a String**](https://www.javatpoint.com/how-to-remove-a-particular-character-from-a-string)

How to reverse String in Java

There are many ways to reverse String in Java. We can reverse String using StringBuffer, StringBuilder, iteration etc. Let's see the ways to reverse String in Java.

1) By StringBuilder / StringBuffer

*File: StringFormatter.java*

1. **public** **class** StringFormatter {
2. **public** **static** String reverseString(String str){
3. StringBuilder sb=**new** StringBuilder(str);
4. sb.reverse();
5. **return** sb.toString();
6. }
7. }

*File: TestStringFormatter.java*

1. **public** **class** TestStringFormatter {
2. **public** **static** **void** main(String[] args) {
3. System.out.println(StringFormatter.reverseString("my name is khan"));
4. System.out.println(StringFormatter.reverseString("I am sonoo jaiswal"));
5. }
6. }

Output:

nahk si eman ym

lawsiaj oonos ma I

2) By Reverse Iteration

*File: StringFormatter.java*

1. **public** **class** StringFormatter {
2. **public** **static** String reverseString(String str){
3. **char** ch[]=str.toCharArray();
4. String rev="";
5. **for**(**int** i=ch.length-1;i>=0;i--){
6. rev+=ch[i];
7. }
8. **return** rev;
9. }
10. }

*File: TestStringFormatter.java*

1. **public** **class** TestStringFormatter {
2. **public** **static** **void** main(String[] args) {
3. System.out.println(StringFormatter.reverseString("my name is khan"));
4. System.out.println(StringFormatter.reverseString("I am sonoo jaiswal"));
5. }
6. }

Output:

nahk si eman ym

lawsiaj oonos ma I

# Java String FAQs or Interview Questions

A list of top Java String FAQs (Frequently Asked Questions) or interview questions are given below. These questions can be asked by the interviewer.

### **1) How many objects will be created in the following code?**

String s1="javatpoint";

String s2="javatpoint";

**Answer:** Only one.

### **2) What is the difference between equals() method and == operator?**

The equals() method matches content of the strings whereas == operator matches object or reference of the strings.

### **3) Is String class final?**

**Answer:** Yes.

### [**4) How to reverse String in java?**](https://www.javatpoint.com/how-to-reverse-string-in-java)

Input:

this is javatpoint

Output:

tnioptavaj si siht

### [**5) How to check Palindrome String in java?**](https://www.javatpoint.com/how-to-check-palindrome-string-in-java)

Input:

nitin

Output:

true

Input:

jatin

Output:

false

### [**6) Write a java program to capitalize each word in string?**](https://www.javatpoint.com/java-program-to-capitalize-each-word-in-string)

Input:

this is javatpoint

Output:

This Is Javatpoint

### [**7) Write a java program to reverse each word in string?**](https://www.javatpoint.com/java-program-to-reverse-each-word-in-string)

Input:

this is javatpoint

Output:

siht si tnioptavaj

### [**8) Write a java program to tOGGLE each word in string?**](https://www.javatpoint.com/java-program-to-toggle-each-word-in-string)

Input:

this is javatpoint

Output:

tHIS iS jAVATPOINT

### [**9) Write a java program reverse tOGGLE each word in string?**](https://www.javatpoint.com/java-program-to-reverse-toggle-each-word-in-string)

Input:

this is javatpoint

Output:

sIHT sI tNIOPTAVAJ

### [**10) What is the difference between String and StringBuffer in java?**](https://www.javatpoint.com/difference-between-string-and-stringbuffer)

### [**11) What is the difference between StringBuffer and StringBuilder in java?**](https://www.javatpoint.com/difference-between-stringbuffer-and-stringbuilder)

### [**12) What does intern() method in java?**](https://www.javatpoint.com/java-string-intern)

### [**13) How to convert String to int in java?**](https://www.javatpoint.com/java-string-to-int)

### [**14) How to convert int to String in java?**](https://www.javatpoint.com/java-int-to-string)

### [**15) How to convert String to Date in java?**](https://www.javatpoint.com/java-string-to-date)

### [**16) How to Optimize Java String Creation?**](https://www.javatpoint.com/how-to-optimize-java-string-creation)

### [**17) Java Program to check whether two Strings are anagram or not**](https://www.javatpoint.com/java-program-to-check-whether-two-strings-are-anagram-or-not)

### [**18) Java program to find the percentage of uppercase, lowercase, digits and special characters in a String**](https://www.javatpoint.com/java-program-to-find-percentage-of-uppercase-lowercase-digits-and-special-characters)

### [**19) How to convert String to Integer and Integer to String in Java**](https://www.javatpoint.com/how-to-convert-string-to-integer-and-integer-to-string-in-java)

### [**20) Java Program to find duplicate characters in a String**](https://www.javatpoint.com/java-program-to-find-duplicate-characters-in-a-string)

### [**21) Java Program to prove that strings are immutable in java**](https://www.javatpoint.com/java-program-to-prove-that-strings-are-immutable-in-java)

### [**22) Java Program to remove all white spaces from a String**](https://www.javatpoint.com/java-program-to-remove-all-white-spaces-from-a-string)

### [**23) Java Program to check whether one String is a rotation of another**](https://www.javatpoint.com/java-program-to-check-whether-one-string-is-a-rotation-of-another)

### [**24) Java Program to count the number of words in a String**](https://www.javatpoint.com/java-program-to-count-the-number-of-words-in-a-string)

### [**25) Java Program to reverse a given String with preserving the position of space**](https://www.javatpoint.com/java-program-to-reverse-a-string-preserving-the-position-of-space)

### [**26) How to swap two String variables without third variable**](https://www.javatpoint.com/how-to-swap-two-string-variables-without-using-third-variable-in-java)

### [**27) How to remove a particular character from a String**](https://www.javatpoint.com/how-to-remove-a-particular-character-from-a-string)

# Java String charAt()

The **java string charAt()** method returns a char value at the given index number. The index number starts from 0. It returns StringIndexOutOfBoundsException if given index number is greater than this string or negative index number.

### **Signature**

The signature of string charAt() method is given below:

1. **public** **char** charAt(**int** index)

### **Parameter**

**index** : index number, starts with 0

### **Returns**

**char value**

### **Specified by**

**CharSequence** interface

### **Throws**

**StringIndexOutOfBoundsException** : if index is negative value or greater than this string length.

## Java String charAt() method example

1. **public** **class** CharAtExample{
2. **public** **static** **void** main(String args[]){
3. String name="javatpoint";
4. **char** ch=name.charAt(4);//returns the char value at the 4th index
5. System.out.println(ch);
6. }}

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

Output:

t

## StringIndexOutOfBoundsException with charAt()

Let's see the example of charAt() method where we are passing greater index value. In such case, it throws StringIndexOutOfBoundsException at run time.

1. **public** **class** CharAtExample{
2. **public** **static** **void** main(String args[]){
3. String name="javatpoint";
4. **char** ch=name.charAt(10);//returns the char value at the 10th index
5. System.out.println(ch);
6. }}

Output:

Exception in thread "main" java.lang.StringIndexOutOfBoundsException:

String index out of range: 10

at java.lang.String.charAt(String.java:658)

at CharAtExample.main(CharAtExample.java:4)

# Java String compareTo()

The **java string compareTo()** method compares the given string with current string lexicographically. It returns positive number, negative number or 0.

It compares strings on the basis of Unicode value of each character in the strings.

If first string is lexicographically greater than second string, it returns positive number (difference of character value). If first string is less than second string lexicographically, it returns negative number and if first string is lexicographically equal to second string, it returns 0.

1. **if** s1 > s2, it returns positive number
2. **if** s1 < s2, it returns negative number
3. **if** s1 == s2, it returns 0

### **Signature**

1. **public** **int** compareTo(String anotherString)

### **Parameters**

**anotherString**: represents string that is to be compared with current string

### **Returns**

an integer value

## Java String compareTo() method example

1. **public** **class** CompareToExample{
2. **public** **static** **void** main(String args[]){
3. String s1="hello";
4. String s2="hello";
5. String s3="meklo";
6. String s4="hemlo";
7. String s5="flag";
8. System.out.println(s1.compareTo(s2));//0 because both are equal
9. System.out.println(s1.compareTo(s3));//-5 because "h" is 5 times lower than "m"
10. System.out.println(s1.compareTo(s4));//-1 because "l" is 1 times lower than "m"
11. System.out.println(s1.compareTo(s5));//2 because "h" is 2 times greater than "f"
12. }}

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

Output:

0

-5

-1

2

## Java String compareTo(): empty string

If you compare string with blank or empty string, it returns length of the string. If second string is empty, result would be positive. If first string is empty, result would be negative.

1. **public** **class** CompareToExample2{
2. **public** **static** **void** main(String args[]){
3. String s1="hello";
4. String s2="";
5. String s3="me";
6. System.out.println(s1.compareTo(s2));
7. System.out.println(s2.compareTo(s3));
8. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=CompareToExample2)

Output:

5

-2

# Java String concat

The **java string concat()** method combines specified string at the end of this string. It returns combined string. It is like appending another string.

### **Signature**

The signature of string concat() method is given below:

1. **public** String concat(String anotherString)

### **Parameter**

**anotherString** : another string i.e. to be combined at the end of this string.

### **Returns**

combined string

## Java String concat() method example

1. **public** **class** ConcatExample{
2. **public** **static** **void** main(String args[]){
3. String s1="java string";
4. s1.concat("is immutable");
5. System.out.println(s1);
6. s1=s1.concat(" is immutable so assign it explicitly");
7. System.out.println(s1);
8. }}

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

java string

java string is immutable so assign it explicitly

# Java String contains

The **java string contains()** method searches the sequence of characters in this string. It returns *true* if sequence of char values are found in this string otherwise returns *false*.

### **Signature**

The signature of string contains() method is given below:

1. **public** **boolean** contains(CharSequence sequence)

### **Parameter**

**sequence** : specifies the sequence of characters to be searched.

### **Returns**

**true** if sequence of char value exists, otherwise **false**.

### **Throws**

**NullPointerException** : if sequence is null.

## Java String contains() method example

1. **class** ContainsExample{
2. **public** **static** **void** main(String args[]){
3. String name="what do you know about me";
4. System.out.println(name.contains("do you know"));
5. System.out.println(name.contains("about"));
6. System.out.println(name.contains("hello"));
7. }}

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

true

true

false

# Java String endsWith

The **java string endsWith()** method checks if this string ends with given suffix. It returns true if this string ends with given suffix else returns false.

### **Signature**

The syntax or signature of endsWith() method is given below.

1. **public** **boolean** endsWith(String suffix)

### **Parameter**

**suffix** : Sequence of character

### **Returns**

true or false

## Java String endsWith() method example

1. **public** **class** EndsWithExample{
2. **public** **static** **void** main(String args[]){
3. String s1="java by javatpoint";
4. System.out.println(s1.endsWith("t"));
5. System.out.println(s1.endsWith("point"));
6. }}

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=EndsWithExample)

Output:

true

true

# Java String equals

The **java string equals()** method compares the two given strings based on the content of the string. If any character is not matched, it returns false. If all characters are matched, it returns true.

The String equals() method overrides the equals() method of Object class.

### **Signature**

1. **public** **boolean** equals(Object anotherObject)

### **Parameter**

**anotherObject** : another object i.e. compared with this string.

### **Returns**

**true** if characters of both strings are equal otherwise **false**.

### **Overrides**

equals() method of java Object class.

## Java String equals() method example

1. **public** **class** EqualsExample{
2. **public** **static** **void** main(String args[]){
3. String s1="javatpoint";
4. String s2="javatpoint";
5. String s3="JAVATPOINT";
6. String s4="python";
7. System.out.println(s1.equals(s2));//true because content and case is same
8. System.out.println(s1.equals(s3));//false because case is not same
9. System.out.println(s1.equals(s4));//false because content is not same
10. }}

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

true

false

false

# Java String equalsIgnoreCase()

The **String equalsIgnoreCase()** method compares the two given strings on the basis of content of the string irrespective of case of the string. It is like equals() method but doesn't check case. If any character is not matched, it returns false otherwise it returns true.

### **Signature**

1. **public** **boolean** equalsIgnoreCase(String str)

### **Parameter**

**str** : another string i.e. compared with this string.

### **Returns**

It returns **true** if characters of both strings are equal ignoring case otherwise **false**.

## Java String equalsIgnoreCase() method example

1. **public** **class** EqualsIgnoreCaseExample{
2. **public** **static** **void** main(String args[]){
3. String s1="javatpoint";
4. String s2="javatpoint";
5. String s3="JAVATPOINT";
6. String s4="python";
7. System.out.println(s1.equalsIgnoreCase(s2));//true because content and case both are same
8. System.out.println(s1.equalsIgnoreCase(s3));//true because case is ignored
9. System.out.println(s1.equalsIgnoreCase(s4));//false because content is not same
10. }}

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

true

true

false

# Java String format

The **java string format()** method returns the formatted string by given locale, format and arguments.

If you don't specify the locale in String.format() method, it uses default locale by calling *Locale.getDefault()* method.

The format() method of java language is like *sprintf()* function in c language and *printf()* method of java language.

### **Signature**

There are two type of string format() method:

1. **public** **static** String format(String format, Object... args)
2. and,
3. **public** **static** String format(Locale locale, String format, Object... args)

### **Parameters**

**locale** : specifies the locale to be applied on the format() method.

**format** : format of the string.

**args** : arguments for the format string. It may be zero or more.

### **Returns**

formatted string

### **Throws**

**NullPointerException** : if format is null.

**IllegalFormatException** : if format is illegal or incompatible.

## Java String format() method example

1. **public** **class** FormatExample{
2. **public** **static** **void** main(String args[]){
3. String name="sonoo";
4. String sf1=String.format("name is %s",name);
5. String sf2=String.format("value is %f",32.33434);
6. String sf3=String.format("value is %32.12f",32.33434);//returns 12 char fractional part filling with 0
8. System.out.println(sf1);
9. System.out.println(sf2);
10. System.out.println(sf3);
11. }}

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

name is sonoo

value is 32.334340

value is 32.334340000000

# Java String getBytes()

The **java string getBytes()** method returns the byte array of the string. In other words, it returns sequence of bytes.

### **Signature**

There are 3 variant of getBytes() method. The signature or syntax of string getBytes() method is given below:

1. **public** **byte**[] getBytes()
2. **public** **byte**[] getBytes(Charset charset)
3. **public** **byte**[] getBytes(String charsetName)**throws** UnsupportedEncodingException

### **Returns**

sequence of bytes.

## Java String getBytes() method example

1. **public** **class** StringGetBytesExample{
2. **public** **static** **void** main(String args[]){
3. String s1="ABCDEFG";
4. **byte**[] barr=s1.getBytes();
5. **for**(**int** i=0;i<barr.length;i++){
6. System.out.println(barr[i]);
7. }
8. }}

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

Output:

65

66

67

68

69

70

71

# Java String getChars()

The **java string getChars()** method copies the content of this string into specified char array. There are 4 arguments passed in getChars() method. The signature of getChars() method is given below:

### **Signature**

The signature or syntax of string getChars() method is given below:

1. **public** **void** getChars(**int** srcBeginIndex, **int** srcEndIndex, **char**[] destination, **int** dstBeginIndex)

### **Returns**

It doesn't return any value.

### **Throws**

It throws StringIndexOutOfBoundsException if beginIndex is greater than endIndex.

## Java String getChars() method example

1. **public** **class** StringGetCharsExample{
2. **public** **static** **void** main(String args[]){
3. String str = **new** String("hello javatpoint how r u");
4. **char**[] ch = **new** **char**[10];
5. **try**{
6. str.getChars(6, 16, ch, 0);
7. System.out.println(ch);
8. }**catch**(Exception ex){System.out.println(ex);}
9. }}

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

Output:

javatpoint

# Java String indexOf

The **java string indexOf()** method returns index of given character value or substring. If it is not found, it returns -1. The index counter starts from zero.

### **Signature**

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | int indexOf(int ch) | returns index position for the given char value |
| 2 | int indexOf(int ch, int fromIndex) | returns index position for the given char value and from index |
| 3 | int indexOf(String substring) | returns index position for the given substring |
| 4 | int indexOf(String substring, int fromIndex) | returns index position for the given substring and from index |

There are 4 types of indexOf method in java. The signature of indexOf methods are given below:

### **Parameters**

**ch**: char value i.e. a single character e.g. 'a'

**fromIndex**: index position from where index of the char value or substring is retured

**substring**: substring to be searched in this string

### **Returns**

index of the string

## Java String indexOf() method example

1. **public** **class** IndexOfExample{
2. **public** **static** **void** main(String args[]){
3. String s1="this is index of example";
4. //passing substring
5. **int** index1=s1.indexOf("is");//returns the index of is substring
6. **int** index2=s1.indexOf("index");//returns the index of index substring
7. System.out.println(index1+"  "+index2);//2 8
9. //passing substring with from index
10. **int** index3=s1.indexOf("is",4);//returns the index of is substring after 4th index
11. System.out.println(index3);//5 i.e. the index of another is
13. //passing char value
14. **int** index4=s1.indexOf('s');//returns the index of s char value
15. System.out.println(index4);//3
16. }}

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

2 8

5

3

# Java String intern

The **java string intern()** method returns the interned string. It returns the canonical representation of string.

It can be used to return string from pool memory, if it is created by new keyword.

### **Signature**

The signature of intern method is given below:

1. **public** String intern()

### **Returns**

interned string

## Java String intern() method example

1. **public** **class** InternExample{
2. **public** **static** **void** main(String args[]){
3. String s1=**new** String("hello");
4. String s2="hello";
5. String s3=s1.intern();//returns string from pool, now it will be same as s2
6. System.out.println(s1==s2);//false because reference is different
7. System.out.println(s2==s3);//true because reference is same
8. }}

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

false

true

# Java String isEmpty

The **java string isEmpty()** method checks if this string is empty. It returns *true*, if length of string is 0 otherwise *false*.

The isEmpty() method of String class is included in java string since JDK 1.6.

### **Signature**

The signature or syntax of string isEmpty() method is given below:

1. **public** **boolean** isEmpty()

### **Returns**

true if length is 0 otherwise false.

### **Since**

**1.6**

## Java String isEmpty() method example

1. **public** **class** IsEmptyExample{
2. **public** **static** **void** main(String args[]){
3. String s1="";
4. String s2="javatpoint";
6. System.out.println(s1.isEmpty());
7. System.out.println(s2.isEmpty());
8. }}

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

true

false

# Java String join

The **java string join()** method returns a string joined with given delimiter. In string join method, delimiter is copied for each elements.

In case of null element, "null" is added. The join() method is included in java string since JDK 1.8.

There are two types of join() methods in java string.

### **Signature**

The signature or syntax of string join method is given below:

1. **public** **static** String join(CharSequence delimiter, CharSequence... elements)
2. and
3. **public** **static** String join(CharSequence delimiter, Iterable<? **extends** CharSequence> elements)

### **Parameters**

**delimiter** : char value to be added with each element

**elements** : char value to be attached with delimiter

### **Returns**

joined string with delimiter

### **Throws**

**NullPointerException** if element or delimiter is null.

### **Since**

**1.8**

## Java String join() method example

1. **public** **class** StringJoinExample{
2. **public** **static** **void** main(String args[]){
3. String joinString1=String.join("-","welcome","to","javatpoint");
4. System.out.println(joinString1);
5. }}

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

welcome-to-javatpoint

# Java String lastIndexOf

The **java string lastIndexOf()** method returns last index of the given character value or substring. If it is not found, it returns -1. The index counter starts from zero.

### **Signature**

There are 4 types of lastIndexOf method in java. The signature of lastIndexOf methods are given below:

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | int lastIndexOf(int ch) | returns last index position for the given char value |
| 2 | int lastIndexOf(int ch, int fromIndex) | returns last index position for the given char value and from index |
| 3 | int lastIndexOf(String substring) | returns last index position for the given substring |
| 4 | int lastIndexOf(String substring, int fromIndex) | returns last index position for the given substring and from index |

### **Parameters**

**ch**: char value i.e. a single character e.g. 'a'

**fromIndex**: index position from where index of the char value or substring is retured

**substring**: substring to be searched in this string

### **Returns**

last index of the string

## Java String lastIndexOf() method example

1. **public** **class** LastIndexOfExample{
2. **public** **static** **void** main(String args[]){
3. String s1="this is index of example";//there are 2 's' characters in this sentence
4. **int** index1=s1.lastIndexOf('s');//returns last index of 's' char value
5. System.out.println(index1);//6
6. }}

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=LastIndexOfExample)

Output:

6

# Java String length

The **java string length()** method length of the string. It returns count of total number of characters. The length of java string is same as the unicode code units of the string.

### **Signature**

The signature of the string length() method is given below:

1. **public** **int** length()

### **Specified by**

CharSequence interface

### **Returns**

length of characters

## Java String length() method example

1. **public** **class** LengthExample{
2. **public** **static** **void** main(String args[]){
3. String s1="javatpoint";
4. String s2="python";
5. System.out.println("string length is: "+s1.length());//10 is the length of javatpoint string
6. System.out.println("string length is: "+s2.length());//6 is the length of python string
7. }}

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

string length is: 10

string length is: 6

# Java String replace

The **java string replace()** method returns a string replacing all the old char or CharSequence to new char or CharSequence.

Since JDK 1.5, a new replace() method is introduced, allowing you to replace a sequence of char values.

### **Signature**

There are two type of replace methods in java string.

1. **public** String replace(**char** oldChar, **char** newChar)
2. and
3. **public** String replace(CharSequence target, CharSequence replacement)

The second replace method is added since JDK 1.5.

### **Parameters**

**oldChar** : old character

**newChar** : new character

**target** : target sequence of characters

**replacement** : replacement sequence of characters

### **Returns**

replaced string

## Java String replace(char old, char new) method example

1. **public** **class** ReplaceExample1{
2. **public** **static** **void** main(String args[]){
3. String s1="javatpoint is a very good website";
4. String replaceString=s1.replace('a','e');//replaces all occurrences of 'a' to 'e'
5. System.out.println(replaceString);
6. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=ReplaceExample1)

jevetpoint is e very good website

## Java String replace(CharSequence target, CharSequence replacement) method example

1. **public** **class** ReplaceExample2{
2. **public** **static** **void** main(String args[]){
3. String s1="my name is khan my name is java";
4. String replaceString=s1.replace("is","was");//replaces all occurrences of "is" to "was"
5. System.out.println(replaceString);
6. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=ReplaceExample2)

my name was khan my name was java

# Java String replaceAll

The **java string replaceAll()** method returns a string replacing all the sequence of characters matching regex and replacement string.

### **Signature**

1. **public** String replaceAll(String regex, String replacement)

### **Parameters**

**regex** : regular expression

**replacement** : replacement sequence of characters

### **Returns**

replaced string

## Java String replaceAll() example: replace character

Let's see an example to replace all the occurrences of **a single character**.

1. **public** **class** ReplaceAllExample1{
2. **public** **static** **void** main(String args[]){
3. String s1="javatpoint is a very good website";
4. String replaceString=s1.replaceAll("a","e");//replaces all occurrences of "a" to "e"
5. System.out.println(replaceString);
6. }}

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=ReplaceAllExample1)

jevetpoint is e very good website

## Java String replaceAll() example: replace word

Let's see an example to replace all the occurrences of **single word or set of words**.

1. **public** **class** ReplaceAllExample2{
2. **public** **static** **void** main(String args[]){
3. String s1="My name is Khan. My name is Bob. My name is Sonoo.";
4. String replaceString=s1.replaceAll("is","was");//replaces all occurrences of "is" to "was"
5. System.out.println(replaceString);
6. }}

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=ReplaceAllExample2)

My name was Khan. My name was Bob. My name was Sonoo.

## Java String replaceAll() example: remove white spaces

Let's see an example to remove all the occurrences of **white spaces**.

1. **public** **class** ReplaceAllExample3{
2. **public** **static** **void** main(String args[]){
3. String s1="My name is Khan. My name is Bob. My name is Sonoo.";
4. String replaceString=s1.replaceAll("\\s","");
5. System.out.println(replaceString);
6. }}

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=ReplaceAllExample3)

MynamewasKhan.MynamewasBob.MynamewasSonoo.

[Click me to know about regex](https://www.javatpoint.com/java-regex)

# Java String split

The **java string split()** method splits this string against given regular expression and returns a char array.

### **Signature**

There are two signature for split() method in java string.

1. **public** String split(String regex)
2. and,
3. **public** String split(String regex, **int** limit)

### **Parameter**

**regex** : regular expression to be applied on string.

**limit** : limit for the number of strings in array. If it is zero, it will returns all the strings matching regex.

### **Returns**

array of strings

### **Throws**

**PatternSyntaxException** if pattern for regular expression is invalid

### **Since**

1.4

## Java String split() method example

The given example returns total number of words in a string excluding space only. It also includes special characters.

1. **public** **class** SplitExample{
2. **public** **static** **void** main(String args[]){
3. String s1="java string split method by javatpoint";
4. String[] words=s1.split("\\s");//splits the string based on whitespace
5. //using java foreach loop to print elements of string array
6. **for**(String w:words){
7. System.out.println(w);
8. }
9. }}

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

java

string

split

method

by

javatpoint

## Java String split() method with regex and length example

1. **public** **class** SplitExample2{
2. **public** **static** **void** main(String args[]){
3. String s1="welcome to split world";
4. System.out.println("returning words:");
5. **for**(String w:s1.split("\\s",0)){
6. System.out.println(w);
7. }
8. System.out.println("returning words:");
9. **for**(String w:s1.split("\\s",1)){
10. System.out.println(w);
11. }
12. System.out.println("returning words:");
13. **for**(String w:s1.split("\\s",2)){
14. System.out.println(w);
15. }
17. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=SplitExample2)

returning words:

welcome

to

split

world

returning words:

welcome to split world

returning words:

welcome

to split world

# Java String startsWith

The **java string startsWith()** method checks if this string starts with given prefix. It returns true if this string starts with given prefix else returns false.

### **Signature**

The syntax or signature of startWith() method is given below.

1. **public** **boolean** startsWith(String prefix)
2. **public** **boolean** startsWith(String prefix, **int** offset)

### **Parameter**

**prefix** : Sequence of character

### **Returns**

true or false

## Java String startsWith() method example

1. **public** **class** StartsWithExample{
2. **public** **static** **void** main(String args[]){
3. String s1="java string split method by javatpoint";
4. System.out.println(s1.startsWith("ja"));
5. System.out.println(s1.startsWith("java string"));
6. }}

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=StartsWithExample)

Output:

true

true

# Java String substring

The **java string substring()** method returns a part of the string.

We pass begin index and end index number position in the java substring method where start index is inclusive and end index is exclusive. In other words, start index starts from 0 whereas end index starts from 1.

There are two types of substring methods in java string.

### **Signature**

1. **public** String substring(**int** startIndex)
2. and
3. **public** String substring(**int** startIndex, **int** endIndex)

If you don't specify endIndex, java substring() method will return all the characters from startIndex.

### **Parameters**

**startIndex** : starting index is inclusive

**endIndex** : ending index is exclusive

### **Returns**

specified string

### **Throws**

**StringIndexOutOfBoundsException** if start index is negative value or end index is lower than starting index.

## Java String substring() method example

1. **public** **class** SubstringExample{
2. **public** **static** **void** main(String args[]){
3. String s1="javatpoint";
4. System.out.println(s1.substring(2,4));//returns va
5. System.out.println(s1.substring(2));//returns vatpoint
6. }}

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

va

vatpoint

# Java String toCharArray

The **java string toCharArray()** method converts this string into character array. It returns a newly created character array, its length is similar to this string and its contents are initialized with the characters of this string.

### **Signature**

The signature or syntax of string toCharArray() method is given below:

1. **public** **char**[] toCharArray()

### **Returns**

character array

## Java String toCharArray() method example

1. **public** **class** StringToCharArrayExample{
2. **public** **static** **void** main(String args[]){
3. String s1="hello";
4. **char**[] ch=s1.toCharArray();
5. **for**(**int** i=0;i<ch.length;i++){
6. System.out.print(ch[i]);
7. }
8. }}

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=StringToCharArrayExample)

Output:

hello

# Java String toLowerCase()

The **java string toLowerCase()** method returns the string in lowercase letter. In other words, it converts all characters of the string into lower case letter.

The toLowerCase() method works same as toLowerCase(Locale.getDefault()) method. It internally uses the default locale.

### **Signature**

There are two variant of toLowerCase() method. The signature or syntax of string toLowerCase() method is given below:

1. **public** String toLowerCase()
2. **public** String toLowerCase(Locale locale)

The second method variant of toLowerCase(), converts all the characters into lowercase using the rules of given Locale.

### **Returns**

string in lowercase letter.

## Java String toLowerCase() method example

1. **public** **class** StringLowerExample{
2. **public** **static** **void** main(String args[]){
3. String s1="JAVATPOINT HELLO stRIng";
4. String s1lower=s1.toLowerCase();
5. System.out.println(s1lower);
6. }}

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

Output:

javatpoint hello string

# Java String toUpperCase

The **java string toUpperCase()** method returns the string in uppercase letter. In other words, it converts all characters of the string into upper case letter.

The toUpperCase() method works same as toUpperCase(Locale.getDefault()) method. It internally uses the default locale.

### **Signature**

There are two variant of toUpperCase() method. The signature or syntax of string toUpperCase() method is given below:

1. **public** String toUpperCase()
2. **public** String toUpperCase(Locale locale)

The second method variant of toUpperCase(), converts all the characters into uppercase using the rules of given Locale.

### **Returns**

string in uppercase letter.

## Java String toUpperCase() method example

1. **public** **class** StringUpperExample{
2. **public** **static** **void** main(String args[]){
3. String s1="hello string";
4. String s1upper=s1.toUpperCase();
5. System.out.println(s1upper);
6. }}

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

Output:

HELLO STRING

# Java String trim

The **java string trim()** method eliminates leading and trailing spaces. The unicode value of space character is '\u0020'. The trim() method in java string checks this unicode value before and after the string, if it exists then removes the spaces and returns the omitted string.

#### The string trim() method doesn't omits middle spaces.

### **Signature**

The signature or syntax of string trim method is given below:

1. **public** String trim()

### **Returns**

string with omitted leading and trailing spaces

## Java String trim() method example

1. **public** **class** StringTrimExample{
2. **public** **static** **void** main(String args[]){
3. String s1="  hello string   ";
4. System.out.println(s1+"javatpoint");//without trim()
5. System.out.println(s1.trim()+"javatpoint");//with trim()
6. }}

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

hello string javatpoint

hello stringjavatpoint

# Java String valueOf

The **java string valueOf()** method converts different types of values into string. By the help of string valueOf() method, you can convert int to string, long to string, boolean to string, character to string, float to string, double to string, object to string and char array to string.

### **Signature**

The signature or syntax of string valueOf() method is given below:

1. **public** **static** String valueOf(**boolean** b)
2. **public** **static** String valueOf(**char** c)
3. **public** **static** String valueOf(**char**[] c)
4. **public** **static** String valueOf(**int** i)
5. **public** **static** String valueOf(**long** l)
6. **public** **static** String valueOf(**float** f)
7. **public** **static** String valueOf(**double** d)
8. **public** **static** String valueOf(Object o)

### **Returns**

string representation of given value

## Java String valueOf() method example

1. **public** **class** StringValueOfExample{
2. **public** **static** **void** main(String args[]){
3. **int** value=30;
4. String s1=String.valueOf(value);
5. System.out.println(s1+10);//concatenating string with 10
6. }}

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=StringValueOfExample)

Output:

3010

# Java Regex

The **Java Regex** or Regular Expression is an API to *define pattern for searching or manipulating strings*.

It is widely used to define constraint on strings such as password and email validation. After learning java regex tutorial, you will be able to test your own regular expressions by the Java Regex Tester Tool.

Java Regex API provides 1 interface and 3 classes in **java.util.regex** package.

#### java.util.regex package

It provides following classes and interface for regular expressions. The Matcher and Pattern classes are widely used in java regular expression.

1. MatchResult interface
2. Matcher class
3. Pattern class
4. PatternSyntaxException class

## Matcher class

It implements **MatchResult** interface. It is a *regex engine* i.e. used to perform match operations on a character sequence.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | boolean matches() | test whether the regular expression matches the pattern. |
| 2 | boolean find() | finds the next expression that matches the pattern. |
| 3 | boolean find(int start) | finds the next expression that matches the pattern from the given start number. |
| 4 | String group() | returns the matched subsequence. |
| 5 | int start() | returns the starting index of the matched subsequence. |
| 6 | int end() | returns the ending index of the matched subsequence. |
| 7 | int groupCount() | returns the total number of the matched subsequence. |

## Pattern class

It is the *compiled version of a regular expression*. It is used to define a pattern for the regex engine.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | static Pattern compile(String regex) | compiles the given regex and return the instance of pattern. |
| 2 | Matcher matcher(CharSequence input) | creates a matcher that matches the given input with pattern. |
| 3 | static boolean matches(String regex, CharSequence input) | It works as the combination of compile and matcher methods. It compiles the regular expression and matches the given input with the pattern. |
| 4 | String[] split(CharSequence input) | splits the given input string around matches of given pattern. |
| 5 | String pattern() | returns the regex pattern. |

### **Example of Java Regular Expressions**

There are three ways to write the regex example in java.

1. **import** java.util.regex.\*;
2. **public** **class** RegexExample1{
3. **public** **static** **void** main(String args[]){
4. //1st way
5. Pattern p = Pattern.compile(".s");//. represents single character
6. Matcher m = p.matcher("as");
7. **boolean** b = m.matches();
9. //2nd way
10. **boolean** b2=Pattern.compile(".s").matcher("as").matches();
12. //3rd way
13. **boolean** b3 = Pattern.matches(".s", "as");
15. System.out.println(b+" "+b2+" "+b3);
16. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=RegexExample1)

#### Output

true true true

## Regular Expression . Example

The . (dot) represents a single character.

1. **import** java.util.regex.\*;
2. **class** RegexExample2{
3. **public** **static** **void** main(String args[]){
4. System.out.println(Pattern.matches(".s", "as"));//true (2nd char is s)
5. System.out.println(Pattern.matches(".s", "mk"));//false (2nd char is not s)
6. System.out.println(Pattern.matches(".s", "mst"));//false (has more than 2 char)
7. System.out.println(Pattern.matches(".s", "amms"));//false (has more than 2 char)
8. System.out.println(Pattern.matches("..s", "mas"));//true (3rd char is s)
9. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=RegexExample2)

## Regex Character classes

|  |  |  |
| --- | --- | --- |
| **No.** | **Character Class** | **Description** |
| 1 | [abc] | a, b, or c (simple class) |
| 2 | [^abc] | Any character except a, b, or c (negation) |
| 3 | [a-zA-Z] | a through z or A through Z, inclusive (range) |
| 4 | [a-d[m-p]] | a through d, or m through p: [a-dm-p] (union) |
| 5 | [a-z&&[def]] | d, e, or f (intersection) |
| 6 | [a-z&&[^bc]] | a through z, except for b and c: [ad-z] (subtraction) |
| 7 | [a-z&&[^m-p]] | a through z, and not m through p: [a-lq-z](subtraction) |

## Regular Expression Character classes Example

1. **import** java.util.regex.\*;
2. **class** RegexExample3{
3. **public** **static** **void** main(String args[]){
4. System.out.println(Pattern.matches("[amn]", "abcd"));//false (not a or m or n)
5. System.out.println(Pattern.matches("[amn]", "a"));//true (among a or m or n)
6. System.out.println(Pattern.matches("[amn]", "ammmna"));//false (m and a comes more than once)
7. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=RegexExample3)

## Regex Quantifiers

The quantifiers specify the number of occurrences of a character.

|  |  |
| --- | --- |
| **Regex** | **Description** |
| X? | X occurs once or not at all |
| X+ | X occurs once or more times |
| X\* | X occurs zero or more times |
| X{n} | X occurs n times only |
| X{n,} | X occurs n or more times |
| X{y,z} | X occurs at least y times but less than z times |

## Regular Expression Character classes and Quantifiers Example

1. **import** java.util.regex.\*;
2. **class** RegexExample4{
3. **public** **static** **void** main(String args[]){
4. System.out.println("? quantifier ....");
5. System.out.println(Pattern.matches("[amn]?", "a"));//true (a or m or n comes one time)
6. System.out.println(Pattern.matches("[amn]?", "aaa"));//false (a comes more than one time)
7. System.out.println(Pattern.matches("[amn]?", "aammmnn"));//false (a m and n comes more than one time)
8. System.out.println(Pattern.matches("[amn]?", "aazzta"));//false (a comes more than one time)
9. System.out.println(Pattern.matches("[amn]?", "am"));//false (a or m or n must come one time)
11. System.out.println("+ quantifier ....");
12. System.out.println(Pattern.matches("[amn]+", "a"));//true (a or m or n once or more times)
13. System.out.println(Pattern.matches("[amn]+", "aaa"));//true (a comes more than one time)
14. System.out.println(Pattern.matches("[amn]+", "aammmnn"));//true (a or m or n comes more than once)
15. System.out.println(Pattern.matches("[amn]+", "aazzta"));//false (z and t are not matching pattern)
17. System.out.println("\* quantifier ....");
18. System.out.println(Pattern.matches("[amn]\*", "ammmna"));//true (a or m or n may come zero or more times)
20. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=RegexExample4)

## Regex Metacharacters

The regular expression metacharacters work as a short codes.

|  |  |
| --- | --- |
| **Regex** | **Description** |
| . | Any character (may or may not match terminator) |
| \d | Any digits, short of [0-9] |
| \D | Any non-digit, short for [^0-9] |
| \s | Any whitespace character, short for [\t\n\x0B\f\r] |
| \S | Any non-whitespace character, short for [^\s] |
| \w | Any word character, short for [a-zA-Z\_0-9] |
| \W | Any non-word character, short for [^\w] |
| \b | A word boundary |
| \B | A non word boundary |

## Regular Expression Metacharacters Example

1. **import** java.util.regex.\*;
2. **class** RegexExample5{
3. **public** **static** **void** main(String args[]){
4. System.out.println("metacharacters d....");\\d means digit
6. System.out.println(Pattern.matches("\\d", "abc"));//false (non-digit)
7. System.out.println(Pattern.matches("\\d", "1"));//true (digit and comes once)
8. System.out.println(Pattern.matches("\\d", "4443"));//false (digit but comes more than once)
9. System.out.println(Pattern.matches("\\d", "323abc"));//false (digit and char)
11. System.out.println("metacharacters D....");\\D means non-digit
13. System.out.println(Pattern.matches("\\D", "abc"));//false (non-digit but comes more than once)
14. System.out.println(Pattern.matches("\\D", "1"));//false (digit)
15. System.out.println(Pattern.matches("\\D", "4443"));//false (digit)
16. System.out.println(Pattern.matches("\\D", "323abc"));//false (digit and char)
17. System.out.println(Pattern.matches("\\D", "m"));//true (non-digit and comes once)
19. System.out.println("metacharacters D with quantifier....");
20. System.out.println(Pattern.matches("\\D\*", "mak"));//true (non-digit and may come 0 or more times)
22. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=RegexExample5)

## Regular Expression Question 1

1. /\*Create a regular expression that accepts alpha numeric characters only. Its
2. length must be 6 characters long only.\*/
4. **import** java.util.regex.\*;
5. **class** RegexExample6{
6. **public** **static** **void** main(String args[]){
7. System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "arun32"));//true
8. System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "kkvarun32"));//false (more than 6 char)
9. System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "JA2Uk2"));//true
10. System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "arun$2"));//false ($ is not matched)
11. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=RegexExample6)

## Regular Expression Question 2

1. /\*Create a regular expression that accepts 10 digit numeric characters
2. starting with 7, 8 or 9 only.\*/
4. **import** java.util.regex.\*;
5. **class** RegexExample7{
6. **public** **static** **void** main(String args[]){
7. System.out.println("by character classes and quantifiers ...");
8. System.out.println(Pattern.matches("[789]{1}[0-9]{9}", "9953038949"));//true
9. System.out.println(Pattern.matches("[789][0-9]{9}", "9953038949"));//true
11. System.out.println(Pattern.matches("[789][0-9]{9}", "99530389490"));//false (11 characters)
12. System.out.println(Pattern.matches("[789][0-9]{9}", "6953038949"));//false (starts from 6)
13. System.out.println(Pattern.matches("[789][0-9]{9}", "8853038949"));//true
15. System.out.println("by metacharacters ...");
16. System.out.println(Pattern.matches("[789]{1}\\d{9}", "8853038949"));//true
17. System.out.println(Pattern.matches("[789]{1}\\d{9}", "3853038949"));//false (starts from 3)
19. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=RegexExample7)

## Java Regex Finder Example

1. **import** java.util.regex.Pattern;
2. **import** java.util.Scanner;
3. **import** java.util.regex.Matcher;
4. **public** **class** RegexExample8{
5. **public** **static** **void** main(String[] args){
6. Scanner sc=**new** Scanner(System.in);
7. **while** (**true**) {
8. System.out.println("Enter regex pattern:");
9. Pattern pattern = Pattern.compile(sc.nextLine());
10. System.out.println("Enter text:");
11. Matcher matcher = pattern.matcher(sc.nextLine());
12. **boolean** found = **false**;
13. **while** (matcher.find()) {
14. System.out.println("I found the text "+matcher.group()+" starting at index "+
15. matcher.start()+" and ending at index "+matcher.end());
16. found = **true**;
17. }
18. **if**(!found){
19. System.out.println("No match found.");
20. }
21. }
22. }
23. }

Output:

Enter regex pattern: java

Enter text: this is java, do you know java

I found the text java starting at index 8 and ending at index 12

I found the text java starting at index 26 and ending at index 30

# 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:

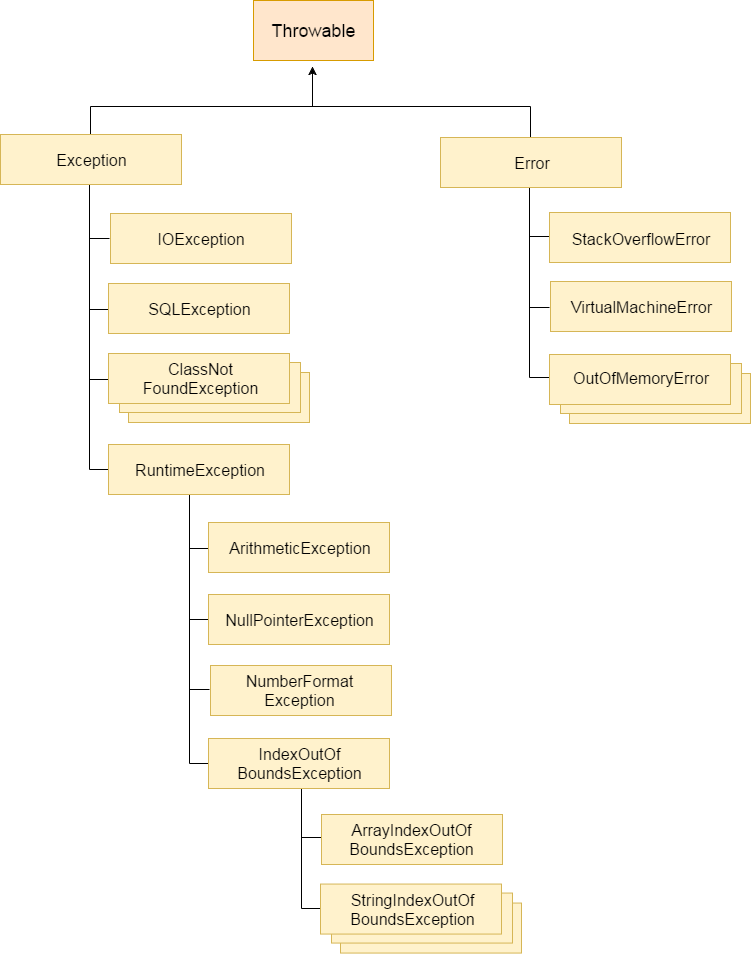
1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5;//exception occurs
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;

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);}
6. System.out.println("rest of the code...");
7. }
8. }

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

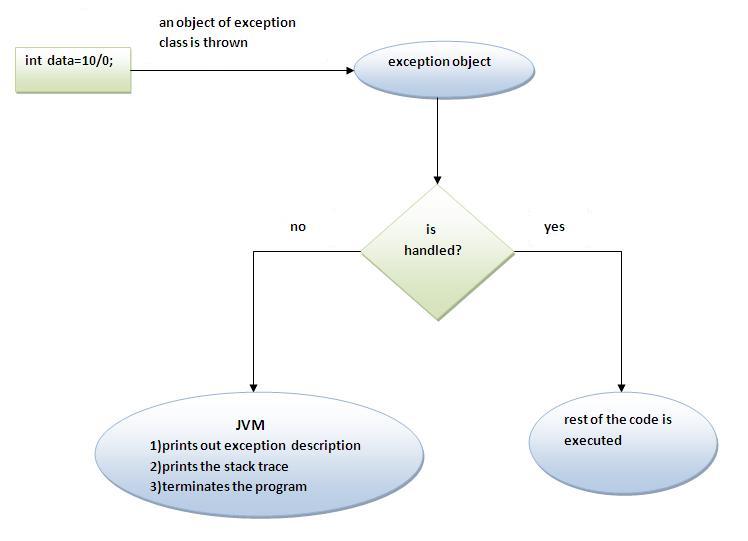
Output:

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 catch multiple exceptions

Java Multi catch block

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

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

1. **public** **class** TestMultipleCatchBlock{
2. **public** **static** **void** main(String args[]){
3. **try**{
4. **int** a[]=**new** **int**[5];
5. a[5]=30/0;
6. }
7. **catch**(ArithmeticException e){System.out.println("task1 is completed");}
8. **catch**(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
9. **catch**(Exception e){System.out.println("common task completed");}
11. System.out.println("rest of the code...");
12. }
13. }

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

Output:task1 completed

rest of the code...

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

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

1. **class** TestMultipleCatchBlock1{
2. **public** **static** **void** main(String args[]){
3. **try**{
4. **int** a[]=**new** **int**[5];
5. a[5]=30/0;
6. }
7. **catch**(Exception e){System.out.println("common task completed");}
8. **catch**(ArithmeticException e){System.out.println("task1 is completed");}
9. **catch**(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
10. System.out.println("rest of the code...");
11. }
12. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestMultipleCatchBlock1)

Output:

Compile-time error

# Java Nested try block

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

### **Why use nested try block**

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

### **Syntax:**

1. ....
2. **try**
3. {
4. statement 1;
5. statement 2;
6. **try**
7. {
8. statement 1;
9. statement 2;
10. }
11. **catch**(Exception e)
12. {
13. }
14. }
15. **catch**(Exception e)
16. {
17. }
18. ....

## Java nested try example

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

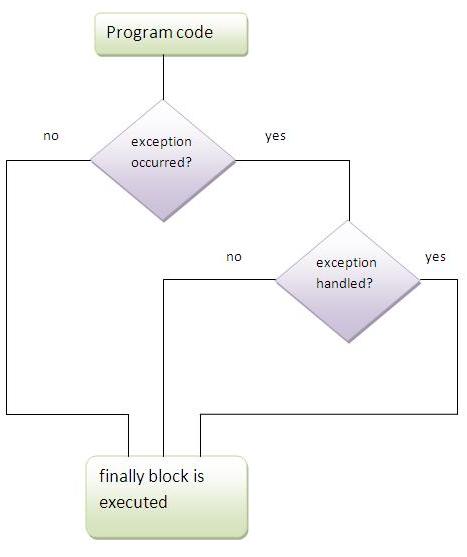
1. **class** Excep6{
2. **public** **static** **void** main(String args[]){
3. **try**{
4. **try**{
5. System.out.println("going to divide");
6. **int** b =39/0;
7. }**catch**(ArithmeticException e){System.out.println(e);}
9. **try**{
10. **int** a[]=**new** **int**[5];
11. a[5]=4;
12. }**catch**(ArrayIndexOutOfBoundsException e){System.out.println(e);}
14. System.out.println("other statement);
15. }**catch**(Exception e){System.out.println("handeled");}
17. System.out.println("normal flow..");
18. }
19. }

# Java finally block

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

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

Java finally block follows try or catch block.



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

## Why use java finally

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

## Usage of Java finally

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

### **Case 1**

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

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).

Java throw exception

Java throw keyword

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

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

The syntax of java throw keyword is given below.

1. **throw** exception;

Let's see the example of throw IOException.

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

java throw keyword example

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

1. **public** **class** TestThrow1{
2. **static** **void** validate(**int** age){
3. **if**(age<18)
4. **throw** **new** ArithmeticException("not valid");
5. **else**
6. System.out.println("welcome to vote");
7. }
8. **public** **static** **void** main(String args[]){
9. validate(13);
10. System.out.println("rest of the code...");
11. }
12. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestThrow1)

Output:

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

Java Exception propagation

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

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

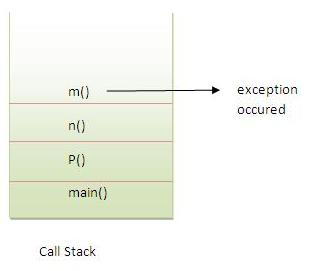
***Program of Exception Propagation***

1. **class** TestExceptionPropagation1{
2. **void** m(){
3. **int** data=50/0;
4. }
5. **void** n(){
6. m();
7. }
8. **void** p(){
9. **try**{
10. n();
11. }**catch**(Exception e){System.out.println("exception handled");}
12. }
13. **public** **static** **void** main(String args[]){
14. TestExceptionPropagation1 obj=**new** TestExceptionPropagation1();
15. obj.p();
16. System.out.println("normal flow...");
17. }
18. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestExceptionPropagation1)

Output:exception handled

normal flow...



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

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

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

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

1. **class** TestExceptionPropagation2{
2. **void** m(){
3. **throw** **new** java.io.IOException("device error");//checked exception
4. }
5. **void** n(){
6. m();
7. }
8. **void** p(){
9. **try**{
10. n();
11. }**catch**(Exception e){System.out.println("exception handeled");}
12. }
13. **public** **static** **void** main(String args[]){
14. TestExceptionPropagation2 obj=**new** TestExceptionPropagation2();
15. obj.p();
16. System.out.println("normal flow");
17. }
18. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestExceptionPropagation2)

Output:Compile Time Error

# Java throws keyword

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

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

### **Syntax of java throws**

1. return\_type method\_name() **throws** exception\_class\_name{
2. //method code
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.

## Java throws example

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

1. **import** java.io.IOException;
2. **class** Testthrows1{
3. **void** m()**throws** IOException{
4. **throw** **new** IOException("device error");//checked exception
5. }
6. **void** n()**throws** IOException{
7. m();
8. }
9. **void** p(){
10. **try**{
11. n();
12. }**catch**(Exception e){System.out.println("exception handled");}
13. }
14. **public** **static** **void** main(String args[]){
15. Testthrows1 obj=**new** Testthrows1();
16. obj.p();
17. System.out.println("normal flow...");
18. }
19. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testthrows1)

Output:

exception handled

normal flow...

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

|  |
| --- |
| There are two cases:   1. **Case1:**You caught the exception i.e. handle the exception using try/catch. 2. **Case2:**You declare the exception i.e. specifying throws with the method. |

### **Case1: You handle the exception**

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

1. **import** java.io.\*;
2. **class** M{
3. **void** method()**throws** IOException{
4. **throw** **new** IOException("device error");
5. }
6. }
7. **public** **class** Testthrows2{
8. **public** **static** **void** main(String args[]){
9. **try**{
10. M m=**new** M();
11. m.method();
12. }**catch**(Exception e){System.out.println("exception handled");}
14. System.out.println("normal flow...");
15. }
16. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testthrows2)

Output:exception handled

normal flow...

### **Case2: You declare the exception**

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

***A)Program if exception does not occur***

1. **import** java.io.\*;
2. **class** M{
3. **void** method()**throws** IOException{
4. System.out.println("device operation performed");
5. }
6. }
7. **class** Testthrows3{
8. **public** **static** **void** main(String args[])**throws** IOException{//declare exception
9. M m=**new** M();
10. m.method();
12. System.out.println("normal flow...");
13. }
14. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testthrows3)

Output:device operation performed

normal flow...

***B)Program if exception occurs***

1. **import** java.io.\*;
2. **class** M{
3. **void** method()**throws** IOException{
4. **throw** **new** IOException("device error");
5. }
6. }
7. **class** Testthrows4{
8. **public** **static** **void** main(String args[])**throws** IOException{//declare exception
9. M m=**new** M();
10. m.method();
12. System.out.println("normal flow...");
13. }
14. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testthrows4)

Output:Runtime Exception

### **Difference between throw and throws**

[Click me for details](https://www.javatpoint.com/difference-between-throw-and-throws-in-java)

### **Que) Can we rethrow an exception?**

Yes, by throwing same exception in catch block.

Difference between throw and throws in Java

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

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

Java throw example

1. **void** m(){
2. **throw** **new** ArithmeticException("sorry");
3. }

Java throws example

1. **void** m()**throws** ArithmeticException{
2. //method code
3. }

Java throw and throws example

1. **void** m()**throws** ArithmeticException{
2. **throw** **new** ArithmeticException("sorry");
3. }

Difference between final, finally and finalize

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

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

Java final example

1. **class** FinalExample{
2. **public** **static** **void** main(String[] args){
3. **final** **int** x=100;
4. x=200;//Compile Time Error
5. }}

Java finally example

1. **class** FinallyExample{
2. **public** **static** **void** main(String[] args){
3. **try**{
4. **int** x=300;
5. }**catch**(Exception e){System.out.println(e);}
6. **finally**{System.out.println("finally block is executed");}
7. }}

Java finalize example

1. **class** FinalizeExample{
2. **public** **void** finalize(){System.out.println("finalize called");}
3. **public** **static** **void** main(String[] args){
4. FinalizeExample f1=**new** FinalizeExample();
5. FinalizeExample f2=**new** FinalizeExample();
6. f1=**null**;
7. f2=**null**;
8. System.gc();
9. }}

# ExceptionHandling with MethodOverriding in Java

|  |
| --- |
| There are many rules if we talk about methodoverriding with exception handling. The Rules are as follows:   * **If the superclass method does not declare an exception**   + If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception. * **If the superclass method declares an exception**   + If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception. |

### **If the superclass method does not declare an exception**

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

1. **import** java.io.\*;
2. **class** Parent{
3. **void** msg(){System.out.println("parent");}
4. }
6. **class** TestExceptionChild **extends** Parent{
7. **void** msg()**throws** IOException{
8. System.out.println("TestExceptionChild");
9. }
10. **public** **static** **void** main(String args[]){
11. Parent p=**new** TestExceptionChild();
12. p.msg();
13. }
14. }

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

Output:Compile Time Error

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

1. **import** java.io.\*;
2. **class** Parent{
3. **void** msg(){System.out.println("parent");}
4. }
6. **class** TestExceptionChild1 **extends** Parent{
7. **void** msg()**throws** ArithmeticException{
8. System.out.println("child");
9. }
10. **public** **static** **void** main(String args[]){
11. Parent p=**new** TestExceptionChild1();
12. p.msg();
13. }
14. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestExceptionChild1)

Output:child

### **If the superclass method declares an exception**

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

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

1. **import** java.io.\*;
2. **class** Parent{
3. **void** msg()**throws** ArithmeticException{System.out.println("parent");}
4. }
6. **class** TestExceptionChild2 **extends** Parent{
7. **void** msg()**throws** Exception{System.out.println("child");}
9. **public** **static** **void** main(String args[]){
10. Parent p=**new** TestExceptionChild2();
11. **try**{
12. p.msg();
13. }**catch**(Exception e){}
14. }
15. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestExceptionChild2)

Output:Compile Time Error

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

1. **import** java.io.\*;
2. **class** Parent{
3. **void** msg()**throws** Exception{System.out.println("parent");}
4. }
6. **class** TestExceptionChild3 **extends** Parent{
7. **void** msg()**throws** Exception{System.out.println("child");}
9. **public** **static** **void** main(String args[]){
10. Parent p=**new** TestExceptionChild3();
11. **try**{
12. p.msg();
13. }**catch**(Exception e){}
14. }
15. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestExceptionChild3)

Output:child

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

1. **import** java.io.\*;
2. **class** Parent{
3. **void** msg()**throws** Exception{System.out.println("parent");}
4. }
6. **class** TestExceptionChild4 **extends** Parent{
7. **void** msg()**throws** ArithmeticException{System.out.println("child");}
9. **public** **static** **void** main(String args[]){
10. Parent p=**new** TestExceptionChild4();
11. **try**{
12. p.msg();
13. }**catch**(Exception e){}
14. }
15. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestExceptionChild4)

Output:child

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

1. **import** java.io.\*;
2. **class** Parent{
3. **void** msg()**throws** Exception{System.out.println("parent");}
4. }
6. **class** TestExceptionChild5 **extends** Parent{
7. **void** msg(){System.out.println("child");}
9. **public** **static** **void** main(String args[]){
10. Parent p=**new** TestExceptionChild5();
11. **try**{
12. p.msg();
13. }**catch**(Exception e){}
14. }
15. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestExceptionChild5)

Output:child

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.

1. **class** InvalidAgeException **extends** Exception{
2. InvalidAgeException(String s){
3. **super**(s);
4. }
5. }
6. **class** TestCustomException1{
8. **static** **void** validate(**int** age)**throws** InvalidAgeException{
9. **if**(age<18)
10. **throw** **new** InvalidAgeException("not valid");
11. **else**
12. System.out.println("welcome to vote");
13. }
15. **public** **static** **void** main(String args[]){
16. **try**{
17. validate(13);
18. }**catch**(Exception m){System.out.println("Exception occured: "+m);}
20. System.out.println("rest of the code...");
21. }
22. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCustomException1)

Output:Exception occured: InvalidAgeException:not valid

rest of the code...

# Java Inner Classes

1. [Java Inner classes](https://www.javatpoint.com/java-inner-class)
2. [Advantage of Inner class](https://www.javatpoint.com/java-inner-class#nestedad)
3. [Difference between nested class and inner class](https://www.javatpoint.com/java-inner-class#nesteddiff)
4. [Types of Nested classes](https://www.javatpoint.com/java-inner-class#nestedtypes)

**Java inner class** or nested class is a class which is declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.

Additionally, it can access all the members of outer class including private data members and methods.

#### Syntax of Inner class

1. **class** Java\_Outer\_class{
2. //code
3. **class** Java\_Inner\_class{
4. //code
5. }
6. }

### **Advantage of java inner classes**

There are basically three advantages of inner classes in java. They are as follows:

1) Nested classes represent a special type of relationship that is **it can access all the members (data members and methods) of outer class** including private.

2) Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.

3) **Code Optimization**: It requires less code to write.

Do You Know

* What is the internal code generated by the compiler for member inner class ?
* What are the two ways to create annonymous inner class ?
* Can we access the non-final local variable inside the local inner class ?
* How to access the static nested class ?
* Can we define an interface within the class ?
* Can we define a class within the interface ?

### **Difference between nested class and inner class in Java**

Inner class is a part of nested class. Non-static nested classes are known as inner classes.

### **Types of Nested classes**

There are two types of nested classes non-static and static nested classes.The non-static nested classes are also known as inner classes.

* Non-static nested class (inner class)
  1. Member inner class
  2. Anonymous inner class
  3. Local inner class
* Static nested class

|  |  |
| --- | --- |
| **Type** | **Description** |
| [Member Inner Class](https://www.javatpoint.com/member-inner-class) | A class created within class and outside method. |
| [Anonymous Inner Class](https://www.javatpoint.com/anonymous-inner-class) | A class created for implementing interface or extending class. Its name is decided by the java compiler. |
| [Local Inner Class](https://www.javatpoint.com/local-inner-class) | A class created within method. |
| [Static Nested Class](https://www.javatpoint.com/static-nested-class) | A static class created within class. |
| [Nested Interface](https://www.javatpoint.com/nested-interface) | An interface created within class or interface. |

Java Member inner class

A non-static class that is created inside a class but outside a method is called member inner class.

Syntax:

1. **class** Outer{
2. //code
3. **class** Inner{
4. //code
5. }
6. }

Java Member inner class example

In this example, we are creating msg() method in member inner class that is accessing the private data member of outer class.

1. **class** TestMemberOuter1{
2. **private** **int** data=30;
3. **class** Inner{
4. **void** msg(){System.out.println("data is "+data);}
5. }
6. **public** **static** **void** main(String args[]){
7. TestMemberOuter1 obj=**new** TestMemberOuter1();
8. TestMemberOuter1.Inner in=obj.**new** Inner();
9. in.msg();
10. }
11. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestMemberOuter1)

Output:

data is 30

Internal working of Java member inner class

The java compiler creates two class files in case of inner class. The class file name of inner class is "Outer$Inner". If you want to instantiate inner class, you must have to create the instance of outer class. In such case, instance of inner class is created inside the instance of outer class.

Internal code generated by the compiler

The java compiler creates a class file named Outer$Inner in this case. The Member inner class have the reference of Outer class that is why it can access all the data members of Outer class including private.

1. **import** java.io.PrintStream;
2. **class** Outer$Inner
3. {
4. **final** Outer **this**$0;
5. Outer$Inner()
6. {   **super**();
7. **this**$0 = Outer.**this**;
8. }
9. **void** msg()
10. {
11. System.out.println((**new** StringBuilder()).append("data is ")
12. .append(Outer.access$000(Outer.**this**)).toString());
13. }
14. }

# Java Anonymous inner class

A class that have no name is known as anonymous inner class in java. It should be used if you have to override method of class or interface. Java Anonymous inner class can be created by two ways:

1. Class (may be abstract or concrete).
2. Interface

### **Java anonymous inner class example using class**

1. **abstract** **class** Person{
2. **abstract** **void** eat();
3. }
4. **class** TestAnonymousInner{
5. **public** **static** **void** main(String args[]){
6. Person p=**new** Person(){
7. **void** eat(){System.out.println("nice fruits");}
8. };
9. p.eat();
10. }
11. }

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

Output:

nice fruits

## Internal working of given code

1. Person p=**new** Person(){
2. **void** eat(){System.out.println("nice fruits");}
3. };
4. A class is created but its name is decided by the compiler which extends the Person class and provides the implementation of the eat() method.
5. An object of Anonymous class is created that is referred by p reference variable of Person type.

## Internal class generated by the compiler

1. **import** java.io.PrintStream;
2. **static** **class** TestAnonymousInner$1 **extends** Person
3. {
4. TestAnonymousInner$1(){}
5. **void** eat()
6. {
7. System.out.println("nice fruits");
8. }
9. }

## Java anonymous inner class example using interface

1. **interface** Eatable{
2. **void** eat();
3. }
4. **class** TestAnnonymousInner1{
5. **public** **static** **void** main(String args[]){
6. Eatable e=**new** Eatable(){
7. **public** **void** eat(){System.out.println("nice fruits");}
8. };
9. e.eat();
10. }
11. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestAnnonymousInner1)

Output:

nice fruits

### **Internal working of given code**

It performs two main tasks behind this code:

1. Eatable p=**new** Eatable(){
2. **void** eat(){System.out.println("nice fruits");}
3. };
4. A class is created but its name is decided by the compiler which implements the Eatable interface and provides the implementation of the eat() method.
5. An object of Anonymous class is created that is referred by p reference variable of Eatable type.

### **Internal class generated by the compiler**

1. **import** java.io.PrintStream;
2. **static** **class** TestAnonymousInner1$1 **implements** Eatable
3. {
4. TestAnonymousInner1$1(){}
5. **void** eat(){System.out.println("nice fruits");}
6. }

# Java Local inner class

A class i.e. created inside a method is called local inner class in java. If you want to invoke the methods of local inner class, you must instantiate this class inside the method.

## Java local inner class example

1. **public** **class** localInner1{
2. **private** **int** data=30;//instance variable
3. **void** display(){
4. **class** Local{
5. **void** msg(){System.out.println(data);}
6. }
7. Local l=**new** Local();
8. l.msg();
9. }
10. **public** **static** **void** main(String args[]){
11. localInner1 obj=**new** localInner1();
12. obj.display();
13. }
14. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=localInner1)

Output:

30

### **Internal class generated by the compiler**

In such case, compiler creates a class named Simple$1Local that have the reference of the outer class.

1. **import** java.io.PrintStream;
2. **class** localInner1$Local
3. {
4. **final** localInner1 **this**$0;
5. localInner1$Local()
6. {
7. **super**();
8. **this**$0 = Simple.**this**;
9. }
10. **void** msg()
11. {
12. System.out.println(localInner1.access$000(localInner1.**this**));
13. }
14. }

#### Rule: Local variable can't be private, public or protected.

## Rules for Java Local Inner class

#### 1) Local inner class cannot be invoked from outside the method.

#### 2) Local inner class cannot access non-final local variable till JDK 1.7. Since JDK 1.8, it is possible to access the non-final local variable in local inner class.

### **Example of local inner class with local variable**

1. **class** localInner2{
2. **private** **int** data=30;//instance variable
3. **void** display(){
4. **int** value=50;//local variable must be final till jdk 1.7 only
5. **class** Local{
6. **void** msg(){System.out.println(value);}
7. }
8. Local l=**new** Local();
9. l.msg();
10. }
11. **public** **static** **void** main(String args[]){
12. localInner2 obj=**new** localInner2();
13. obj.display();
14. }
15. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=localInner2)

Output:

50

# Java static nested class

A static class i.e. created inside a class is called static nested class in java. It cannot access non-static data members and methods. It can be accessed by outer class name.

* It can access static data members of outer class including private.
* Static nested class cannot access non-static (instance) data member or method.

## Java static nested class example with instance method

1. **class** TestOuter1{
2. **static** **int** data=30;
3. **static** **class** Inner{
4. **void** msg(){System.out.println("data is "+data);}
5. }
6. **public** **static** **void** main(String args[]){
7. TestOuter1.Inner obj=**new** TestOuter1.Inner();
8. obj.msg();
9. }
10. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestOuter1)

Output:

data is 30

In this example, you need to create the instance of static nested class because it has instance method msg(). But you don't need to create the object of Outer class because nested class is static and static properties, methods or classes can be accessed without object.

### **Internal class generated by the compiler**

1. **import** java.io.PrintStream;
2. **static** **class** TestOuter1$Inner
3. {
4. TestOuter1$Inner(){}
5. **void** msg(){
6. System.out.println((**new** StringBuilder()).append("data is ")
7. .append(TestOuter1.data).toString());
8. }
9. }

## Java static nested class example with static method

If you have the static member inside static nested class, you don't need to create instance of static nested class.

1. **class** TestOuter2{
2. **static** **int** data=30;
3. **static** **class** Inner{
4. **static** **void** msg(){System.out.println("data is "+data);}
5. }
6. **public** **static** **void** main(String args[]){
7. TestOuter2.Inner.msg();//no need to create the instance of static nested class
8. }
9. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestOuter2)

Output:

data is 30

# Java Nested Interface

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

### **Points to remember for nested interfaces**

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

* Nested interface must be public if it is declared inside the interface but it can have any access modifier if declared within the class.
* Nested interfaces are declared static implicitely.

### **Syntax of nested interface which is declared within the interface**

1. **interface** interface\_name{
2. ...
3. **interface** nested\_interface\_name{
4. ...
5. }
6. }

### **Syntax of nested interface which is declared within the class**

1. **class** class\_name{
2. ...
3. **interface** nested\_interface\_name{
4. ...
5. }
6. }

### **Example of nested interface which is declared within the interface**

|  |
| --- |
| In this example, we are going to learn how to declare the nested interface and how we can access it. |

1. **interface** Showable{
2. **void** show();
3. **interface** Message{
4. **void** msg();
5. }
6. }
7. **class** TestNestedInterface1 **implements** Showable.Message{
8. **public** **void** msg(){System.out.println("Hello nested interface");}
10. **public** **static** **void** main(String args[]){
11. Showable.Message message=**new** TestNestedInterface1();//upcasting here
12. message.msg();
13. }
14. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestNestedInterface1)

[download the example of nested interface](https://www.javatpoint.com/src/nested/nestedinterface.zip)

Output:hello nested interface

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

### **Internal code generated by the java compiler for nested interface Message**

|  |
| --- |
| The java compiler internally creates public and static interface as displayed below:. |

1. **public** **static** **interface** Showable$Message
2. {
3. **public** **abstract** **void** msg();
4. }

### **Example of nested interface which is declared within the class**

|  |
| --- |
| Let's see how can we define an interface inside the class and how can we access it. |

1. **class** A{
2. **interface** Message{
3. **void** msg();
4. }
5. }
7. **class** TestNestedInterface2 **implements** A.Message{
8. **public** **void** msg(){System.out.println("Hello nested interface");}
10. **public** **static** **void** main(String args[]){
11. A.Message message=**new** TestNestedInterface2();//upcasting here
12. message.msg();
13. }
14. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestNestedInterface2)

Output:hello nested interface

### **Can we define a class inside the interface?**

Yes, If we define a class inside the interface, java compiler creates a static nested class. Let's see how can we define a class within the interface:

1. **interface** M{
2. **class** A{}
3. }

# Collections in Java

1. [Java Collection Framework](https://www.javatpoint.com/collections-in-java)
2. [Hierarchy of Collection Framework](https://www.javatpoint.com/collections-in-java#collectionhierarchy)
3. [Collection interface](https://www.javatpoint.com/collections-in-java#collectionmethods)
4. [Iterator interface](https://www.javatpoint.com/collections-in-java#collectioniterator)

**Collections in java** is a framework that provides an architecture to store and manipulate the group of objects.

All the operations that you perform on a data such as searching, sorting, insertion, manipulation, deletion etc. can be performed by Java Collections.

Java Collection simply means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque etc.) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet etc).

#### What is Collection in java

Collection represents a single unit of objects i.e. a group.

#### What is framework in java

* provides readymade architecture.
* represents set of classes and interface.
* is optional.

#### What is Collection framework

Collection framework represents a unified architecture for storing and manipulating group of objects. It has:

1. Interfaces and its implementations i.e. classes
2. Algorithm

Do You Know ?

* What are the two ways to iterate the elements of a collection ?
* What is the difference between ArrayList and LinkedList classes in collection framework?
* What is the difference between ArrayList and Vector classes in collection framework?
* What is the difference between HashSet and HashMap classes in collection framework?
* What is the difference between HashMap and Hashtable class?
* What is the difference between Iterator and Enumeration interface in collection framework?
* How can we sort the elements of an object. What is the difference between Comparable and Comparator interfaces?
* What does the hashcode() method ?
* What is the difference between java collection and java collections ?

### **Hierarchy of Collection Framework**

Let us see the hierarchy of collection framework.The **java.util** package contains all the classes and interfaces for Collection framework.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | public boolean add(Object element) | is used to insert an element in this collection. |
| 2 | public boolean addAll(Collection c) | is used to insert the specified collection elements in the invoking collection. |
| 3 | public boolean remove(Object element) | is used to delete an element from this collection. |
| 4 | public boolean removeAll(Collection c) | is used to delete all the elements of specified collection from the invoking collection. |
| 5 | public boolean retainAll(Collection c) | is used to delete all the elements of invoking collection except the specified collection. |
| 6 | public int size() | return the total number of elements in the collection. |
| 7 | public void clear() | removes the total no of element from the collection. |
| 8 | public boolean contains(Object element) | is used to search an element. |
| 9 | public boolean containsAll(Collection c) | is used to search the specified collection in this collection. |
| 10 | public Iterator iterator() | returns an iterator. |
| 11 | public Object[] toArray() | converts collection into array. |
| 12 | public boolean isEmpty() | checks if collection is empty. |
| 13 | public boolean equals(Object element) | matches two collection. |
| 14 | public int hashCode() | returns the hashcode number for collection. |

hierarchy of collection framework

### **Methods of Collection interface**

There are many methods declared in the Collection interface. They are as follows:

### **Iterator interface**

|  |
| --- |
| Iterator interface provides the facility of iterating the elements in forward direction only. |

#### Methods of Iterator interface

There are only three methods in the Iterator interface. They are:

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | public boolean hasNext() | It returns true if iterator has more elements. |
| 2 | public Object next() | It returns the element and moves the cursor pointer to the next element. |
| 3 | public void remove() | It removes the last elements returned by the iterator. It is rarely used. |

What we are going to learn in Java Collections Framework

| **Interface** | **Hash Table** | **Resizable Array** | **Balanced Tree** | **Linked List** | **Hash Table + Linked List** |
| --- | --- | --- | --- | --- | --- |
| Set | [HashSet](https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html) |  | [TreeSet](https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html) |  | [LinkedHashSet](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html) |
| List |  | [ArrayList](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html) |  | [LinkedList](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html) |  |
| Deque |  | [ArrayDeque](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayDeque.html) |  | [LinkedList](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html) |  |
| Map | [HashMap](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) |  | [TreeMap](https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html) |  | [LinkedHashMap](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html) |

1. [ArrayList class](https://www.javatpoint.com/java-arraylist)
2. [LinkedList class](https://www.javatpoint.com/java-linkedlist)
3. [List interface](https://www.javatpoint.com/java-list)
4. [HashSet class](https://www.javatpoint.com/java-hashset)
5. [LinkedHashSet class](https://www.javatpoint.com/java-linkedhashset)
6. [TreeSet class](https://www.javatpoint.com/java-treeset)
7. [PriorityQueue class](https://www.javatpoint.com/java-priorityqueue)
8. [Map interface](https://www.javatpoint.com/java-map)
9. [HashMap class](https://www.javatpoint.com/java-hashmap)
10. [LinkedHashMap class](https://www.javatpoint.com/java-linkedhashmap)
11. [TreeMap class](https://www.javatpoint.com/TreeMap-class-in-collection-framework)
12. [Hashtable class](https://www.javatpoint.com/Hashtable-class-in-collection-framework)
13. [Sorting](https://www.javatpoint.com/Sorting-in-collection-framework)
14. [Comparable interface](https://www.javatpoint.com/Comparable-interface-in-collection-framework)
15. [Comparator interface](https://www.javatpoint.com/Comparator-interface-in-collection-framework)
16. [Properties class in Java](https://www.javatpoint.com/properties-class-in-java)

# Java ArrayList class

Java ArrayList class hierarchy

Java ArrayList class uses a dynamic array for storing the elements. It inherits AbstractList class and implements List interface.

The important points about Java ArrayList class are:

* Java ArrayList class can contain duplicate elements.
* Java ArrayList class maintains insertion order.
* Java ArrayList class is non synchronized.
* Java ArrayList allows random access because array works at the index basis.
* In Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

### **Hierarchy of ArrayList class**

As shown in above diagram, Java ArrayList class extends AbstractList class which implements List interface. The List interface extends Collection and Iterable interfaces in hierarchical order.

### **ArrayList class declaration**

Let's see the declaration for java.util.ArrayList class.

1. **public** **class** ArrayList<E> **extends** AbstractList<E> **implements** List<E>, RandomAccess, Cloneable, Serializable

### **Constructors of Java ArrayList**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| ArrayList() | It is used to build an empty array list. |
| ArrayList(Collection c) | It is used to build an array list that is initialized with the elements of the collection c. |
| ArrayList(int capacity) | It is used to build an array list that has the specified initial capacity. |

### **Methods of Java ArrayList**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void add(int index, Object element) | It is used to insert the specified element at the specified position index in a list. |
| boolean addAll(Collection c) | It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. |
| void clear() | It is used to remove all of the elements from this list. |
| int lastIndexOf(Object o) | It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element. |
| Object[] toArray() | It is used to return an array containing all of the elements in this list in the correct order. |
| Object[] toArray(Object[] a) | It is used to return an array containing all of the elements in this list in the correct order. |
| boolean add(Object o) | It is used to append the specified element to the end of a list. |
| boolean addAll(int index, Collection c) | It is used to insert all of the elements in the specified collection into this list, starting at the specified position. |
| Object clone() | It is used to return a shallow copy of an ArrayList. |
| int indexOf(Object o) | It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element. |
| void trimToSize() | It is used to trim the capacity of this ArrayList instance to be the list's current size. |

### **Java Non-generic Vs Generic Collection**

Java collection framework was non-generic before JDK 1.5. Since 1.5, it is generic.

Java new generic collection allows you to have only one type of object in collection. Now it is type safe so typecasting is not required at run time.

Let's see the old non-generic example of creating java collection.

1. ArrayList al=**new** ArrayList();//creating old non-generic arraylist

Let's see the new generic example of creating java collection.

1. ArrayList<String> al=**new** ArrayList<String>();//creating new generic arraylist

In generic collection, we specify the type in angular braces. Now ArrayList is forced to have only specified type of objects in it. If you try to add another type of object, it gives *compile time error*.

For more information of java generics, click here [Java Generics Tutorial](https://www.javatpoint.com/generics-in-java).

### **Java ArrayList Example**

1. **import** java.util.\*;
2. **class** TestCollection1{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> list=**new** ArrayList<String>();//Creating arraylist
5. list.add("Ravi");//Adding object in arraylist
6. list.add("Vijay");
7. list.add("Ravi");
8. list.add("Ajay");
9. //Traversing list through Iterator
10. Iterator itr=list.iterator();
11. **while**(itr.hasNext()){
12. System.out.println(itr.next());
13. }
14. }
15. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection1)

Ravi

Vijay

Ravi

Ajay

### **Two ways to iterate the elements of collection in java**

There are two ways to traverse collection elements:

1. By Iterator interface.
2. By for-each loop.

In the above example, we have seen traversing ArrayList by Iterator. Let's see the example to traverse ArrayList elements using for-each loop.

### **Iterating Collection through for-each loop**

1. **import** java.util.\*;
2. **class** TestCollection2{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> al=**new** ArrayList<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ravi");
8. al.add("Ajay");
9. **for**(String obj:al)
10. System.out.println(obj);
11. }
12. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection2)

Ravi

Vijay

Ravi

Ajay

### **User-defined class objects in Java ArrayList**

Let's see an example where we are storing Student class object in array list.

1. **class** Student{
2. **int** rollno;
3. String name;
4. **int** age;
5. Student(**int** rollno,String name,**int** age){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.age=age;
9. }
10. }
11. **import** java.util.\*;
12. **public** **class** TestCollection3{
13. **public** **static** **void** main(String args[]){
14. //Creating user-defined class objects
15. Student s1=**new** Student(101,"Sonoo",23);
16. Student s2=**new** Student(102,"Ravi",21);
17. Student s2=**new** Student(103,"Hanumat",25);
18. //creating arraylist
19. ArrayList<Student> al=**new** ArrayList<Student>();
20. al.add(s1);//adding Student class object
21. al.add(s2);
22. al.add(s3);
23. //Getting Iterator
24. Iterator itr=al.iterator();
25. //traversing elements of ArrayList object
26. **while**(itr.hasNext()){
27. Student st=(Student)itr.next();
28. System.out.println(st.rollno+" "+st.name+" "+st.age);
29. }
30. }
31. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection3)

101 Sonoo 23

102 Ravi 21

103 Hanumat 25

### **Example of addAll(Collection c) method**

1. **import** java.util.\*;
2. **class** TestCollection4{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> al=**new** ArrayList<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ajay");
8. ArrayList<String> al2=**new** ArrayList<String>();
9. al2.add("Sonoo");
10. al2.add("Hanumat");
11. al.addAll(al2);//adding second list in first list
12. Iterator itr=al.iterator();
13. **while**(itr.hasNext()){
14. System.out.println(itr.next());
15. }
16. }
17. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection4)

Ravi

Vijay

Ajay

Sonoo

Hanumat

### **Example of removeAll() method**

1. **import** java.util.\*;
2. **class** TestCollection5{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> al=**new** ArrayList<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ajay");
8. ArrayList<String> al2=**new** ArrayList<String>();
9. al2.add("Ravi");
10. al2.add("Hanumat");
11. al.removeAll(al2);
12. System.out.println("iterating the elements after removing the elements of al2...");
13. Iterator itr=al.iterator();
14. **while**(itr.hasNext()){
15. System.out.println(itr.next());
16. }
18. }
19. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection5)

iterating the elements after removing the elements of al2...

Vijay

Ajay

### **Example of retainAll() method**

1. **import** java.util.\*;
2. **class** TestCollection6{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> al=**new** ArrayList<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ajay");
8. ArrayList<String> al2=**new** ArrayList<String>();
9. al2.add("Ravi");
10. al2.add("Hanumat");
11. al.retainAll(al2);
12. System.out.println("iterating the elements after retaining the elements of al2...");
13. Iterator itr=al.iterator();
14. **while**(itr.hasNext()){
15. System.out.println(itr.next());
16. }
17. }
18. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection6)

iterating the elements after retaining the elements of al2...

Ravi

### **Java ArrayList Example: Book**

Let's see an ArrayList example where we are adding books to list and printing all the books.

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** ArrayListExample {
15. **public** **static** **void** main(String[] args) {
16. //Creating list of Books
17. List<Book> list=**new** ArrayList<Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to list
23. list.add(b1);
24. list.add(b2);
25. list.add(b3);
26. //Traversing list
27. **for**(Book b:list){
28. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
29. }
30. }
31. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection101)

Output:

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications & Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

# Java LinkedList class

Java LinkedList class hierarchy

Java LinkedList class uses doubly linked list to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.

The important points about Java LinkedList are:

* Java LinkedList class can contain duplicate elements.
* Java LinkedList class maintains insertion order.
* Java LinkedList class is non synchronized.
* In Java LinkedList class, manipulation is fast because no shifting needs to be occurred.
* Java LinkedList class can be used as list, stack or queue.

### **Hierarchy of LinkedList class**

As shown in above diagram, Java LinkedList class extends AbstractSequentialList class and implements List and Deque interfaces.

### **Doubly Linked List**

In case of doubly linked list, we can add or remove elements from both side.

java LinkedList class using doubly linked list

### **LinkedList class declaration**

Let's see the declaration for java.util.LinkedList class.

1. **public** **class** LinkedList<E> **extends** AbstractSequentialList<E> **implements** List<E>, Deque<E>, Cloneable, Serializable

### **Constructors of Java LinkedList**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| LinkedList() | It is used to construct an empty list. |
| LinkedList(Collection c) | It is used to construct a list containing the elements of the specified collection, in the order they are returned by the collection's iterator. |

### **Methods of Java LinkedList**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void add(int index, Object element) | It is used to insert the specified element at the specified position index in a list. |
| void addFirst(Object o) | It is used to insert the given element at the beginning of a list. |
| void addLast(Object o) | It is used to append the given element to the end of a list. |
| int size() | It is used to return the number of elements in a list |
| boolean add(Object o) | It is used to append the specified element to the end of a list. |
| boolean contains(Object o) | It is used to return true if the list contains a specified element. |
| boolean remove(Object o) | It is used to remove the first occurence of the specified element in a list. |
| Object getFirst() | It is used to return the first element in a list. |
| Object getLast() | It is used to return the last element in a list. |
| int indexOf(Object o) | It is used to return the index in a list of the first occurrence of the specified element, or -1 if the list does not contain any element. |
| int lastIndexOf(Object o) | It is used to return the index in a list of the last occurrence of the specified element, or -1 if the list does not contain any element. |

### **Java LinkedList Example**

1. **import** java.util.\*;
2. **public** **class** TestCollection7{
3. **public** **static** **void** main(String args[]){
5. LinkedList<String> al=**new** LinkedList<String>();
6. al.add("Ravi");
7. al.add("Vijay");
8. al.add("Ravi");
9. al.add("Ajay");
11. Iterator<String> itr=al.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection7)

Output:Ravi

Vijay

Ravi

Ajay

### **Java LinkedList Example: Book**

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** LinkedListExample {
15. **public** **static** **void** main(String[] args) {
16. //Creating list of Books
17. List<Book> list=**new** LinkedList<Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to list
23. list.add(b1);
24. list.add(b2);
25. list.add(b3);
26. //Traversing list
27. **for**(Book b:list){
28. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
29. }
30. }
31. }

Output:

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications & Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

# Difference between ArrayList and LinkedList

ArrayList and LinkedList both implements List interface and maintains insertion order. Both are non synchronized classes.

But there are many differences between ArrayList and LinkedList classes that are given below.

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| 1) ArrayList internally uses **dynamic array** to store the elements. | LinkedList internally uses **doubly linked list** to store the elements. |
| 2) Manipulation with ArrayList is **slow** because it internally uses array. If any element is removed from the array, all the bits are shifted in memory. | Manipulation with LinkedList is **faster** than ArrayList because it uses doubly linked list so no bit shifting is required in memory. |
| 3) ArrayList class can **act as a list** only because it implements List only. | LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces. |
| 4) ArrayList is **better for storing and accessing** data. | LinkedList is **better for manipulating** data. |

### **Example of ArrayList and LinkedList in Java**

Let's see a simple example where we are using ArrayList and LinkedList both.

1. **import** java.util.\*;
2. **class** TestArrayLinked{
3. **public** **static** **void** main(String args[]){
5. List<String> al=**new** ArrayList<String>();//creating arraylist
6. al.add("Ravi");//adding object in arraylist
7. al.add("Vijay");
8. al.add("Ravi");
9. al.add("Ajay");
11. List<String> al2=**new** LinkedList<String>();//creating linkedlist
12. al2.add("James");//adding object in linkedlist
13. al2.add("Serena");
14. al2.add("Swati");
15. al2.add("Junaid");
17. System.out.println("arraylist: "+al);
18. System.out.println("linkedlist: "+al2);
19. }
20. }

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

Output:

arraylist: [Ravi,Vijay,Ravi,Ajay]

linkedlist: [James,Serena,Swati,Junaid]

# Java List Interface

List Interface is the subinterface of Collection.It contains methods to insert and delete elements in index basis.It is a factory of ListIterator interface.

### **List Interface declaration**

1. **public** **interface** List<E> **extends** Collection<E>

### **Methods of Java List Interface**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void add(int index,Object element) | It is used to insert element into the invoking list at the index passed in the index. |
| boolean addAll(int index,Collection c) | It is used to insert all elements of c into the invoking list at the index passed in the index. |
| object get(int index) | It is used to return the object stored at the specified index within the invoking collection. |
| object set(int index,Object element) | It is used to assign element to the location specified by index within the invoking list. |
| object remove(int index) | It is used to remove the element at position index from the invoking list and return the deleted element. |
| ListIterator listIterator() | It is used to return an iterator to the start of the invoking list. |
| ListIterator listIterator(int index) | It is used to return an iterator to the invoking list that begins at the specified index. |

### **Java List Example**

1. **import** java.util.\*;
2. **public** **class** ListExample{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> al=**new** ArrayList<String>();
5. al.add("Amit");
6. al.add("Vijay");
7. al.add("Kumar");
8. al.add(1,"Sachin");
9. System.out.println("Element at 2nd position: "+al.get(2));
10. **for**(String s:al){
11. System.out.println(s);
12. }
13. }
14. }

Output:

Element at 2nd position: Vijay

Amit

Sachin

Vijay

Kumar

## Java ListIterator Interface

ListIterator Interface is used to traverse the element in backward and forward direction.

### **ListIterator Interface declaration**

1. **public** **interface** ListIterator<E> **extends** Iterator<E>

### **Methods of Java ListIterator Interface:**

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean hasNext() | This method return true if the list iterator has more elements when traversing the list in the forward direction. |
| Object next() | This method return the next element in the list and advances the cursor position. |
| boolean hasPrevious() | This method return true if this list iterator has more elements when traversing the list in the reverse direction. |
| Object previous() | This method return the previous element in the list and moves the cursor position backwards. |

### **Example of ListIterator Interface**

1. **import** java.util.\*;
2. **public** **class** TestCollection8{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> al=**new** ArrayList<String>();
5. al.add("Amit");
6. al.add("Vijay");
7. al.add("Kumar");
8. al.add(1,"Sachin");
9. System.out.println("element at 2nd position: "+al.get(2));
10. ListIterator<String> itr=al.listIterator();
11. System.out.println("traversing elements in forward direction...");
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. System.out.println("traversing elements in backward direction...");
16. **while**(itr.hasPrevious()){
17. System.out.println(itr.previous());
18. }
19. }
20. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection8)

Output:

element at 2nd position: Vijay

traversing elements in forward direction...

Amit

Sachin

Vijay

Kumar

traversing elements in backward direction...

Kumar

Vijay

Sachin

Amit

### **Example of ListIterator Interface: Book**

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** ListExample {
15. **public** **static** **void** main(String[] args) {
16. //Creating list of Books
17. List<Book> list=**new** ArrayList<Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to list
23. list.add(b1);
24. list.add(b2);
25. list.add(b3);
26. //Traversing list
27. **for**(Book b:list){
28. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
29. }
30. }
31. }

Output:

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications & Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

# Java HashSet class

Java HashSet class hierarchy

Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

The important points about Java HashSet class are:

* HashSet stores the elements by using a mechanism called **hashing.**
* HashSet contains unique elements only.

## Difference between List and Set

List can contain duplicate elements whereas Set contains unique elements only.

### **Hierarchy of HashSet class**

The HashSet class extends AbstractSet class which implements Set interface. The Set interface inherits Collection and Iterable interfaces in hierarchical order.

### **HashSet class declaration**

Let's see the declaration for java.util.HashSet class.

1. **public** **class** HashSet<E> **extends** AbstractSet<E> **implements** Set<E>, Cloneable, Serializable

### **Constructors of Java HashSet class:**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| HashSet() | It is used to construct a default HashSet. |
| HashSet(Collection c) | It is used to initialize the hash set by using the elements of the collection c. |
| HashSet(int capacity) | It is used to initialize the capacity of the hash set to the given integer value capacity. The capacity grows automatically as elements are added to the HashSet. |

### **Methods of Java HashSet class:**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void clear() | It is used to remove all of the elements from this set. |
| boolean contains(Object o) | It is used to return true if this set contains the specified element. |
| boolean add(Object o) | It is used to adds the specified element to this set if it is not already present. |
| boolean isEmpty() | It is used to return true if this set contains no elements. |
| boolean remove(Object o) | It is used to remove the specified element from this set if it is present. |
| Object clone() | It is used to return a shallow copy of this HashSet instance: the elements themselves are not cloned. |
| Iterator iterator() | It is used to return an iterator over the elements in this set. |
| int size() | It is used to return the number of elements in this set. |

### **Java HashSet Example**

1. **import** java.util.\*;
2. **class** TestCollection9{
3. **public** **static** **void** main(String args[]){
4. //Creating HashSet and adding elements
5. HashSet<String> set=**new** HashSet<String>();
6. set.add("Ravi");
7. set.add("Vijay");
8. set.add("Ravi");
9. set.add("Ajay");
10. //Traversing elements
11. Iterator<String> itr=set.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection9)

Ajay

Vijay

Ravi

### **Java HashSet Example: Book**

Let's see a HashSet example where we are adding books to set and printing all the books.

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** HashSetExample {
15. **public** **static** **void** main(String[] args) {
16. HashSet<Book> set=**new** HashSet<Book>();
17. //Creating Books
18. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
19. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
20. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
21. //Adding Books to HashSet
22. set.add(b1);
23. set.add(b2);
24. set.add(b3);
25. //Traversing HashSet
26. **for**(Book b:set){
27. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
28. }
29. }
30. }

Output:

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications & Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

# Java LinkedHashSet class

Java HashSet class hierarchy

Java LinkedHashSet class is a Hash table and Linked list implementation of the set interface. It inherits HashSet class and implements Set interface.

The important points about Java LinkedHashSet class are:

* Contains unique elements only like HashSet.
* Provides all optional set operations, and permits null elements.
* Maintains insertion order.

## Hierarchy of LinkedHashSet class

The LinkedHashSet class extends HashSet class which implements Set interface. The Set interface inherits Collection and Iterable interfaces in hierarchical order.

### **LinkedHashSet class declaration**

Let's see the declaration for java.util.LinkedHashSet class.

1. **public** **class** LinkedHashSet<E> **extends** HashSet<E> **implements** Set<E>, Cloneable, Serializable

### **Constructors of Java LinkedHashSet class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| HashSet() | It is used to construct a default HashSet. |
| HashSet(Collection c) | It is used to initialize the hash set by using the elements of the collection c. |
| LinkedHashSet(int capacity) | It is used initialize the capacity of the linkedhashset to the given integer value capacity. |
| LinkedHashSet(int capacity, float fillRatio) | It is used to initialize both the capacity and the fill ratio (also called load capacity) of the hash set from its argument. |

### **Example of LinkedHashSet class:**

1. **import** java.util.\*;
2. **class** TestCollection10{
3. **public** **static** **void** main(String args[]){
4. LinkedHashSet<String> al=**new** LinkedHashSet<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ravi");
8. al.add("Ajay");
9. Iterator<String> itr=al.iterator();
10. **while**(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection10)

Ravi

Vijay

Ajay

### **Java LinkedHashSet Example: Book**

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** LinkedHashSetExample {
15. **public** **static** **void** main(String[] args) {
16. LinkedHashSet<Book> hs=**new** LinkedHashSet<Book>();
17. //Creating Books
18. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
19. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
20. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
21. //Adding Books to hash table
22. hs.add(b1);
23. hs.add(b2);
24. hs.add(b3);
25. //Traversing hash table
26. **for**(Book b:hs){
27. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
28. }
29. }
30. }

Output:

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications & Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

# Java TreeSet class

TreeSet class hierarchy

Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements NavigableSet interface. The objects of TreeSet class are stored in ascending order.

The important points about Java TreeSet class are:

* Contains unique elements only like HashSet.
* Access and retrieval times are quiet fast.
* Maintains ascending order.

### **Hierarchy of TreeSet class**

As shown in above diagram, Java TreeSet class implements NavigableSet interface. The NavigableSet interface extends SortedSet, Set, Collection and Iterable interfaces in hierarchical order.

### **TreeSet class declaration**

Let's see the declaration for java.util.TreeSet class.

1. **public** **class** TreeSet<E> **extends** AbstractSet<E> **implements** NavigableSet<E>, Cloneable, Serializable

### **Constructors of Java TreeSet class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| TreeSet() | It is used to construct an empty tree set that will be sorted in an ascending order according to the natural order of the tree set. |
| TreeSet(Collection c) | It is used to build a new tree set that contains the elements of the collection c. |
| TreeSet(Comparator comp) | It is used to construct an empty tree set that will be sorted according to given comparator. |
| TreeSet(SortedSet ss) | It is used to build a TreeSet that contains the elements of the given SortedSet. |

### **Methods of Java TreeSet class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean addAll(Collection c) | It is used to add all of the elements in the specified collection to this set. |
| boolean contains(Object o) | It is used to return true if this set contains the specified element. |
| boolean isEmpty() | It is used to return true if this set contains no elements. |
| boolean remove(Object o) | It is used to remove the specified element from this set if it is present. |
| void add(Object o) | It is used to add the specified element to this set if it is not already present. |
| void clear() | It is used to remove all of the elements from this set. |
| Object clone() | It is used to return a shallow copy of this TreeSet instance. |
| Object first() | It is used to return the first (lowest) element currently in this sorted set. |
| Object last() | It is used to return the last (highest) element currently in this sorted set. |
| int size() | It is used to return the number of elements in this set. |

### **Java TreeSet Example**

1. **import** java.util.\*;
2. **class** TestCollection11{
3. **public** **static** **void** main(String args[]){
4. //Creating and adding elements
5. TreeSet<String> al=**new** TreeSet<String>();
6. al.add("Ravi");
7. al.add("Vijay");
8. al.add("Ravi");
9. al.add("Ajay");
10. //Traversing elements
11. Iterator<String> itr=al.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection11)

Output:

Ajay

Ravi

Vijay

### **Java TreeSet Example: Book**

Let's see a TreeSet example where we are adding books to set and printing all the books. The elements in TreeSet must be of Comparable type. String and Wrapper classes are Comparable by default. To add user-defined objects in TreeSet, you need to implement Comparable interface.

1. **import** java.util.\*;
2. **class** Book **implements** Comparable<Book>{
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. **public** **int** compareTo(Book b) {
14. **if**(id>b.id){
15. **return** 1;
16. }**else** **if**(id<b.id){
17. **return** -1;
18. }**else**{
19. **return** 0;
20. }
21. }
22. }
23. **public** **class** TreeSetExample {
24. **public** **static** **void** main(String[] args) {
25. Set<Book> set=**new** TreeSet<Book>();
26. //Creating Books
27. Book b1=**new** Book(121,"Let us C","Yashwant Kanetkar","BPB",8);
28. Book b2=**new** Book(233,"Operating System","Galvin","Wiley",6);
29. Book b3=**new** Book(101,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
30. //Adding Books to TreeSet
31. set.add(b1);
32. set.add(b2);
33. set.add(b3);
34. //Traversing TreeSet
35. **for**(Book b:set){
36. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
37. }
38. }
39. }

Output:

101 Data Communications & Networking Forouzan Mc Graw Hill 4

121 Let us C Yashwant Kanetkar BPB 8

233 Operating System Galvin Wiley 6

# Java Queue Interface

Java Queue interface orders the element in FIFO(First In First Out) manner. In FIFO, first element is removed first and last element is removed at last.

### **Queue Interface declaration**

1. **public** **interface** Queue<E> **extends** Collection<E>

### **Methods of Java Queue Interface**

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean add(object) | It is used to insert the specified element into this queue and return true upon success. |
| boolean offer(object) | It is used to insert the specified element into this queue. |
| Object remove() | It is used to retrieves and removes the head of this queue. |
| Object poll() | It is used to retrieves and removes the head of this queue, or returns null if this queue is empty. |
| Object element() | It is used to retrieves, but does not remove, the head of this queue. |
| Object peek() | It is used to retrieves, but does not remove, the head of this queue, or returns null if this queue is empty. |

## PriorityQueue class

The PriorityQueue class provides the facility of using queue. But it does not orders the elements in FIFO manner. It inherits AbstractQueue class.

### **PriorityQueue class declaration**

Let's see the declaration for java.util.PriorityQueue class.

1. **public** **class** PriorityQueue<E> **extends** AbstractQueue<E> **implements** Serializable

### **Java PriorityQueue Example**

1. **import** java.util.\*;
2. **class** TestCollection12{
3. **public** **static** **void** main(String args[]){
4. PriorityQueue<String> queue=**new** PriorityQueue<String>();
5. queue.add("Amit");
6. queue.add("Vijay");
7. queue.add("Karan");
8. queue.add("Jai");
9. queue.add("Rahul");
10. System.out.println("head:"+queue.element());
11. System.out.println("head:"+queue.peek());
12. System.out.println("iterating the queue elements:");
13. Iterator itr=queue.iterator();
14. **while**(itr.hasNext()){
15. System.out.println(itr.next());
16. }
17. queue.remove();
18. queue.poll();
19. System.out.println("after removing two elements:");
20. Iterator<String> itr2=queue.iterator();
21. **while**(itr2.hasNext()){
22. System.out.println(itr2.next());
23. }
24. }
25. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection12)

Output:head:Amit

head:Amit

iterating the queue elements:

Amit

Jai

Karan

Vijay

Rahul

after removing two elements:

Karan

Rahul

Vijay

### **Java PriorityQueue Example: Book**

Let's see a PriorityQueue example where we are adding books to queue and printing all the books. The elements in PriorityQueue must be of Comparable type. String and Wrapper classes are Comparable by default. To add user-defined objects in PriorityQueue, you need to implement Comparable interface.

1. **import** java.util.\*;
2. **class** Book **implements** Comparable<Book>{
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. **public** **int** compareTo(Book b) {
14. **if**(id>b.id){
15. **return** 1;
16. }**else** **if**(id<b.id){
17. **return** -1;
18. }**else**{
19. **return** 0;
20. }
21. }
22. }
23. **public** **class** LinkedListExample {
24. **public** **static** **void** main(String[] args) {
25. Queue<Book> queue=**new** PriorityQueue<Book>();
26. //Creating Books
27. Book b1=**new** Book(121,"Let us C","Yashwant Kanetkar","BPB",8);
28. Book b2=**new** Book(233,"Operating System","Galvin","Wiley",6);
29. Book b3=**new** Book(101,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
30. //Adding Books to the queue
31. queue.add(b1);
32. queue.add(b2);
33. queue.add(b3);
34. System.out.println("Traversing the queue elements:");
35. //Traversing queue elements
36. **for**(Book b:queue){
37. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
38. }
39. queue.remove();
40. System.out.println("After removing one book record:");
41. **for**(Book b:queue){
42. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
43. }
44. }
45. }

Output:

Traversing the queue elements:

101 Data Communications & Networking Forouzan Mc Graw Hill 4

233 Operating System Galvin Wiley 6

121 Let us C Yashwant Kanetkar BPB 8

After removing one book record:

121 Let us C Yashwant Kanetkar BPB 8

233 Operating System Galvin Wiley 6

# Java Deque Interface

Java Deque Interface is a linear collection that supports element insertion and removal at both ends. Deque is an acronym for **"double ended queue".**

## Deque Interface declaration

1. **public** **interface** Deque<E> **extends** Queue<E>

### **Methods of Java Deque Interface**

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean add(object) | It is used to insert the specified element into this deque and return true upon success. |
| boolean offer(object) | It is used to insert the specified element into this deque. |
| Object remove() | It is used to retrieves and removes the head of this deque. |
| Object poll() | It is used to retrieves and removes the head of this deque, or returns null if this deque is empty. |
| Object element() | It is used to retrieves, but does not remove, the head of this deque. |
| Object peek() | It is used to retrieves, but does not remove, the head of this deque, or returns null if this deque is empty. |

java arraydeque hierarchy

## ArrayDeque class

The ArrayDeque class provides the facility of using deque and resizable-array. It inherits AbstractCollection class and implements the Deque interface.

The important points about ArrayDeque class are:

* Unlike Queue, we can add or remove elements from both sides.
* Null elements are not allowed in the ArrayDeque.
* ArrayDeque is not thread safe, in the absence of external synchronization.
* ArrayDeque has no capacity restrictions.
* ArrayDeque is faster than LinkedList and Stack.

### **ArrayDeque Hierarchy**

The hierarchy of ArrayDeque class is given in the figure displayed at the right side of the page.

### **ArrayDeque class declaration**

Let's see the declaration for java.util.ArrayDeque class.

1. **public** **class** ArrayDeque<E> **extends** AbstractCollection<E> **implements** Deque<E>, Cloneable, Serializable

## Java ArrayDeque Example

1. **import** java.util.\*;
2. **public** **class** ArrayDequeExample {
3. **public** **static** **void** main(String[] args) {
4. //Creating Deque and adding elements
5. Deque<String> deque = **new** ArrayDeque<String>();
6. deque.add("Ravi");
7. deque.add("Vijay");
8. deque.add("Ajay");
9. //Traversing elements
10. **for** (String str : deque) {
11. System.out.println(str);
12. }
13. }
14. }

Output:

Ravi

Vijay

Ajay

## Java ArrayDeque Example: offerFirst() and pollLast()

1. **import** java.util.\*;
2. **public** **class** DequeExample {
3. **public** **static** **void** main(String[] args) {
4. Deque<String> deque=**new** ArrayDeque<String>();
5. deque.offer("arvind");
6. deque.offer("vimal");
7. deque.add("mukul");
8. deque.offerFirst("jai");
9. System.out.println("After offerFirst Traversal...");
10. **for**(String s:deque){
11. System.out.println(s);
12. }
13. //deque.poll();
14. //deque.pollFirst();//it is same as poll()
15. deque.pollLast();
16. System.out.println("After pollLast() Traversal...");
17. **for**(String s:deque){
18. System.out.println(s);
19. }
20. }
21. }

Output:

After offerFirst Traversal...

jai

arvind

vimal

mukul

After pollLast() Traversal...

jai

arvind

vimal

## Java ArrayDeque Example: Book

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** ArrayDequeExample {
15. **public** **static** **void** main(String[] args) {
16. Deque<Book> set=**new** ArrayDeque<Book>();
17. //Creating Books
18. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
19. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
20. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
21. //Adding Books to Deque
22. set.add(b1);
23. set.add(b2);
24. set.add(b3);
25. //Traversing ArrayDeque
26. **for**(Book b:set){
27. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
28. }
29. }
30. }

Output:

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications & Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

# Java Map Interface

A map contains values on the basis of key i.e. key and value pair. Each key and value pair is known as an entry. Map contains only unique keys.

Map is useful if you have to search, update or delete elements on the basis of key.

### **Useful methods of Map interface**

|  |  |
| --- | --- |
| **Method** | **Description** |
| Object put(Object key, Object value) | It is used to insert an entry in this map. |
| void putAll(Map map) | It is used to insert the specified map in this map. |
| Object remove(Object key) | It is used to delete an entry for the specified key. |
| Object get(Object key) | It is used to return the value for the specified key. |
| boolean containsKey(Object key) | It is used to search the specified key from this map. |
| Set keySet() | It is used to return the Set view containing all the keys. |
| Set entrySet() | It is used to return the Set view containing all the keys and values. |

## Map.Entry Interface

Entry is the sub interface of Map. So we will be accessed it by Map.Entry name. It provides methods to get key and value.

### **Methods of Map.Entry interface**

|  |  |
| --- | --- |
| **Method** | **Description** |
| Object getKey() | It is used to obtain key. |
| Object getValue() | It is used to obtain value. |

### **Java Map Example: Generic (New Style)**

1. **import** java.util.\*;
2. **class** MapInterfaceExample{
3. **public** **static** **void** main(String args[]){
4. Map<Integer,String> map=**new** HashMap<Integer,String>();
5. map.put(100,"Amit");
6. map.put(101,"Vijay");
7. map.put(102,"Rahul");
8. **for**(Map.Entry m:map.entrySet()){
9. System.out.println(m.getKey()+" "+m.getValue());
10. }
11. }
12. }

Output:

102 Rahul

100 Amit

101 Vijay

### **Java Map Example: Non-Generic (Old Style)**

1. //Non-generic
2. **import** java.util.\*;
3. **public** **class** MapExample1 {
4. **public** **static** **void** main(String[] args) {
5. Map map=**new** HashMap();
6. //Adding elements to map
7. map.put(1,"Amit");
8. map.put(5,"Rahul");
9. map.put(2,"Jai");
10. map.put(6,"Amit");
11. //Traversing Map
12. Set set=map.entrySet();//Converting to Set so that we can traverse
13. Iterator itr=set.iterator();
14. **while**(itr.hasNext()){
15. //Converting to Map.Entry so that we can get key and value separately
16. Map.Entry entry=(Map.Entry)itr.next();
17. System.out.println(entry.getKey()+" "+entry.getValue());
18. }
19. }
20. }

Output:

1 Amit

2 Jai

5 Rahul

6 Amit

# Java HashMap class

Java HashMap class hierarchy

Java HashMap class implements the map interface by using a hashtable. It inherits AbstractMap class and implements Map interface.

The important points about Java HashMap class are:

* A HashMap contains values based on the key.
* It contains only unique elements.
* It may have one null key and multiple null values.
* It maintains no order.

### **Hierarchy of HashMap class**

As shown in the above figure, HashMap class extends AbstractMap class and implements Map interface.

### **HashMap class declaration**

Let's see the declaration for java.util.HashMap class.

1. **public** **class** HashMap<K,V> **extends** AbstractMap<K,V> **implements** Map<K,V>, Cloneable, Serializable

### **HashMap class Parameters**

Let's see the Parameters for java.util.HashMap class.

* **K**: It is the type of keys maintained by this map.
* **V**: It is the type of mapped values.

### **Constructors of Java HashMap class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| HashMap() | It is used to construct a default HashMap. |
| HashMap(Map m) | It is used to initializes the hash map by using the elements of the given Map object m. |
| HashMap(int capacity) | It is used to initializes the capacity of the hash map to the given integer value, capacity. |
| HashMap(int capacity, float fillRatio) | It is used to initialize both the capacity and fill ratio of the hash map by using its arguments. |

### **Methods of Java HashMap class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void clear() | It is used to remove all of the mappings from this map. |
| boolean containsKey(Object key) | It is used to return true if this map contains a mapping for the specified key. |
| boolean containsValue(Object value) | It is used to return true if this map maps one or more keys to the specified value. |
| boolean isEmpty() | It is used to return true if this map contains no key-value mappings. |
| Object clone() | It is used to return a shallow copy of this HashMap instance: the keys and values themselves are not cloned. |
| Set entrySet() | It is used to return a collection view of the mappings contained in this map. |
| Set keySet() | It is used to return a set view of the keys contained in this map. |
| Object put(Object key, Object value) | It is used to associate the specified value with the specified key in this map. |
| int size() | It is used to return the number of key-value mappings in this map. |
| Collection values() | It is used to return a collection view of the values contained in this map. |

### **Java HashMap Example**

1. **import** java.util.\*;
2. **class** TestCollection13{
3. **public** **static** **void** main(String args[]){
4. HashMap<Integer,String> hm=**new** HashMap<Integer,String>();
5. hm.put(100,"Amit");
6. hm.put(101,"Vijay");
7. hm.put(102,"Rahul");
8. **for**(Map.Entry m:hm.entrySet()){
9. System.out.println(m.getKey()+" "+m.getValue());
10. }
11. }
12. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection13)

Output:102 Rahul

100 Amit

101 Vijay

### **Java HashMap Example: remove()**

1. **import** java.util.\*;
2. **public** **class** HashMapExample {
3. **public** **static** **void** main(String args[]) {
4. // create and populate hash map
5. HashMap<Integer, String> map = **new** HashMap<Integer, String>();
6. map.put(101,"Let us C");
7. map.put(102, "Operating System");
8. map.put(103, "Data Communication and Networking");
9. System.out.println("Values before remove: "+ map);
10. // Remove value for key 102
11. map.remove(102);
12. System.out.println("Values after remove: "+ map);
13. }
14. }

Output:

Values before remove: {102=Operating System, 103=Data Communication and Networking, 101=Let us C}

Values after remove: {103=Data Communication and Networking, 101=Let us C}

### **Difference between HashSet and HashMap**

HashSet contains only values whereas HashMap contains entry(key and value).

### **Java HashMap Example: Book**

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** MapExample {
15. **public** **static** **void** main(String[] args) {
16. //Creating map of Books
17. Map<Integer,Book> map=**new** HashMap<Integer,Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to map
23. map.put(1,b1);
24. map.put(2,b2);
25. map.put(3,b3);
27. //Traversing map
28. **for**(Map.Entry<Integer, Book> entry:map.entrySet()){
29. **int** key=entry.getKey();
30. Book b=entry.getValue();
31. System.out.println(key+" Details:");
32. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
33. }
34. }
35. }

Output:

1 Details:

101 Let us C Yashwant Kanetkar BPB 8

2 Details:

102 Data Communications & Networking Forouzan Mc Graw Hill 4

3 Details:

103 Operating System Galvin Wiley 6

# Java LinkedHashMap class

Java LinkedHashMap class hierarchy

Java LinkedHashMap class is Hash table and Linked list implementation of the Map interface, with predictable iteration order. It inherits HashMap class and implements the Map interface.

The important points about Java LinkedHashMap class are:

* A LinkedHashMap contains values based on the key.
* It contains only unique elements.
* It may have one null key and multiple null values.
* It is same as HashMap instead maintains insertion order.

### **LinkedHashMap class declaration**

Let's see the declaration for java.util.LinkedHashMap class.

1. **public** **class** LinkedHashMap<K,V> **extends** HashMap<K,V> **implements** Map<K,V>

### **LinkedHashMap class Parameters**

Let's see the Parameters for java.util.LinkedHashMap class.

* **K**: It is the type of keys maintained by this map.
* **V**: It is the type of mapped values.

### **Constructors of Java LinkedHashMap class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| LinkedHashMap() | It is used to construct a default LinkedHashMap. |
| LinkedHashMap(int capacity) | It is used to initialize a LinkedHashMap with the given capacity. |
| LinkedHashMap(int capacity, float fillRatio) | It is used to initialize both the capacity and the fillRatio. |
| LinkedHashMap(Map m) | It is used to initialize the LinkedHashMap with the elements from the given Map class m. |

### **Methods of Java LinkedHashMap class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| Object get(Object key) | It is used to return the value to which this map maps the specified key. |
| void clear() | It is used to remove all mappings from this map. |
| boolean containsKey(Object key) | It is used to return true if this map maps one or more keys to the specified value. |

### **Java LinkedHashMap Example**

1. **import** java.util.\*;
2. **class** TestCollection14{
3. **public** **static** **void** main(String args[]){
5. LinkedHashMap<Integer,String> hm=**new** LinkedHashMap<Integer,String>();
7. hm.put(100,"Amit");
8. hm.put(101,"Vijay");
9. hm.put(102,"Rahul");
11. **for**(Map.Entry m:hm.entrySet()){
12. System.out.println(m.getKey()+" "+m.getValue());
13. }
14. }
15. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection14)

Output:100 Amit

101 Vijay

102 Rahul

### **Java LinkedHashMap Example:remove()**

1. **import** java.util.\*;
2. **public** **class** LinkedHashMapExample {
3. **public** **static** **void** main(String args[]) {
4. // Create and populate linked hash map
5. Map<Integer, String> map = **new** LinkedHashMap<Integer, String>();
6. map.put(101,"Let us C");
7. map.put(102, "Operating System");
8. map.put(103, "Data Communication and Networking");
9. System.out.println("Values before remove: "+ map);
10. // Remove value for key 102
11. map.remove(102);
12. System.out.println("Values after remove: "+ map);
13. }
14. }

Output:

Values before remove: {101=Let us C, 102=Operating System, 103=Data Communication and Networking}

Values after remove: {101=Let us C, 103=Data Communication and Networking}

### **Java LinkedHashMap Example: Book**

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** MapExample {
15. **public** **static** **void** main(String[] args) {
16. //Creating map of Books
17. Map<Integer,Book> map=**new** LinkedHashMap<Integer,Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to map
23. map.put(2,b2);
24. map.put(1,b1);
25. map.put(3,b3);
27. //Traversing map
28. **for**(Map.Entry<Integer, Book> entry:map.entrySet()){
29. **int** key=entry.getKey();
30. Book b=entry.getValue();
31. System.out.println(key+" Details:");
32. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
33. }
34. }
35. }

Output:

2 Details:

102 Data Communications & Networking Forouzan Mc Graw Hill 4

1 Details:

101 Let us C Yashwant Kanetkar BPB 8

3 Details:

103 Operating System Galvin Wiley 6

# Java TreeMap class

Java TreeMap class hierarchy

Java TreeMap class implements the Map interface by using a tree. It provides an efficient means of storing key/value pairs in sorted order.

The important points about Java TreeMap class are:

* A TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.
* It contains only unique elements.
* It cannot have null key but can have multiple null values.
* It is same as HashMap instead maintains ascending order.

### **TreeMap class declaration**

Let's see the declaration for java.util.TreeMap class.

1. **public** **class** TreeMap<K,V> **extends** AbstractMap<K,V> **implements** NavigableMap<K,V>, Cloneable, Serializable

### **TreeMap class Parameters**

Let's see the Parameters for java.util.TreeMap class.

* **K**: It is the type of keys maintained by this map.
* **V**: It is the type of mapped values.

### **Constructors of Java TreeMap class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| TreeMap() | It is used to construct an empty tree map that will be sorted using the natural order of its key. |
| TreeMap(Comparator comp) | It is used to construct an empty tree-based map that will be sorted using the comparator comp. |
| TreeMap(Map m) | It is used to initialize a tree map with the entries from **m**, which will be sorted using the natural order of the keys. |
| TreeMap(SortedMap sm) | It is used to initialize a tree map with the entries from the SortedMap **sm**, which will be sorted in the same order as **sm.** |

### **Methods of Java TreeMap class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean containsKey(Object key) | It is used to return true if this map contains a mapping for the specified key. |
| boolean containsValue(Object value) | It is used to return true if this map maps one or more keys to the specified value. |
| Object firstKey() | It is used to return the first (lowest) key currently in this sorted map. |
| Object get(Object key) | It is used to return the value to which this map maps the specified key. |
| Object lastKey() | It is used to return the last (highest) key currently in this sorted map. |
| Object remove(Object key) | It is used to remove the mapping for this key from this TreeMap if present. |
| void putAll(Map map) | It is used to copy all of the mappings from the specified map to this map. |
| Set entrySet() | It is used to return a set view of the mappings contained in this map. |
| int size() | It is used to return the number of key-value mappings in this map. |
| Collection values() | It is used to return a collection view of the values contained in this map. |

### **Java TreeMap Example:**

1. **import** java.util.\*;
2. **class** TestCollection15{
3. **public** **static** **void** main(String args[]){
4. TreeMap<Integer,String> hm=**new** TreeMap<Integer,String>();
5. hm.put(100,"Amit");
6. hm.put(102,"Ravi");
7. hm.put(101,"Vijay");
8. hm.put(103,"Rahul");
9. **for**(Map.Entry m:hm.entrySet()){
10. System.out.println(m.getKey()+" "+m.getValue());
11. }
12. }
13. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection15)

Output:100 Amit

101 Vijay

102 Ravi

103 Rahul

### **Java TreeMap Example: remove()**

1. **import** java.util.\*;
2. **public** **class** TreeMapExample {
3. **public** **static** **void** main(String args[]) {
4. // Create and populate tree map
5. Map<Integer, String> map = **new** TreeMap<Integer, String>();
6. map.put(102,"Let us C");
7. map.put(103, "Operating System");
8. map.put(101, "Data Communication and Networking");
9. System.out.println("Values before remove: "+ map);
10. // Remove value for key 102
11. map.remove(102);
12. System.out.println("Values after remove: "+ map);
13. }
14. }

Output:

Values before remove: {101=Data Communication and Networking, 102=Let us C, 103=Operating System}

Values after remove: {101=Data Communication and Networking, 103=Operating System}

### **What is difference between HashMap and TreeMap?**

|  |  |
| --- | --- |
| **HashMap** | **TreeMap** |
| 1) HashMap can contain one null key. | TreeMap can not contain any null key. |
| 2) HashMap maintains no order. | TreeMap maintains ascending order. |

### **Java TreeMap Example: Book**

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** MapExample {
15. **public** **static** **void** main(String[] args) {
16. //Creating map of Books
17. Map<Integer,Book> map=**new** TreeMap<Integer,Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to map
23. map.put(2,b2);
24. map.put(1,b1);
25. map.put(3,b3);
27. //Traversing map
28. **for**(Map.Entry<Integer, Book> entry:map.entrySet()){
29. **int** key=entry.getKey();
30. Book b=entry.getValue();
31. System.out.println(key+" Details:");
32. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
33. }
34. }
35. }

Output:

1 Details:

101 Let us C Yashwant Kanetkar BPB 8

2 Details:

102 Data Communications & Networking Forouzan Mc Graw Hill 4

3 Details:

103 Operating System Galvin Wiley 6

# Java Hashtable class

Java Hashtable class implements a hashtable, which maps keys to values. It inherits Dictionary class and implements the Map interface.

The important points about Java Hashtable class are:

* A Hashtable is an array of list. Each list is known as a bucket. The position of bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.
* It contains only unique elements.
* It may have not have any null key or value.
* It is synchronized.

### **Hashtable class declaration**

Let's see the declaration for java.util.Hashtable class.

1. **public** **class** Hashtable<K,V> **extends** Dictionary<K,V> **implements** Map<K,V>, Cloneable, Serializable

### **Hashtable class Parameters**

Let's see the Parameters for java.util.Hashtable class.

* **K**: It is the type of keys maintained by this map.
* **V**: It is the type of mapped values.

### **Constructors of Java Hashtable class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| Hashtable() | It is the default constructor of hash table it instantiates the Hashtable class. |
| Hashtable(int size) | It is used to accept an integer parameter and creates a hash table that has an initial size specified by integer value size. |
| Hashtable(int size, float fillRatio) | It is used to create a hash table that has an initial size specified by size and a fill ratio specified by fillRatio. |

### **Methods of Java Hashtable class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void clear() | It is used to reset the hash table. |
| boolean contains(Object value) | This method return true if some value equal to the value exist within the hash table, else return false. |
| boolean containsValue(Object value) | This method return true if some value equal to the value exists within the hash table, else return false. |
| boolean containsKey(Object key) | This method return true if some key equal to the key exists within the hash table, else return false. |
| boolean isEmpty() | This method return true if the hash table is empty; returns false if it contains at least one key. |
| void rehash() | It is used to increase the size of the hash table and rehashes all of its keys. |
| Object get(Object key) | This method return the object that contains the value associated with the key. |
| Object remove(Object key) | It is used to remove the key and its value. This method return the value associated with the key. |
| int size() | This method return the number of entries in the hash table. |

### **Java Hashtable Example**

1. **import** java.util.\*;
2. **class** TestCollection16{
3. **public** **static** **void** main(String args[]){
4. Hashtable<Integer,String> hm=**new** Hashtable<Integer,String>();
6. hm.put(100,"Amit");
7. hm.put(102,"Ravi");
8. hm.put(101,"Vijay");
9. hm.put(103,"Rahul");
11. **for**(Map.Entry m:hm.entrySet()){
12. System.out.println(m.getKey()+" "+m.getValue());
13. }
14. }
15. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection16)

Output:

103 Rahul

102 Ravi

101 Vijay

100 Amit

### **Java Hashtable Example: remove()**

1. **import** java.util.\*;
2. **public** **class** HashtableExample {
3. **public** **static** **void** main(String args[]) {
4. // create and populate hash table
5. Hashtable<Integer, String> map = **new** Hashtable<Integer, String>();
6. map.put(102,"Let us C");
7. map.put(103, "Operating System");
8. map.put(101, "Data Communication and Networking");
9. System.out.println("Values before remove: "+ map);
10. // Remove value for key 102
11. map.remove(102);
12. System.out.println("Values after remove: "+ map);
13. }
14. }

Output:

Values before remove: {103=Operating System, 102=Let us C, 101=Data Communication and Networking}

Values after remove: {103=Operating System, 101=Data Communication and Networking}

### **Java Hashtable Example: Book**

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** HashtableExample {
15. **public** **static** **void** main(String[] args) {
16. //Creating map of Books
17. Map<Integer,Book> map=**new** Hashtable<Integer,Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to map
23. map.put(1,b1);
24. map.put(2,b2);
25. map.put(3,b3);
26. //Traversing map
27. **for**(Map.Entry<Integer, Book> entry:map.entrySet()){
28. **int** key=entry.getKey();
29. Book b=entry.getValue();
30. System.out.println(key+" Details:");
31. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
32. }
33. }
34. }

Output:

3 Details:

103 Operating System Galvin Wiley 6

2 Details:

102 Data Communications & Networking Forouzan Mc Graw Hill 4

1 Details:

101 Let us C Yashwant Kanetkar BPB 8

# Difference between HashMap and Hashtable

HashMap and Hashtable both are used to store data in key and value form. Both are using hashing technique to store unique keys.

But there are many differences between HashMap and Hashtable classes that are given below.

|  |  |
| --- | --- |
| **HashMap** | **Hashtable** |
| 1) HashMap is **non synchronized**. It is not-thread safe and can't be shared between many threads without proper synchronization code. | Hashtable is **synchronized**. It is thread-safe and can be shared with many threads. |
| 2) HashMap **allows one null key and multiple null values**. | Hashtable **doesn't allow any null key or value**. |
| 3) HashMap is a **new class introduced in JDK 1.2**. | Hashtable is a **legacy class**. |
| 4) HashMap is **fast**. | Hashtable is **slow**. |
| 5) We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap); | Hashtable is internally synchronized and can't be unsynchronized. |
| 6) HashMap is **traversed by Iterator**. | Hashtable is **traversed by Enumerator and Iterator**. |
| 7) Iterator in HashMap is **fail-fast**. | Enumerator in Hashtable is **not fail-fast**. |
| 8) HashMap inherits **AbstractMap** class. | Hashtable inherits **Dictionary** class. |

# Java EnumSet class

Java EnumSet class is the specialized Set implementation for use with enum types. It inherits AbstractSet class and implements the Set interface.

### **EnumSet class hierarchy**

The hierarchy of EnumSet class is given in the figure given below.

EnumSet class hierarchy

## EnumSet class declaration

Let's see the declaration for java.util.EnumSet class.

1. **public** **abstract** **class** EnumSet<E **extends** Enum<E>> **extends** AbstractSet<E> **implements** Cloneable, Serializable

### **Methods of Java EnumSet class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| static <E extends Enum<E>> EnumSet<E> allOf(Class<E> elementType) | It is used to create an enum set containing all of the elements in the specified element type. |
| static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> c) | It is used to create an enum set initialized from the specified collection. |
| static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> elementType) | It is used to create an empty enum set with the specified element type. |
| static <E extends Enum<E>> EnumSet<E> of(E e) | It is used to create an enum set initially containing the specified element. |
| static <E extends Enum<E>> EnumSet<E> range(E from, E to) | It is used to create an enum set initially containing the specified elements. |
| EnumSet<E> clone() | It is used to return a copy of this set. |

## Java EnumSet Example

1. **import** java.util.\*;
2. **enum** days {
3. SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
4. }
5. **public** **class** EnumSetExample {
6. **public** **static** **void** main(String[] args) {
7. Set<days> set = EnumSet.of(days.TUESDAY, days.WEDNESDAY);
8. // Traversing elements
9. Iterator<days> iter = set.iterator();
10. **while** (iter.hasNext())
11. System.out.println(iter.next());
12. }
13. }

Output:

TUESDAY

WEDNESDAY

## Java EnumSet Example: allOf() and noneOf()

1. **import** java.util.\*;
2. **enum** days {
3. SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
4. }
5. **public** **class** EnumSetExample {
6. **public** **static** **void** main(String[] args) {
7. Set<days> set1 = EnumSet.allOf(days.**class**);
8. System.out.println("Week Days:"+set1);
9. Set<days> set2 = EnumSet.noneOf(days.**class**);
10. System.out.println("Week Days:"+set2);
11. }
12. }

Output:

Week Days:[SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY]

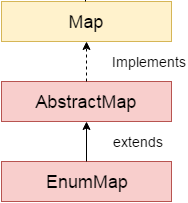
Week Days:[]

# Java EnumMap class

Java EnumMap class is the specialized Map implementation for enum keys. It inherits Enum and AbstractMap classes.

### **EnumMap class hierarchy**

The hierarchy of EnumMap class is given in the figure given below.



## EnumMap class declaration

Let's see the declaration for java.util.EnumMap class.

1. **public** **class** EnumMap<K **extends** Enum<K>,V> **extends** AbstractMap<K,V> **implements** Serializable, Cloneable

## EnumMap class Parameters

Let's see the Parameters for java.util.EnumMap class.

* **K:** It is the type of keys maintained by this map.
* **V:** It is the type of mapped values.

### **Constructors of Java EnumMap class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| EnumMap(Class<K> keyType) | It is used to create an empty enum map with the specified key type. |
| EnumMap(EnumMap<K,? extends V> m) | It is used to create an enum map with the same key type as the specified enum map. |
| EnumMap(Map<K,? extends V> m) | It is used to create an enum map initialized from the specified map. |

### **Methods of Java EnumMap class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void clear() | It is used to remove all mappings from this map. |
| boolean containsKey(Object key) | This method return true if this map contains a mapping for the specified key. |
| boolean containsValue(Object value) | This method return true if this map maps one or more keys to the specified value. |
| boolean equals(Object o) | It is used to compare the specified object with this map for equality. |
| V get(Object key) | This method returns the value to which the specified key is mapped. |
| V put(K key, V value) | It is used to associate the specified value with the specified key in this map. |
| V remove(Object key) | It is used to remove the mapping for this key. |
| Collection<V> values() | It is used to return a Collection view of the values contained in this map. |
| int size() | It is used to return the number of key-value mappings in this map. |

## Java EnumMap Example

1. **import** java.util.\*;
2. **public** **class** EnumMapExample {
3. // create an enum
4. **public** **enum** Days {
5. Monday, Tuesday, Wednesday, Thursday
6. };
7. **public** **static** **void** main(String[] args) {
8. //create and populate enum map
9. EnumMap<Days, String> map = **new** EnumMap<Days, String>(Days.**class**);
10. map.put(Days.Monday, "1");
11. map.put(Days.Tuesday, "2");
12. map.put(Days.Wednesday, "3");
13. map.put(Days.Thursday, "4");
14. // print the map
15. **for**(Map.Entry m:map.entrySet()){
16. System.out.println(m.getKey()+" "+m.getValue());
17. }
18. }
19. }

Output:

Monday 1

Tuesday 2

Wednesday 3

Thursday 4

## Java EnumMap Example: Book

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** EnumMapExample {
15. // Creating enum
16. **public** **enum** Key{
17. One, Two, Three
18. };
19. **public** **static** **void** main(String[] args) {
20. EnumMap<Key, Book> map = **new** EnumMap<Key, Book>(Key.**class**);
21. // Creating Books
22. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
23. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
24. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
25. // Adding Books to Map
26. map.put(Key.One, b1);
27. map.put(Key.Two, b2);
28. map.put(Key.Three, b3);
29. // Traversing EnumMap
30. **for**(Map.Entry<Key, Book> entry:map.entrySet()){
31. Book b=entry.getValue();
32. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
33. }
34. }
35. }

Output:

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications & Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

# Java Collections class

Java collection class is used exclusively with static methods that operate on or return collections. It inherits Object class.

The important points about Java Collections class are:

* Java Collection class supports the **polymorphic algorithms** that operate on collections.
* Java Collection class throws a **NullPointerException** if the collections or class objects provided to them are null.

## Collections class declaration

Let's see the declaration for Java.util.Collections class.

1. **public** **class** Collections **extends** Object

### **Methods of Java Collections class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| static <T> boolean addAll(Collection<? super T> c, T... elements) | It is used to add all of the specified elements to the specified collection. |
| static <T> Queue<T> asLifoQueue(Deque<T> deque) | It is used to return a view of a Deque as a Last-In-First-Out (LIFO) Queue. |
| static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T< c) | It is used to search the specified list for the specified object using the binary search algorithm. |
| static <E> List<E> checkedList(List<E> list, Class<E> type) | It is used to return a dynamically typesafe view of the specified list. |
| static <E> Set<E> checkedSet(Set<E> s, Class<E> type) | It is used to return a dynamically typesafe view of the specified set. |
| static <E> SortedSet<E>checkedSortedSet(SortedSet<E> s, Class<E> type) | It is used to return a dynamically typesafe view of the specified sorted set |
| static void reverse(List<?> list) | It is used to reverse the order of the elements in the specified list. |
| static <T> T max(Collection<? extends T> coll, Comparator<? super T> comp) | It is used to return the maximum element of the given collection, according to the order induced by the specified comparator. |
| static <T extends Object & Comparable<? super T>>T min(Collection<? extends T> coll) | It is used to return the minimum element of the given collection, according to the natural ordering of its elements. |
| static boolean replaceAll(List list, T oldVal, T newVal) | It is used to replace all occurrences of one specified value in a list with another. |

## Java Collections Example

1. **import** java.util.\*;
2. **public** **class** CollectionsExample {
3. **public** **static** **void** main(String a[]){
4. List<String> list = **new** ArrayList<String>();
5. list.add("C");
6. list.add("Core Java");
7. list.add("Advance Java");
8. System.out.println("Initial collection value:"+list);
9. Collections.addAll(list, "Servlet","JSP");
10. System.out.println("After adding elements collection value:"+list);
11. String[] strArr = {"C#", ".Net"};
12. Collections.addAll(list, strArr);
13. System.out.println("After adding array collection value:"+list);
14. }
15. }

Output:

Initial collection value:[C, Core Java, Advance Java]

After adding elements collection value:[C, Core Java, Advance Java, Servlet, JSP]

After adding array collection value:[C, Core Java, Advance Java, Servlet, JSP, C#, .Net]

## Java Collections Example: max()

1. **import** java.util.\*;
2. **public** **class** CollectionsExample {
3. **public** **static** **void** main(String a[]){
4. List<Integer> list = **new** ArrayList<Integer>();
5. list.add(46);
6. list.add(67);
7. list.add(24);
8. list.add(16);
9. list.add(8);
10. list.add(12);
11. System.out.println("Value of maximum element from the collection: "+Collections.max(list));
12. }
13. }

Output:

Value of maximum element from the collection: 67

## Java Collections Example: min()

1. **import** java.util.\*;
2. **public** **class** CollectionsExample {
3. **public** **static** **void** main(String a[]){
4. List<Integer> list = **new** ArrayList<Integer>();
5. list.add(46);
6. list.add(67);
7. list.add(24);
8. list.add(16);
9. list.add(8);
10. list.add(12);
11. System.out.println("Value of minimum element from the collection: "+Collections.min(list));
12. }
13. }

Output:

Value of minimum element from the collection: 8

# Sorting in Collection

We can sort the elements of:

1. String objects
2. Wrapper class objects
3. User-defined class objects

|  |
| --- |
| **Collections** class provides static methods for sorting the elements of collection.If collection elements are of Set type, we can use TreeSet.But We cannot sort the elements of List.Collections class provides methods for sorting the elements of List type elements. |

### **Method of Collections class for sorting List elements**

|  |
| --- |
| **public void sort(List list):** is used to sort the elements of List.List elements must be of Comparable type. |

#### Note: String class and Wrapper classes implements the Comparable interface.So if you store the objects of string or wrapper classes, it will be Comparable.

### **Example of Sorting the elements of List that contains string objects**

1. **import** java.util.\*;
2. **class** TestSort1{
3. **public** **static** **void** main(String args[]){
5. ArrayList<String> al=**new** ArrayList<String>();
6. al.add("Viru");
7. al.add("Saurav");
8. al.add("Mukesh");
9. al.add("Tahir");
11. Collections.sort(al);
12. Iterator itr=al.iterator();
13. **while**(itr.hasNext()){
14. System.out.println(itr.next());
15. }
16. }
17. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestSort1)

Output:Mukesh

Saurav

Tahir

Viru

### **Example of Sorting the elements of List that contains Wrapper class objects**

1. **import** java.util.\*;
2. **class** TestSort2{
3. **public** **static** **void** main(String args[]){
5. ArrayList al=**new** ArrayList();
6. al.add(Integer.valueOf(201));
7. al.add(Integer.valueOf(101));
8. al.add(230);//internally will be converted into objects as Integer.valueOf(230)
10. Collections.sort(al);
12. Iterator itr=al.iterator();
13. **while**(itr.hasNext()){
14. System.out.println(itr.next());
15. }
16. }
17. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestSort2)

Output:101

201

230

# Java Comparable interface

Java Comparable interface is used to order the objects of user-defined class.This interface is found in java.lang package and contains only one method named compareTo(Object). It provide single sorting sequence only i.e. you can sort the elements on based on single data member only. For example it may be rollno, name, age or anything else.

### **compareTo(Object obj) method**

**public int compareTo(Object obj):** is used to compare the current object with the specified object.

We can sort the elements of:

1. String objects
2. Wrapper class objects
3. User-defined class objects

### **Collections class**

**Collections** class provides static methods for sorting the elements of collections. If collection elements are of Set or Map, we can use TreeSet or TreeMap. But We cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements.

### **Method of Collections class for sorting List elements**

**public void sort(List list):** is used to sort the elements of List. List elements must be of Comparable type.

#### Note: String class and Wrapper classes implements Comparable interface by default. So if you store the objects of string or wrapper classes in list, set or map, it will be Comparable by default.

## Java Comparable Example

Let's see the example of Comparable interface that sorts the list elements on the basis of age.

*File: Student.java*

1. **class** Student **implements** Comparable<Student>{
2. **int** rollno;
3. String name;
4. **int** age;
5. Student(**int** rollno,String name,**int** age){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.age=age;
9. }
11. **public** **int** compareTo(Student st){
12. **if**(age==st.age)
13. **return** 0;
14. **else** **if**(age>st.age)
15. **return** 1;
16. **else**
17. **return** -1;
18. }
19. }

*File: TestSort3.java*

1. **import** java.util.\*;
2. **import** java.io.\*;
3. **public** **class** TestSort3{
4. **public** **static** **void** main(String args[]){
5. ArrayList<Student> al=**new** ArrayList<Student>();
6. al.add(**new** Student(101,"Vijay",23));
7. al.add(**new** Student(106,"Ajay",27));
8. al.add(**new** Student(105,"Jai",21));
10. Collections.sort(al);
11. **for**(Student st:al){
12. System.out.println(st.rollno+" "+st.name+" "+st.age);
13. }
14. }
15. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestSort3)

Output:105 Jai 21

101 Vijay 23

106 Ajay 27

Java Comparator interface

**Java Comparator interface** is used to order the objects of user-defined class.

This interface is found in java.util package and contains 2 methods compare(Object obj1,Object obj2) and equals(Object element).

It provides multiple sorting sequence i.e. you can sort the elements on the basis of any data member, for example rollno, name, age or anything else.

compare() method

**public int compare(Object obj1,Object obj2):** compares the first object with second object.

Collections class

**Collections** class provides static methods for sorting the elements of collection. If collection elements are of Set or Map, we can use TreeSet or TreeMap. But we cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements also.

Method of Collections class for sorting List elements

**public void sort(List list, Comparator c):** is used to sort the elements of List by the given Comparator.

Java Comparator Example (Non-generic Old Style)

Let's see the example of sorting the elements of List on the basis of age and name. In this example, we have created 4 java classes:

1. Student.java
2. AgeComparator.java
3. NameComparator.java
4. Simple.java

**Student.java**

This class contains three fields rollno, name and age and a parameterized constructor.

1. **class** Student{
2. **int** rollno;
3. String name;
4. **int** age;
5. Student(**int** rollno,String name,**int** age){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.age=age;
9. }
10. }

**AgeComparator.java**

This class defines comparison logic based on the age. If age of first object is greater than the second, we are returning positive value, it can be any one such as 1, 2 , 10 etc. If age of first object is less than the second object, we are returning negative value, it can be any negative value and if age of both objects are equal, we are returning 0.

1. **import** java.util.\*;
2. **class** AgeComparator **implements** Comparator{
3. **public** **int** compare(Object o1,Object o2){
4. Student s1=(Student)o1;
5. Student s2=(Student)o2;
7. **if**(s1.age==s2.age)
8. **return** 0;
9. **else** **if**(s1.age>s2.age)
10. **return** 1;
11. **else**
12. **return** -1;
13. }
14. }

**NameComparator.java**

This class provides comparison logic based on the name. In such case, we are using the compareTo() method of String class, which internally provides the comparison logic.

1. **import** java.util.\*;
2. **class** NameComparator **implements** Comparator{
3. **public** **int** compare(Object o1,Object o2){
4. Student s1=(Student)o1;
5. Student s2=(Student)o2;
7. **return** s1.name.compareTo(s2.name);
8. }
9. }

**Simple.java**

In this class, we are printing the objects values by sorting on the basis of name and age.

1. **import** java.util.\*;
2. **import** java.io.\*;
4. **class** Simple{
5. **public** **static** **void** main(String args[]){
7. ArrayList al=**new** ArrayList();
8. al.add(**new** Student(101,"Vijay",23));
9. al.add(**new** Student(106,"Ajay",27));
10. al.add(**new** Student(105,"Jai",21));
12. System.out.println("Sorting by Name...");
14. Collections.sort(al,**new** NameComparator());
15. Iterator itr=al.iterator();
16. **while**(itr.hasNext()){
17. Student st=(Student)itr.next();
18. System.out.println(st.rollno+" "+st.name+" "+st.age);
19. }
21. System.out.println("sorting by age...");
23. Collections.sort(al,**new** AgeComparator());
24. Iterator itr2=al.iterator();
25. **while**(itr2.hasNext()){
26. Student st=(Student)itr2.next();
27. System.out.println(st.rollno+" "+st.name+" "+st.age);
28. }

31. }
32. }

Sorting by Name...

106 Ajay 27

105 Jai 21

101 Vijay 23

Sorting by age...

105 Jai 21

101 Vijay 23

106 Ajay 27

Java Comparator Example (Generic)

**Student.java**

1. **class** Student{
2. **int** rollno;
3. String name;
4. **int** age;
5. Student(**int** rollno,String name,**int** age){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.age=age;
9. }
10. }

**AgeComparator.java**

1. **import** java.util.\*;
2. **class** AgeComparator **implements** Comparator<Student>{
3. **public** **int** compare(Student s1,Student s2){
4. **if**(s1.age==s2.age)
5. **return** 0;
6. **else** **if**(s1.age>s2.age)
7. **return** 1;
8. **else**
9. **return** -1;
10. }
11. }

**NameComparator.java**

This class provides comparison logic based on the name. In such case, we are using the compareTo() method of String class, which internally provides the comparison logic.

1. **import** java.util.\*;
2. **class** NameComparator **implements** Comparator<Student>{
3. **public** **int** compare(Student s1,Student s2){
4. **return** s1.name.compareTo(s2.name);
5. }
6. }

**Simple.java**

In this class, we are printing the objects values by sorting on the basis of name and age.

1. **import** java.util.\*;
2. **import** java.io.\*;
3. **class** Simple{
4. **public** **static** **void** main(String args[]){
6. ArrayList<Student> al=**new** ArrayList<Student>();
7. al.add(**new** Student(101,"Vijay",23));
8. al.add(**new** Student(106,"Ajay",27));
9. al.add(**new** Student(105,"Jai",21));
11. System.out.println("Sorting by Name...");
13. Collections.sort(al,**new** NameComparator());
14. **for**(Student st: al){
15. System.out.println(st.rollno+" "+st.name+" "+st.age);
16. }
18. System.out.println("sorting by age...");
20. Collections.sort(al,**new** AgeComparator());
21. **for**(Student st: al){
22. System.out.println(st.rollno+" "+st.name+" "+st.age);
23. }
25. }
26. }

Output:Sorting by Name...

106 Ajay 27

105 Jai 21

101 Vijay 23

Sorting by age...

105 Jai 21

101 Vijay 23

106 Ajay 27

# Properties class in Java

The **properties** object contains key and value pair both as a string. The java.util.Properties class is the subclass of Hashtable.

It can be used to get property value based on the property key. The Properties class provides methods to get data from properties file and store data into properties file. Moreover, it can be used to get properties of system.

### **Advantage of properties file**

**Recompilation is not required, if information is changed from properties file:** If any information is changed from the properties file, you don't need to recompile the java class. It is used to store information which is to be changed frequently.

#### Methods of Properties class

The commonly used methods of Properties class are given below.

|  |  |
| --- | --- |
| **Method** | **Description** |
| public void load(Reader r) | loads data from the Reader object. |
| public void load(InputStream is) | loads data from the InputStream object |
| public String getProperty(String key) | returns value based on the key. |
| public void setProperty(String key,String value) | sets the property in the properties object. |
| public void store(Writer w, String comment) | writers the properties in the writer object. |
| public void store(OutputStream os, String comment) | writes the properties in the OutputStream object. |
| storeToXML(OutputStream os, String comment) | writers the properties in the writer object for generating xml document. |
| public void storeToXML(Writer w, String comment, String encoding) | writers the properties in the writer object for generating xml document with specified encoding. |

### **Example of Properties class to get information from properties file**

To get information from the properties file, create the properties file first.

**db.properties**

1. user=system
2. password=oracle

Now, lets create the java class to read the data from the properties file.

**Test.java**

1. **import** java.util.\*;
2. **import** java.io.\*;
3. **public** **class** Test {
4. **public** **static** **void** main(String[] args)**throws** Exception{
5. FileReader reader=**new** FileReader("db.properties");
7. Properties p=**new** Properties();
8. p.load(reader);
10. System.out.println(p.getProperty("user"));
11. System.out.println(p.getProperty("password"));
12. }
13. }

Output:system

oracle

Now if you change the value of the properties file, you don't need to compile the java class again. That means no maintenance problem.

### **Example of Properties class to get all the system properties**

By System.getProperties() method we can get all the properties of system. Let's create the class that gets information from the system properties.

**Test.java**

1. **import** java.util.\*;
2. **import** java.io.\*;
3. **public** **class** Test {
4. **public** **static** **void** main(String[] args)**throws** Exception{
6. Properties p=System.getProperties();
7. Set set=p.entrySet();
9. Iterator itr=set.iterator();
10. **while**(itr.hasNext()){
11. Map.Entry entry=(Map.Entry)itr.next();
12. System.out.println(entry.getKey()+" = "+entry.getValue());
13. }
15. }
16. }

Output:

java.runtime.name = Java(TM) SE Runtime Environment

sun.boot.library.path = C:\Program Files\Java\jdk1.7.0\_01\jre\bin

java.vm.version = 21.1-b02

java.vm.vendor = Oracle Corporation

java.vendor.url = http://java.oracle.com/

path.separator = ;

java.vm.name = Java HotSpot(TM) Client VM

file.encoding.pkg = sun.io

user.country = US

user.script =

sun.java.launcher = SUN\_STANDARD

...........

### **Example of Properties class to create properties file**

Now lets write the code to create the properties file.

**Test.java**

1. **import** java.util.\*;
2. **import** java.io.\*;
3. **public** **class** Test {
4. **public** **static** **void** main(String[] args)**throws** Exception{
6. Properties p=**new** Properties();
7. p.setProperty("name","Sonoo Jaiswal");
8. p.setProperty("email","sonoojaiswal@javatpoint.com");
10. p.store(**new** FileWriter("info.properties"),"Javatpoint Properties Example");
12. }
13. }

Let's see the generated properties file.

**info.properties**

1. #Javatpoint Properties Example
2. #Thu Oct 03 22:35:53 IST 2013
3. email=sonoojaiswal@javatpoint.com
4. name=Sonoo Jaiswal

# Difference between ArrayList and Vector

ArrayList and Vector both implements List interface and maintains insertion order.

But there are many differences between ArrayList and Vector classes that are given below.

|  |  |
| --- | --- |
| **ArrayList** | **Vector** |
| 1) ArrayList is **not synchronized**. | Vector is **synchronized**. |
| 2) ArrayList **increments 50%** of current array size if number of element exceeds from its capacity. | Vector **increments 100%** means doubles the array size if total number of element exceeds than its capacity. |
| 3) ArrayList is **not a legacy** class, it is introduced in JDK 1.2. | Vector is a **legacy** class. |
| 4) ArrayList is **fast** because it is non-synchronized. | Vector is **slow** because it is synchronized i.e. in multithreading environment, it will hold the other threads in runnable or non-runnable state until current thread releases the lock of object. |
| 5) ArrayList uses **Iterator** interface to traverse the elements. | Vector uses **Enumeration** interface to traverse the elements. But it can use Iterator also. |

### **Example of Java ArrayList**

Let's see a simple example where we are using ArrayList to store and traverse the elements.

1. **import** java.util.\*;
2. **class** TestArrayList21{
3. **public** **static** **void** main(String args[]){
5. List<String> al=**new** ArrayList<String>();//creating arraylist
6. al.add("Sonoo");//adding object in arraylist
7. al.add("Michael");
8. al.add("James");
9. al.add("Andy");
10. //traversing elements using Iterator
11. Iterator itr=al.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestArrayList21)

Output:

Sonoo

Michael

James

Andy

### **Example of Java Vector**

Let's see a simple example of java Vector class that uses Enumeration interface.

1. **import** java.util.\*;
2. **class** TestVector1{
3. **public** **static** **void** main(String args[]){
4. Vector<String> v=**new** Vector<String>();//creating vector
5. v.add("umesh");//method of Collection
6. v.addElement("irfan");//method of Vector
7. v.addElement("kumar");
8. //traversing elements using Enumeration
9. Enumeration e=v.elements();
10. **while**(e.hasMoreElements()){
11. System.out.println(e.nextElement());
12. }
13. }
14. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestVector1)

Output:

umesh

irfan

kumar