# 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. }

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [**next →**](https://www.javatpoint.com/difference-between-stringbuffer-and-stringbuilder)[**← prev**](https://www.javatpoint.com/StringBuilder-class)  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. | |

Java Arrays

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

**Java array** is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.

In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimentional or multidimentional arrays in Java.

Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.

Advantages

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

Disadvantages

* **Size Limit:** We can store only the 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 which grows automatically.

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 Java Array

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

1. //Java Program to illustrate how to declare, instantiate, initialize
2. //and traverse the Java array.
3. **class** Testarray{
4. **public** **static** **void** main(String args[]){
5. **int** a[]=**new** **int**[5];//declaration and instantiation
6. a[0]=10;//initialization
7. a[1]=20;
8. a[2]=70;
9. a[3]=40;
10. a[4]=50;
11. //traversing array
12. **for**(**int** i=0;i<a.length;i++)//length is the property of array
13. System.out.println(a[i]);
14. }}

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

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. //Java Program to illustrate the use of declaration, instantiation
2. //and initialization of Java array in a single line
3. **class** Testarray1{
4. **public** **static** **void** main(String args[]){
5. **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]);
9. }}

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

Output:

33

3

4

5

For-each Loop for Java Array

We can also print the Java array using **[for-each loop](https://www.javatpoint.com/for-each-loop)**. The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

The syntax of the for-each loop is given below:

1. **for**(data\_type variable:array){
2. //body of the loop
3. }

Let us see the example of print the elements of Java array using the for-each loop.

1. //Java Program to print the array elements using for-each loop
2. **class** Testarray1{
3. **public** **static** **void** main(String args[]){
4. **int** arr[]={33,3,4,5};
5. //printing array using for-each loop
6. **for**(**int** i:arr)
7. System.out.println(i);
8. }}

Output:

33

3

4

5

Passing Array to a 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 the minimum number of an array using a method.

1. //Java Program to demonstrate the way of passing an array
2. //to method.
3. **class** Testarray2{
4. //creating a method which receives an array as a parameter
5. **static** **void** min(**int** arr[]){
6. **int** min=arr[0];
7. **for**(**int** i=1;i<arr.length;i++)
8. **if**(min>arr[i])
9. min=arr[i];
11. System.out.println(min);
12. }
14. **public** **static** **void** main(String args[]){
15. **int** a[]={33,3,4,5};//declaring and initializing an array
16. min(a);//passing array to method
17. }}

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

Output:

3

Anonymous Array in Java

Java supports the feature of an anonymous array, so you don't need to declare the array while passing an array to the method.

1. //Java Program to demonstrate the way of passing an anonymous array
2. //to method.
3. **public** **class** TestAnonymousArray{
4. //creating a method which receives an array as a parameter
5. **static** **void** printArray(**int** arr[]){
6. **for**(**int** i=0;i<arr.length;i++)
7. System.out.println(arr[i]);
8. }
10. **public** **static** **void** main(String args[]){
11. printArray(**new** **int**[]{10,22,44,66});//passing anonymous array to method
12. }}

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TestAnonymousArray" \t "_blank)**

Output:

10

22

44

66

Returning Array from the Method

We can also return an array from the method in Java.

1. //Java Program to return an array from the method
2. **class** TestReturnArray{
3. //creating method which returns an array
4. **static** **int**[] get(){
5. **return** **new** **int**[]{10,30,50,90,60};
6. }
8. **public** **static** **void** main(String args[]){
9. //calling method which returns an array
10. **int** arr[]=get();
11. //printing the values of an array
12. **for**(**int** i=0;i<arr.length;i++)
13. System.out.println(arr[i]);
14. }}

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TestReturnArray" \t "_blank)**

Output:

10

30

50

90

60

ArrayIndexOutOfBoundsException

The Java Virtual Machine (JVM) throws an ArrayIndexOutOfBoundsException if length of the array in negative, equal to the array size or greater than the array size while traversing the array.

1. //Java Program to demonstrate the case of
2. //ArrayIndexOutOfBoundsException in a Java Array.
3. **public** **class** TestArrayException{
4. **public** **static** **void** main(String args[]){
5. **int** arr[]={50,60,70,80};
6. **for**(**int** i=0;i<=arr.length;i++){
7. System.out.println(arr[i]);
8. }
9. }}

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TestArrayException" \t "_blank)**

Output:

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4

at TestArrayException.main(TestArrayException.java:5)

50

60

70

80

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

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

Output:

1 2 3

2 4 5

4 4 5

Jagged Array in Java

If we are creating odd number of columns in a 2D array, it is known as a jagged array. In other words, it is an array of arrays with different number of columns.

1. //Java Program to illustrate the jagged array
2. **class** TestJaggedArray{
3. **public** **static** **void** main(String[] args){
4. //declaring a 2D array with odd columns
5. **int** arr[][] = **new** **int**[3][];
6. arr[0] = **new** **int**[3];
7. arr[1] = **new** **int**[4];
8. arr[2] = **new** **int**[2];
9. //initializing a jagged array
10. **int** count = 0;
11. **for** (**int** i=0; i<arr.length; i++)
12. **for**(**int** j=0; j<arr[i].length; j++)
13. arr[i][j] = count++;
15. //printing the data of a jagged array
16. **for** (**int** i=0; i<arr.length; i++){
17. **for** (**int** j=0; j<arr[i].length; j++){
18. System.out.print(arr[i][j]+" ");
19. }
20. System.out.println();//new line
21. }
22. }
23. }

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TestJaggedArray" \t "_blank)**

Output:

0 1 2

3 4 5 6

7 8

What is the class name of Java array?

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

1. //Java Program to get the class name of array in Java
2. **class** Testarray4{
3. **public** **static** **void** main(String args[]){
4. //declaration and initialization of array
5. **int** arr[]={4,4,5};
6. //getting the class name of Java array
7. Class c=arr.getClass();
8. String name=c.getName();
9. //printing the class name of Java array
10. System.out.println(name);
12. }}

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

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 Copying an Array in Java

1. //Java Program to copy a source array into a destination array in Java
2. **class** TestArrayCopyDemo {
3. **public** **static** **void** main(String[] args) {
4. //declaring a source array
5. **char**[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
6. 'i', 'n', 'a', 't', 'e', 'd' };
7. //declaring a destination array
8. **char**[] copyTo = **new** **char**[7];
9. //copying array using System.arraycopy() method
10. System.arraycopy(copyFrom, 2, copyTo, 0, 7);
11. //printing the destination array
12. System.out.println(String.valueOf(copyTo));
13. }
14. }

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

Output:

caffein

Cloning an Array in Java

Since, Java array implements the Cloneable interface, we can create the clone of the Java array. If we create the clone of a single-dimensional array, it creates the deep copy of the Java array. It means, it will copy the actual value. But, if we create the clone of a multidimensional array, it creates the shallow copy of the Java array which means it copies the references.

1. //Java Program to clone the array
2. **class** Testarray1{
3. **public** **static** **void** main(String args[]){
4. **int** arr[]={33,3,4,5};
5. System.out.println("Printing original array:");
6. **for**(**int** i:arr)
7. System.out.println(i);
9. System.out.println("Printing clone of the array:");
10. **int** carr[]=arr.clone();
11. **for**(**int** i:carr)
12. System.out.println(i);
14. System.out.println("Are both equal?");
15. System.out.println(arr==carr);
17. }}

Output:

Printing original array:

33

3

4

5

Printing clone of the array:

33

3

4

5

Are both equal?

false

Addition of 2 Matrices in Java

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

1. //Java Program to demonstrate the addition of two matrices in Java
2. **class** Testarray5{
3. **public** **static** **void** main(String args[]){
4. //creating two matrices
5. **int** a[][]={{1,3,4},{3,4,5}};
6. **int** b[][]={{1,3,4},{3,4,5}};
8. //creating another matrix to store the sum of two matrices
9. **int** c[][]=**new** **int**[2][3];
11. //adding and printing addition of 2 matrices
12. **for**(**int** i=0;i<2;i++){
13. **for**(**int** j=0;j<3;j++){
14. c[i][j]=a[i][j]+b[i][j];
15. System.out.print(c[i][j]+" ");
16. }
17. System.out.println();//new line
18. }
20. }}

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

Output:

2 6 8

6 8 10

Multiplication of 2 Matrices in Java

In the case of matrix multiplication, a one-row element of the first matrix is multiplied by all the columns of the second matrix which can be understood by the image given below.

Let's see a simple example to multiply two matrices of 3 rows and 3 columns.

1. //Java Program to multiply two matrices
2. **public** **class** MatrixMultiplicationExample{
3. **public** **static** **void** main(String args[]){
4. //creating two matrices
5. **int** a[][]={{1,1,1},{2,2,2},{3,3,3}};
6. **int** b[][]={{1,1,1},{2,2,2},{3,3,3}};
8. //creating another matrix to store the multiplication of two matrices
9. **int** c[][]=**new** **int**[3][3];  //3 rows and 3 columns
11. //multiplying and printing multiplication of 2 matrices
12. **for**(**int** i=0;i<3;i++){
13. **for**(**int** j=0;j<3;j++){
14. c[i][j]=0;
15. **for**(**int** k=0;k<3;k++)
16. {
17. c[i][j]+=a[i][k]\*b[k][j];
18. }//end of k loop
19. System.out.print(c[i][j]+" ");  //printing matrix element
20. }//end of j loop
21. System.out.println();//new line
22. }
23. }}

**[Test it Now](https://compiler.javatpoint.com/opr/test.jsp?filename=MatrixMultiplicationExample" \t "_blank)**

Output:

6 6 6

12 12 12

18 18 18

# 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{}

# 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. }

# Abstract class in Java

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

Before learning the 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 essential things to the user and hides the internal details, for example, sending SMS where you 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 which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

#### Points to Remember

* An abstract class must be declared with an abstract keyword.
* It can have abstract and non-abstract methods.
* It cannot be instantiated.
* It can have constructors and static methods also.
* It can have final methods which will force the subclass not to change the body of the method.

**Example of abstract class**

1. **abstract** **class** A{}

### Abstract Method in Java

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

**Example of abstract method**

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

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

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

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" \t "_blank)**

running safely

### Understanding the real scenario of Abstract class

In this example, Shape is the abstract class, and its implementation is provided by the Rectangle and Circle classes.

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

A **factory method** is a method that returns the instance of the class. 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 a 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" \t "_blank)**

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" \t "_blank)**

Rate of Interest is: 7 %

Rate of Interest is: 8 %

### Abstract class having constructor, data member and methods

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

*File: TestAbstraction2.java*

1. //Example of an abstract class that has abstract and non-abstract methods
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. }
7. //Creating a Child class which inherits Abstract class
8. **class** Honda **extends** Bike{
9. **void** run(){System.out.println("running safely..");}
10. }
11. //Creating a Test class which calls abstract and non-abstract methods
12. **class** TestAbstraction2{
13. **public** **static** **void** main(String args[]){
14. Bike obj = **new** Honda();
15. obj.run();
16. obj.changeGear();
17. }
18. }

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

bike is created

running safely..

gear changed

#### Rule: If there is an 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" \t "_blank)**

compile time error

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

### 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. }}

# 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 abstract methods and variables. It cannot have a method body.

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

It cannot be instantiated just like the abstract class.

Since Java 8, we can have **default and static methods** in an interface.

Since Java 9, we can have **private methods** in an interface.

## 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 an interface?

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

### Syntax:

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 the compiler

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

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

#### The relationship between classes and interfaces

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

## Java Interface Example

In this example, the Printable interface has only one method, and its implementation is provided in the A6 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" \t "_blank)**

Output:

Hello

## Java Interface Example: Drawable

In this example, the Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In a real scenario, an interface is defined by someone else, but its implementation is provided by different implementation providers. Moreover, it is used by someone else. The implementation part is hidden by the user who 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" \t "_blank)**

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" \t "_blank)**

Output:

ROI: 9.15

## Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is 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" \t "_blank)**

Output:Hello

Welcome

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

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

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 an 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" \t "_blank)**

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" \t "_blank)**

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" \t "_blank)**

Output:

drawing rectangle

27

## Q) What is marker or tagged interface?

An interface which has no member is known as a 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 which is known as a 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. }

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [**next →**](https://www.javatpoint.com/package)[**← prev**](https://www.javatpoint.com/interface-in-java) 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** 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. }} |

# 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" \t "_blank)**

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" \t "_blank)**

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" \t "_blank)**

Output:

data is 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. [Exception Example](https://www.javatpoint.com/exception-handling-in-java#exceptionexample)
6. [Scenarios where an 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 exceptions, its type and the difference between checked and unchecked exceptions.

## What is Exception in Java

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

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

## What is Exception Handling

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

### Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application that is why we 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 are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed. If we perform exception handling, the 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

The java.lang.Throwable class is the root class of Java Exception hierarchy which is inherited by two subclasses: Exception and Error. A hierarchy of Java Exception classes are given below:

### Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, 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 which directly inherit 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 which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

### 3) Error

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

## Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java.

|  |  |
| --- | --- |
| **Keyword** | **Description** |
| try | The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone. |
| catch | The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later. |
| finally | The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not. |
| throw | The "throw" keyword is used to throw an exception. |
| throws | The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature. |

## Java Exception Handling Example

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

1. **public** **class** JavaExceptionExample{
2. **public** **static** **void** main(String args[]){
3. **try**{
4. //code that may raise exception
5. **int** data=100/0;
6. }**catch**(ArithmeticException e){System.out.println(e);}
7. //rest code of the program
8. System.out.println("rest of the code...");
9. }
10. }

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=JavaExceptionExample" \t "_blank)**

Output:

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

rest of the code...

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

## Common Scenarios of Java Exceptions

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

### 1) A scenario where ArithmeticException occurs

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

1. **int** a=50/0;//ArithmeticException

### 2) A scenario where NullPointerException occurs

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

1. String s=**null**;
2. System.out.println(s.length());//NullPointerException

### 3) A scenario where NumberFormatException occurs

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

1. String s="abc";
2. **int** i=Integer.parseInt(s);//NumberFormatException

### 4) A scenario where ArrayIndexOutOfBoundsException occurs

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

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

# Java try-catch block

## Java try block

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

If an exception occurs at the particular statement of try block, the rest of the block code will not execute. So, it is recommended not to keeping the code in try block that will not throw an exception.

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

### Syntax of Java try-catch

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

### Syntax of try-finally block

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

## Java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception ( i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

## Problem without exception handling

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

### Example 1

1. **public** **class** TryCatchExample1 {
3. **public** **static** **void** main(String[] args) {
5. **int** data=50/0; //may throw exception
7. System.out.println("rest of the code");
9. }
11. }

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample1" \t "_blank)**

**Output:**

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

As displayed in the above example, the **rest of the code** is not executed (in such case, the **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 the above problem by a java try-catch block.

### Example 2

1. **public** **class** TryCatchExample2 {
3. **public** **static** **void** main(String[] args) {
4. **try**
5. {
6. **int** data=50/0; //may throw exception
7. }
8. //handling the exception
9. **catch**(ArithmeticException e)
10. {
11. System.out.println(e);
12. }
13. System.out.println("rest of the code");
14. }
16. }

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample2" \t "_blank)**

**Output:**

java.lang.ArithmeticException: / by zero

rest of the code

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

### Example 3

In this example, we also kept the code in a try block that will not throw an exception.

1. **public** **class** TryCatchExample3 {
3. **public** **static** **void** main(String[] args) {
4. **try**
5. {
6. **int** data=50/0; //may throw exception
7. // if exception occurs, the remaining statement will not exceute
8. System.out.println("rest of the code");
9. }
10. // handling the exception
11. **catch**(ArithmeticException e)
12. {
13. System.out.println(e);
14. }
16. }
18. }

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample3" \t "_blank)**

**Output:**

java.lang.ArithmeticException: / by zero

Here, we can see that if an exception occurs in the try block, the rest of the block code will not execute.

### Example 4

Here, we handle the exception using the parent class exception.

1. **public** **class** TryCatchExample4 {
3. **public** **static** **void** main(String[] args) {
4. **try**
5. {
6. **int** data=50/0; //may throw exception
7. }
8. // handling the exception by using Exception class
9. **catch**(Exception e)
10. {
11. System.out.println(e);
12. }
13. System.out.println("rest of the code");
14. }
16. }

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample4" \t "_blank)**

**Output:**

java.lang.ArithmeticException: / by zero

rest of the code

### Example 5

Let's see an example to print a custom message on exception.

1. **public** **class** TryCatchExample5 {
3. **public** **static** **void** main(String[] args) {
4. **try**
5. {
6. **int** data=50/0; //may throw exception
7. }
8. // handling the exception
9. **catch**(Exception e)
10. {
11. // displaying the custom message
12. System.out.println("Can't divided by zero");
13. }
14. }
16. }

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample5" \t "_blank)**

**Output:**

Can't divided by zero

### Example 6

Let's see an example to resolve the exception in a catch block.

1. **public** **class** TryCatchExample6 {
3. **public** **static** **void** main(String[] args) {
4. **int** i=50;
5. **int** j=0;
6. **int** data;
7. **try**
8. {
9. data=i/j; //may throw exception
10. }
11. // handling the exception
12. **catch**(Exception e)
13. {
14. // resolving the exception in catch block
15. System.out.println(i/(j+2));
16. }
17. }
18. }

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample6" \t "_blank)**

**Output:**

25

### Example 7

In this example, along with try block, we also enclose exception code in a catch block.

1. **public** **class** TryCatchExample7 {
3. **public** **static** **void** main(String[] args) {
5. **try**
6. {
7. **int** data1=50/0; //may throw exception
9. }
10. // handling the exception
11. **catch**(Exception e)
12. {
13. // generating the exception in catch block
14. **int** data2=50/0; //may throw exception
16. }
17. System.out.println("rest of the code");
18. }
19. }

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample7" \t "_blank)**

**Output:**

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

Here, we can see that the catch block didn't contain the exception code. So, enclose exception code within a try block and use catch block only to handle the exceptions.

### Example 8

In this example, we handle the generated exception (Arithmetic Exception) with a different type of exception class (ArrayIndexOutOfBoundsException).

1. **public** **class** TryCatchExample8 {
3. **public** **static** **void** main(String[] args) {
4. **try**
5. {
6. **int** data=50/0; //may throw exception
8. }
9. // try to handle the ArithmeticException using ArrayIndexOutOfBoundsException
10. **catch**(ArrayIndexOutOfBoundsException e)
11. {
12. System.out.println(e);
13. }
14. System.out.println("rest of the code");
15. }
17. }

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample8" \t "_blank)**

**Output:**

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

### Example 9

Let's see an example to handle another unchecked exception.

1. **public** **class** TryCatchExample9 {
3. **public** **static** **void** main(String[] args) {
4. **try**
5. {
6. **int** arr[]= {1,3,5,7};
7. System.out.println(arr[10]); //may throw exception
8. }
9. // handling the array exception
10. **catch**(ArrayIndexOutOfBoundsException e)
11. {
12. System.out.println(e);
13. }
14. System.out.println("rest of the code");
15. }
17. }

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample9" \t "_blank)**

**Output:**

java.lang.ArrayIndexOutOfBoundsException: 10

rest of the code

### Example 10

Let's see an example to handle checked exception.

1. **import** java.io.FileNotFoundException;
2. **import** java.io.PrintWriter;
4. **public** **class** TryCatchExample10 {
6. **public** **static** **void** main(String[] args) {

9. PrintWriter pw;
10. **try** {
11. pw = **new** PrintWriter("jtp.txt"); //may throw exception
12. pw.println("saved");
13. }
14. // providing the checked exception handler
15. **catch** (FileNotFoundException e) {
17. System.out.println(e);
18. }
19. System.out.println("File saved successfully");
20. }
21. }

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample10" \t "_blank)**

**Output:**

File saved successfully

## 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

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

## Points to remember

* At a time only one exception occurs and at a time only one catch block is executed.
* All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

### Example 1

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

1. **public** **class** MultipleCatchBlock1 {
3. **public** **static** **void** main(String[] args) {
5. **try**{
6. **int** a[]=**new** **int**[5];
7. a[5]=30/0;
8. }
9. **catch**(ArithmeticException e)
10. {
11. System.out.println("Arithmetic Exception occurs");
12. }
13. **catch**(ArrayIndexOutOfBoundsException e)
14. {
15. System.out.println("ArrayIndexOutOfBounds Exception occurs");
16. }
17. **catch**(Exception e)
18. {
19. System.out.println("Parent Exception occurs");
20. }
21. System.out.println("rest of the code");
22. }
23. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=MultipleCatchBlock1)

**Output:**

Arithmetic Exception occurs

rest of the code

### Example 2

1. **public** **class** MultipleCatchBlock2 {
3. **public** **static** **void** main(String[] args) {
5. **try**{
6. **int** a[]=**new** **int**[5];
8. System.out.println(a[10]);
9. }
10. **catch**(ArithmeticException e)
11. {
12. System.out.println("Arithmetic Exception occurs");
13. }
14. **catch**(ArrayIndexOutOfBoundsException e)
15. {
16. System.out.println("ArrayIndexOutOfBounds Exception occurs");
17. }
18. **catch**(Exception e)
19. {
20. System.out.println("Parent Exception occurs");
21. }
22. System.out.println("rest of the code");
23. }
24. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=MultipleCatchBlock2)

**Output:**

ArrayIndexOutOfBounds Exception occurs

rest of the code

### Example 3

In this example, try block contains two exceptions. But at a time only one exception occurs and its corresponding catch block is invoked.

1. **public** **class** MultipleCatchBlock3 {
3. **public** **static** **void** main(String[] args) {
5. **try**{
6. **int** a[]=**new** **int**[5];
7. a[5]=30/0;
8. System.out.println(a[10]);
9. }
10. **catch**(ArithmeticException e)
11. {
12. System.out.println("Arithmetic Exception occurs");
13. }
14. **catch**(ArrayIndexOutOfBoundsException e)
15. {
16. System.out.println("ArrayIndexOutOfBounds Exception occurs");
17. }
18. **catch**(Exception e)
19. {
20. System.out.println("Parent Exception occurs");
21. }
22. System.out.println("rest of the code");
23. }
24. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=MultipleCatchBlock3)

**Output:**

Arithmetic Exception occurs

rest of the code

### Example 4

In this example, we generate NullPointerException, but didn't provide the corresponding exception type. In such case, the catch block containing the parent exception class **Exception** will invoked.

1. **public** **class** MultipleCatchBlock4 {
3. **public** **static** **void** main(String[] args) {
5. **try**{
6. String s=**null**;
7. System.out.println(s.length());
8. }
9. **catch**(ArithmeticException e)
10. {
11. System.out.println("Arithmetic Exception occurs");
12. }
13. **catch**(ArrayIndexOutOfBoundsException e)
14. {
15. System.out.println("ArrayIndexOutOfBounds Exception occurs");
16. }
17. **catch**(Exception e)
18. {
19. System.out.println("Parent Exception occurs");
20. }
21. System.out.println("rest of the code");
22. }
23. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=MultipleCatchBlock4)

**Output:**

Parent Exception occurs

rest of the code

### Example 5

Let's see an example, to handle the exception without maintaining the order of exceptions (i.e. from most specific to most general).

1. **class** MultipleCatchBlock5{
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=MultipleCatchBlock5)

**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" \t "_blank)**

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" \t "_blank)**

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" \t "_blank)**

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" \t "_blank)**

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" \t "_blank)**

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" \t "_blank)**

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" \t "_blank)**

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" \t "_blank)**

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" \t "_blank)**

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" \t "_blank)**

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" \t "_blank)**

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" \t "_blank)**

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" \t "_blank)**

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" \t "_blank)**

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...

# Multithreading in Java

1. [Multithreading](https://www.javatpoint.com/multithreading-in-java)
2. [Multitasking](https://www.javatpoint.com/multithreading-in-java#multitasing)
3. [Process-based multitasking](https://www.javatpoint.com/multithreading-in-java#multiprocessing)
4. [Thread-based multitasking](https://www.javatpoint.com/multithreading-in-java#multithreading)
5. [What is Thread](https://www.javatpoint.com/multithreading-in-java#thread)

**Multithreading in java** is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

### Advantages of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.

2) You **can perform many operations together, so it saves time**.

3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

## Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

* Process-based Multitasking (Multiprocessing)
* Thread-based Multitasking (Multithreading)

### 1) Process-based Multitasking (Multiprocessing)

* Each process has an address in memory. In other words, each process allocates a separate memory area.
* A process is heavyweight.
* Cost of communication between the process is high.
* Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

### 2) Thread-based Multitasking (Multithreading)

* Threads share the same address space.
* A thread is lightweight.
* Cost of communication between the thread is low.

#### Note: At least one process is required for each thread.

## What is Thread in java

A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS, and one process can have multiple threads.

#### Note: At a time one thread is executed only.

## Java Thread class

Java provides **Thread class** to achieve thread programming. Thread class provides constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

|  |
| --- |
| Life cycle of a Thread (Thread States)  1. [Life cycle of a thread](https://www.javatpoint.com/life-cycle-of-a-thread)    1. [New](https://www.javatpoint.com/life-cycle-of-a-thread#threadstatenew)    2. [Runnable](https://www.javatpoint.com/life-cycle-of-a-thread#threadstaterunnable)    3. [Running](https://www.javatpoint.com/life-cycle-of-a-thread#threadstaterunning)    4. [Non-Runnable (Blocked)](https://www.javatpoint.com/life-cycle-of-a-thread#threadstateblocked)    5. [Terminated](https://www.javatpoint.com/life-cycle-of-a-thread#threadstateterminated)   A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.  But for better understanding the threads, we are explaining it in the 5 states.  The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:   1. New 2. Runnable 3. Running 4. Non-Runnable (Blocked) 5. Terminated    1) New The thread is in new state if you create an instance of Thread class but before the invocation of start() method. |

### 2) Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

### 3) Running

The thread is in running state if the thread scheduler has selected it.

### 4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

### 5) Terminated

A thread is in terminated or dead state when its run() method exits.

How to create thread

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

Thread class:

|  |
| --- |
| Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface. |

Commonly used Constructors of Thread class:

|  |
| --- |
| * Thread() * Thread(String name) * Thread(Runnable r) * Thread(Runnable r,String name) |

Commonly used methods of Thread class:

|  |
| --- |
| 1. **public void run():**is used to perform action for a thread. 2. **public void start():**starts the execution of the thread.JVM calls the run() method on the thread. 3. **public void sleep(long miliseconds):**Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds. 4. **public void join():**waits for a thread to die. 5. **public void join(long miliseconds):**waits for a thread to die for the specified miliseconds. 6. **public int getPriority():**returns the priority of the thread. 7. **public int setPriority(int priority):**changes the priority of the thread. 8. **public String getName():**returns the name of the thread. 9. **public void setName(String name):**changes the name of the thread. 10. **public Thread currentThread():**returns the reference of currently executing thread. 11. **public int getId():**returns the id of the thread. 12. **public Thread.State getState():**returns the state of the thread. 13. **public boolean isAlive():**tests if the thread is alive. 14. **public void yield():**causes the currently executing thread object to temporarily pause and allow other threads to execute. 15. **public void suspend():**is used to suspend the thread(depricated). 16. **public void resume():**is used to resume the suspended thread(depricated). 17. **public void stop():**is used to stop the thread(depricated). 18. **public boolean isDaemon():**tests if the thread is a daemon thread. 19. **public void setDaemon(boolean b):**marks the thread as daemon or user thread. 20. **public void interrupt():**interrupts the thread. 21. **public boolean isInterrupted():**tests if the thread has been interrupted. 22. **public static boolean interrupted():**tests if the current thread has been interrupted. |

Runnable interface:

|  |
| --- |
| The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run(). |

|  |
| --- |
| 1. **public void run():**is used to perform action for a thread. |

Starting a thread:

|  |
| --- |
| **start() method** of Thread class is used to start a newly created thread. It performs following tasks:   * A new thread starts(with new callstack). * The thread moves from New state to the Runnable state. * When the thread gets a chance to execute, its target run() method will run. |

1) Java Thread Example by extending Thread class

1. **class** Multi **extends** Thread{
2. **public** **void** run(){
3. System.out.println("thread is running...");
4. }
5. **public** **static** **void** main(String args[]){
6. Multi t1=**new** Multi();
7. t1.start();
8. }
9. }

Output:thread is running...

2) Java Thread Example by implementing Runnable interface

1. **class** Multi3 **implements** Runnable{
2. **public** **void** run(){
3. System.out.println("thread is running...");
4. }
6. **public** **static** **void** main(String args[]){
7. Multi3 m1=**new** Multi3();
8. Thread t1 =**new** Thread(m1);
9. t1.start();
10. }
11. }

Output:thread is running...

|  |
| --- |
| If you are not extending the Thread class,your class object would not be treated as a thread object.So you need to explicitely create Thread class object.We are passing the object of your class that implements Runnable so that your class run() method may execute. |

Thread Scheduler in Java

**Thread scheduler** in java is the part of the JVM that decides which thread should run.

There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.

Only one thread at a time can run in a single process.

The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

Sleep method in java

The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

Syntax of sleep() method in java

The Thread class provides two methods for sleeping a thread:

* public static void sleep(long miliseconds)throws InterruptedException
* public static void sleep(long miliseconds, int nanos)throws InterruptedException

Example of sleep method in java

1. **class** TestSleepMethod1 **extends** Thread{
2. **public** **void** run(){
3. **for**(**int** i=1;i<5;i++){
4. **try**{Thread.sleep(500);}**catch**(InterruptedException e){System.out.println(e);}
5. System.out.println(i);
6. }
7. }
8. **public** **static** **void** main(String args[]){
9. TestSleepMethod1 t1=**new** TestSleepMethod1();
10. TestSleepMethod1 t2=**new** TestSleepMethod1();
12. t1.start();
13. t2.start();
14. }
15. }

Output:

1

1

2

2

3

3

4

4

As you know well that at a time only one thread is executed. If you sleep a thread for the specified time,the thread shedular picks up another thread and so on.

# Can we start a thread twice

No. After starting a thread, it can never be started again. If you does so, an IllegalThreadStateException is thrown. In such case, thread will run once but for second time, it will throw exception.

Let's understand it by the example given below:

1. **public** **class** TestThreadTwice1 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("running...");
4. }
5. **public** **static** **void** main(String args[]){
6. TestThreadTwice1 t1=**new** TestThreadTwice1();
7. t1.start();
8. t1.start();
9. }
10. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestThreadTwice1)

running

Exception in thread "main" java.lang.IllegalThreadStateException

# What if we call run() method directly instead start() method?

|  |
| --- |
| * Each thread starts in a separate call stack. * Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack. |

1. **class** TestCallRun1 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("running...");
4. }
5. **public** **static** **void** main(String args[]){
6. TestCallRun1 t1=**new** TestCallRun1();
7. t1.run();//fine, but does not start a separate call stack
8. }
9. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCallRun1)

Output:running...

***Problem if you direct call run() method***

1. **class** TestCallRun2 **extends** Thread{
2. **public** **void** run(){
3. **for**(**int** i=1;i<5;i++){
4. **try**{Thread.sleep(500);}**catch**(InterruptedException e){System.out.println(e);}
5. System.out.println(i);
6. }
7. }
8. **public** **static** **void** main(String args[]){
9. TestCallRun2 t1=**new** TestCallRun2();
10. TestCallRun2 t2=**new** TestCallRun2();
12. t1.run();
13. t2.run();
14. }
15. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCallRun2)

Output:1

2

3

4

5

1

2

3

4

5

|  |
| --- |
| As you can see in the above program that there is no context-switching because here t1 and t2 will be treated as normal object not thread object. |

# The join() method

The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

### Syntax:

|  |
| --- |
| public void join()throws InterruptedException |
| public void join(long milliseconds)throws InterruptedException |

***Example of join() method***

1. **class** TestJoinMethod1 **extends** Thread{
2. **public** **void** run(){
3. **for**(**int** i=1;i<=5;i++){
4. **try**{
5. Thread.sleep(500);
6. }**catch**(Exception e){System.out.println(e);}
7. System.out.println(i);
8. }
9. }
10. **public** **static** **void** main(String args[]){
11. TestJoinMethod1 t1=**new** TestJoinMethod1();
12. TestJoinMethod1 t2=**new** TestJoinMethod1();
13. TestJoinMethod1 t3=**new** TestJoinMethod1();
14. t1.start();
15. **try**{
16. t1.join();
17. }**catch**(Exception e){System.out.println(e);}
19. t2.start();
20. t3.start();
21. }
22. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestJoinMethod1)

Output:1

2

3

4

5

1

1

2

2

3

3

4

4

5

5

|  |
| --- |
| As you can see in the above example,when t1 completes its task then t2 and t3 starts executing. |

***Example of join(long miliseconds) method***

1. **class** TestJoinMethod2 **extends** Thread{
2. **public** **void** run(){
3. **for**(**int** i=1;i<=5;i++){
4. **try**{
5. Thread.sleep(500);
6. }**catch**(Exception e){System.out.println(e);}
7. System.out.println(i);
8. }
9. }
10. **public** **static** **void** main(String args[]){
11. TestJoinMethod2 t1=**new** TestJoinMethod2();
12. TestJoinMethod2 t2=**new** TestJoinMethod2();
13. TestJoinMethod2 t3=**new** TestJoinMethod2();
14. t1.start();
15. **try**{
16. t1.join(1500);
17. }**catch**(Exception e){System.out.println(e);}
19. t2.start();
20. t3.start();
21. }
22. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestJoinMethod2)

Output:1

2

3

1

4

1

2

5

2

3

3

4

4

5

5

|  |
| --- |
| In the above example,when t1 is completes its task for 1500 miliseconds(3 times) then t2 and t3 starts executing. |

### getName(),setName(String) and getId() method:

|  |
| --- |
| public String getName() |
| public void setName(String name) |
| public long getId() |

1. **class** TestJoinMethod3 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("running...");
4. }
5. **public** **static** **void** main(String args[]){
6. TestJoinMethod3 t1=**new** TestJoinMethod3();
7. TestJoinMethod3 t2=**new** TestJoinMethod3();
8. System.out.println("Name of t1:"+t1.getName());
9. System.out.println("Name of t2:"+t2.getName());
10. System.out.println("id of t1:"+t1.getId());
12. t1.start();
13. t2.start();
15. t1.setName("Sonoo Jaiswal");
16. System.out.println("After changing name of t1:"+t1.getName());
17. }
18. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestJoinMethod3)

Output:Name of t1:Thread-0

Name of t2:Thread-1

id of t1:8

running...

After changling name of t1:Sonoo Jaiswal

running...

### The currentThread() method:

|  |
| --- |
| The currentThread() method returns a reference to the currently executing thread object. |

### Syntax:

|  |
| --- |
| public static Thread currentThread() |

***Example of currentThread() method***

1. **class** TestJoinMethod4 **extends** Thread{
2. **public** **void** run(){
3. System.out.println(Thread.currentThread().getName());
4. }
5. }
6. **public** **static** **void** main(String args[]){
7. TestJoinMethod4 t1=**new** TestJoinMethod4();
8. TestJoinMethod4 t2=**new** TestJoinMethod4();
10. t1.start();
11. t2.start();
12. }
13. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestJoinMethod4)

Output:Thread-0

Thread-1

# Naming Thread and Current Thread

## Naming Thread

The Thread class provides methods to change and get the name of a thread. By default, each thread has a name i.e. thread-0, thread-1 and so on. By we can change the name of the thread by using setName() method. The syntax of setName() and getName() methods are given below:

1. **public String getName():** is used to return the name of a thread.
2. **public void setName(String name):** is used to change the name of a thread.

## Example of naming a thread

1. **class** TestMultiNaming1 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("running...");
4. }
5. **public** **static** **void** main(String args[]){
6. TestMultiNaming1 t1=**new** TestMultiNaming1();
7. TestMultiNaming1 t2=**new** TestMultiNaming1();
8. System.out.println("Name of t1:"+t1.getName());
9. System.out.println("Name of t2:"+t2.getName());
11. t1.start();
12. t2.start();
14. t1.setName("Sonoo Jaiswal");
15. System.out.println("After changing name of t1:"+t1.getName());
16. }
17. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestMultiNaming1)

Output:Name of t1:Thread-0

Name of t2:Thread-1

id of t1:8

running...

After changeling name of t1:Sonoo Jaiswal

running...

## Current Thread

The currentThread() method returns a reference of currently executing thread.

1. **public** **static** Thread currentThread()

### Example of currentThread() method

1. **class** TestMultiNaming2 **extends** Thread{
2. **public** **void** run(){
3. System.out.println(Thread.currentThread().getName());
4. }
5. **public** **static** **void** main(String args[]){
6. TestMultiNaming2 t1=**new** TestMultiNaming2();
7. TestMultiNaming2 t2=**new** TestMultiNaming2();
9. t1.start();
10. t2.start();
11. }
12. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestMultiNaming2)

Output:Thread-0

Thread-1

# Priority of a Thread (Thread Priority):

|  |
| --- |
| Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. |

## 3 constants defined in Thread class:

|  |
| --- |
| 1. public static int MIN\_PRIORITY 2. public static int NORM\_PRIORITY 3. public static int MAX\_PRIORITY |

|  |
| --- |
| Default priority of a thread is 5 (NORM\_PRIORITY). The value of MIN\_PRIORITY is 1 and the value of MAX\_PRIORITY is 10. |

### Example of priority of a Thread:

1. **class** TestMultiPriority1 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("running thread name is:"+Thread.currentThread().getName());
4. System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
6. }
7. **public** **static** **void** main(String args[]){
8. TestMultiPriority1 m1=**new** TestMultiPriority1();
9. TestMultiPriority1 m2=**new** TestMultiPriority1();
10. m1.setPriority(Thread.MIN\_PRIORITY);
11. m2.setPriority(Thread.MAX\_PRIORITY);
12. m1.start();
13. m2.start();
15. }
16. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestMultiPriority1)

Output:running thread name is:Thread-0

running thread priority is:10

running thread name is:Thread-1

running thread priority is:1

# Daemon Thread in Java

**Daemon thread in java** is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.

There are many java daemon threads running automatically e.g. gc, finalizer etc.

You can see all the detail by typing the jconsole in the command prompt. The jconsole tool provides information about the loaded classes, memory usage, running threads etc.

## Points to remember for Daemon Thread in Java

* It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
* Its life depends on user threads.
* It is a low priority thread.

### Why JVM terminates the daemon thread if there is no user thread?

The sole purpose of the daemon thread is that it provides services to user thread for background supporting task. If there is no user thread, why should JVM keep running this thread. That is why JVM terminates the daemon thread if there is no user thread.

### Methods for Java Daemon thread by Thread class

The java.lang.Thread class provides two methods for java daemon thread.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1) | public void setDaemon(boolean status) | is used to mark the current thread as daemon thread or user thread. |
| 2) | public boolean isDaemon() | is used to check that current is daemon. |

### Simple example of Daemon thread in java

*File: MyThread.java*

1. **public** **class** TestDaemonThread1 **extends** Thread{
2. **public** **void** run(){
3. **if**(Thread.currentThread().isDaemon()){//checking for daemon thread
4. System.out.println("daemon thread work");
5. }
6. **else**{
7. System.out.println("user thread work");
8. }
9. }
10. **public** **static** **void** main(String[] args){
11. TestDaemonThread1 t1=**new** TestDaemonThread1();//creating thread
12. TestDaemonThread1 t2=**new** TestDaemonThread1();
13. TestDaemonThread1 t3=**new** TestDaemonThread1();
15. t1.setDaemon(**true**);//now t1 is daemon thread
17. t1.start();//starting threads
18. t2.start();
19. t3.start();
20. }
21. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestDaemonThread1)

#### Output

daemon thread work

user thread work

user thread work

#### Note: If you want to make a user thread as Daemon, it must not be started otherwise it will throw IllegalThreadStateException.

*File: MyThread.java*

1. **class** TestDaemonThread2 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("Name: "+Thread.currentThread().getName());
4. System.out.println("Daemon: "+Thread.currentThread().isDaemon());
5. }
7. **public** **static** **void** main(String[] args){
8. TestDaemonThread2 t1=**new** TestDaemonThread2();
9. TestDaemonThread2 t2=**new** TestDaemonThread2();
10. t1.start();
11. t1.setDaemon(**true**);//will throw exception here
12. t2.start();
13. }
14. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestDaemonThread2)

Output:exception in thread main: java.lang.IllegalThreadStateException

# Synchronized in Java

[Multi-threaded](http://quiz.geeksforgeeks.org/multithreading-in-java/)programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.

So it needs to be made sure by some synchronization method that only one thread can access the resource at a given point of time.

Java provides a way of creating threads and synchronizing their task by using synchronized blocks. Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block in Java is synchronized on some object. All synchronized blocks synchronized on the same object can only have one thread executing inside them at a time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

Following is the general form of a synchronized block:

// Only one thread can execute at a time.

// sync\_object is a reference to an object

// whose lock associates with the [monitor](http://quiz.geeksforgeeks.org/monitors/).

// The code is said to be synchronized on

// the monitor object

synchronized(sync\_object)

{

// Access shared variables and other

// shared resources

}

This synchronization is implemented in Java with a concept called monitors. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.

Following is an example of multi threading with synchronized.

|  |
| --- |
| // A Java program to demonstrate working of  // synchronized.  import java.io.\*;  import java.util.\*;    // A Class used to send a message  class Sender  {      public void send(String msg)      {          System.out.println("Sending\t"  + msg );          try          {              Thread.sleep(1000);          }          catch (Exception e)          {              System.out.println("Thread  interrupted.");          }          System.out.println("\n" + msg + "Sent");      }  }    // Class for send a message using Threads  class ThreadedSend extends Thread  {      private String msg;      Sender  sender;        // Recieves a message object and a string      // message to be sent      ThreadedSend(String m,  Sender obj)      {          msg = m;          sender = obj;      }        public void run()      {          // Only one thread can send a message          // at a time.          synchronized(sender)          {              // synchronizing the snd object              sender.send(msg);          }      }  }    // Driver class  class SyncDemo  {      public static void main(String args[])      {          Sender snd = new Sender();          ThreadedSend S1 =              new ThreadedSend( " Hi " , snd );          ThreadedSend S2 =              new ThreadedSend( " Bye " , snd );            // Start two threads of ThreadedSend type          S1.start();          S2.start();            // wait for threads to end          try          {              S1.join();              S2.join();          }          catch(Exception e)          {              System.out.println("Interrupted");          }      }  } |

Output:

Sending Hi

Hi Sent

Sending Bye

Bye Sent

The output is same every-time we run the program.

In the above example, we chose to synchronize the Sender object inside the run() method of the ThreadedSend class. Alternately, we could define the **whole send() block as synchronized**and it would produce the same result. Then we don’t have to synchronize the Message object inside the run() method in ThreadedSend class.

|  |
| --- |
| // An alternate implementation to demonstrate  // that we can use synchronized with method also.  class Sender  {      public synchronized void send(String msg)      {          System.out.println("Sending\t" + msg );          try          {              Thread.sleep(1000);          }          catch (Exception e)          {              System.out.println("Thread interrupted.");          }          System.out.println("\n" + msg + "Sent");      }  } |

We do not always have to synchronize a whole method. Sometimes it is preferable to **synchronize only part of a method**. Java synchronized blocks inside methods makes this possible.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // One more alternate implementation to demonstrate  // that synchronized can be used with only a part of  // method  class Sender  {      public void send(String msg)      {          synchronized(this)          {              System.out.println("Sending\t" + msg );              try              {                  Thread.sleep(1000);              }              catch (Exception e)              {                  System.out.println("Thread interrupted.");              }              System.out.println("\n" + msg + "Sent");          }      }  } |

# Java Date and Time

The java.time, java.util, java.sql and java.text packages contains classes for representing date and time. Following classes are important for dealing with date in java.

## Java 8 Date/Time API

Java has introduced a new Date and Time API since Java 8. The java.time package contains Java 8 Date and Time classes.

* [java.time.LocalDate class](https://www.javatpoint.com/java-localdate)
* [java.time.LocalTime class](https://www.javatpoint.com/java-localtime)
* [java.time.LocalDateTime class](https://www.javatpoint.com/java-localdatetime)
* [java.time.MonthDay class](https://www.javatpoint.com/java-monthday)
* [java.time.OffsetTime class](https://www.javatpoint.com/java-offsettime)
* [java.time.OffsetDateTime class](https://www.javatpoint.com/java-offsetdatetime)
* [java.time.Clock class](https://www.javatpoint.com/java-clock)
* [java.time.ZonedDateTime class](https://www.javatpoint.com/java-zoneddatetime)
* [java.time.ZoneId class](https://www.javatpoint.com/java-zoneid)
* [java.time.ZoneOffset class](https://www.javatpoint.com/java-zoneoffset)
* [java.time.Year class](https://www.javatpoint.com/java-year)
* [java.time.YearMonth class](https://www.javatpoint.com/java-yearmonth)
* [java.time.Period class](https://www.javatpoint.com/java-period)
* [java.time.Duration class](https://www.javatpoint.com/java-duration)
* [java.time.Instant class](https://www.javatpoint.com/java-instant)
* [java.time.DayOfWeek enum](https://www.javatpoint.com/java-dayofweek-enum)
* [java.time.Month enum](https://www.javatpoint.com/java-month-enum)

## Classical Date/Time API

But classical or old Java Date API is also useful. Let's see the list of classical Date and Time classes.

* [java.util.Date class](https://www.javatpoint.com/java-util-date)
* [java.sql.Date class](https://www.javatpoint.com/java-sql-date)
* [java.util.Calendar class](https://www.javatpoint.com/java-util-calendar)
* java.util.GregorianCalendar class
* [java.util.TimeZone class](https://www.javatpoint.com/java-util-timezone)
* java.sql.Time class
* java.sql.Timestamp class

## Formatting Date and Time

We can format date and time in java by the use of following classes:

* [java.text.DateFormat class](https://www.javatpoint.com/java-date-format)
* [java.text.SimpleDateFormat class](https://www.javatpoint.com/java-simpledateformat)

# Java LocalDate class

Java LocalDate class is an immutable class that represents Date with a default format of yyyy-MM-dd. It inherits Object class and implements the ChronoLocalDate interface

## Java LocalDate class declaration

Let's see the declaration of java.time.LocalDate class.

1. **public** **final** **class** LocalDate **extends** Object
2. **implements** Temporal, TemporalAdjuster, ChronoLocalDate, Serializable

### Methods of Java LocalDate

|  |  |
| --- | --- |
| **Method** | **Description** |
| LocalDateTime atTime(int hour, int minute) | It is used to combine this date with a time to create a LocalDateTime. |
| int compareTo(ChronoLocalDate other) | It is used to compares this date to another date. |
| boolean equals(Object obj) | It is used to check if this date is equal to another date. |
| String format(DateTimeFormatter formatter) | It is used to format this date using the specified formatter. |
| int get(TemporalField field) | It is used to get the value of the specified field from this date as an int. |
| boolean isLeapYear() | It is used to check if the year is a leap year, according to the ISO proleptic calendar system rules. |
| LocalDate minusDays(long daysToSubtract) | It is used to return a copy of this LocalDate with the specified number of days subtracted. |
| LocalDate minusMonths(long monthsToSubtract) | It is used to return a copy of this LocalDate with the specified number of months subtracted. |
| static LocalDate now() | It is used to obtain the current date from the system clock in the default time-zone. |
| LocalDate plusDays(long daysToAdd) | It is used to return a copy of this LocalDate with the specified number of days added. |
| LocalDate plusMonths(long monthsToAdd) | It is used to return a copy of this LocalDate with the specified number of months added. |

## Java LocalDate Example

1. **import** java.time.LocalDate;
2. **public** **class** LocalDateExample1 {
3. **public** **static** **void** main(String[] args) {
4. LocalDate date = LocalDate.now();
5. LocalDate yesterday = date.minusDays(1);
6. LocalDate tomorrow = yesterday.plusDays(2);
7. System.out.println("Today date: "+date);
8. System.out.println("Yesterday date: "+yesterday);
9. System.out.println("Tommorow date: "+tomorrow);
10. }
11. }

**[Test it Now](https://compiler.javatpoint.com/opr/test.jsp?filename=LocalDateExample1" \t "_blank)**

Output:

Today date: 2017-01-13

Yesterday date: 2017-01-12

Tommorow date: 2017-01-14

## Java LocalDate Example: isLeapYear()

1. **import** java.time.LocalDate;
2. **public** **class** LocalDateExample2 {
3. **public** **static** **void** main(String[] args) {
4. LocalDate date1 = LocalDate.of(2017, 1, 13);
5. System.out.println(date1.isLeapYear());
6. LocalDate date2 = LocalDate.of(2016, 9, 23);
7. System.out.println(date2.isLeapYear());
8. }
9. }

**[Test it Now](https://compiler.javatpoint.com/opr/test.jsp?filename=LocalDateExample2" \t "_blank)**

Output:

false

true

## Java LocalDate Example: atTime()

1. **import** java.time.\*;
2. **public** **class** LocalDateExample3 {
3. **public** **static** **void** main(String[] args) {
4. LocalDate date = LocalDate.of(2017, 1, 13);
5. LocalDateTime datetime = date.atTime(1,50,9);
6. System.out.println(datetime);
7. }
8. }

**[Test it Now](https://compiler.javatpoint.com/opr/test.jsp?filename=LocalDateExample3" \t "_blank)**

Output:

2017-01-13T01:50:09

# Collections in Java

A Collection is a group of individual objects represented as a single unit. Java provides Collection Framework which defines several classes and interfaces to represent a group of objects as a single unit.

The Collection interface (**java.util.Collection**) and Map interface (**java.util.Map**) are the two main “root” interfaces of Java collection classes.

**Need for Collection Framework :**  
Before Collection Framework (or before JDK 1.2) was introduced, the standard methods for grouping Java objects (or collections) were Arrays or Vectors or Hashtables. All of these collections had no common interface.

Accessing elements of these Data Structures was a hassle as each had a different method (and syntax) for accessing its members:

|  |
| --- |
| // Java program to show why collection framework was needed  import java.io.\*;  import java.util.\*;    class Test  {      public static void main (String[] args)      {          // Creating instances of array, vector and hashtable          int arr[] = new int[] {1, 2, 3, 4};          Vector<Integer> v = new Vector();          Hashtable<Integer, String> h = new Hashtable();          v.addElement(1);          v.addElement(2);          h.put(1,"geeks");          h.put(2,"4geeks");            // Array instance creation requires [], while Vector          // and hastable require ()          // Vector element insertion requires addElement(), but          // hashtable element insertion requires put()            // Accessing first element of array, vector and hashtable          System.out.println(arr[0]);          System.out.println(v.elementAt(0));          System.out.println(h.get(1));            // Array elements are accessed using [], vector elements          // using elementAt() and hashtable elements using get()      }  } |

Output:

1

1

geeks

As we can see, none of these collections (Array, Vector or Hashtable) implement a standard member access interface. It was very difficult for programmers to write algorithms that can work for all kinds of Collections. Another drawback being that most of the ‘Vector’ methods are final, meaning we cannot extend the ’Vector’ class to implement a similar kind of Collection.  
***Java developers decided to come up with a common interface to deal with the above mentioned problems and introduced the Collection Framework in JDK 1.2***.

Both legacy Vectors and Hashtables were modified to conform to the Collection Framework.

**Advantages of Collection Framework:**

1. Consistent API : The API has a basic set of interfaces like Collection, Set, List, or Map. All classes (ArrayList, LinkedList, Vector, etc) that implement these interfaces have *some* common set of methods.
2. Reduces programming effort: A programmer doesn’t have to worry about the design of Collection, and he can focus on its best use in his program.
3. Increases program speed and quality: Increases performance by providing high-performance implementations of useful data structures and algorithms.

**Hierarchy of Collection Framework**

Collection Map

/ / \ \ |

/ / \ \ |

Set List Queue Dequeue SortedMap

/

/

SortedSet

**Core Interfaces in Collections**

Note that this diagram only shows core interfaces.

**Collection :** Root interface with basic methods like add(), remove(),

contains(), isEmpty(), addAll(), ... etc.

**[Set](https://www.geeksforgeeks.org/set-in-java/) :** Doesn't allow duplicates. Example implementations of Set

interface are HashSet (Hashing based) and TreeSet (balanced

BST based). Note that TreeSet implements **SortedSet**.

**[List](https://www.geeksforgeeks.org/list-interface-java-examples/) :** Can contain duplicates and elements are ordered. Example

implementations are LinkedList (linked list based) and

[ArrayList](https://www.geeksforgeeks.org/array-vs-arraylist-in-java/) (dynamic array based)

**[Queue](https://www.geeksforgeeks.org/queue-interface-java/) :** Typically order elements in FIFO order except exceptions

like PriorityQueue.

**[Deque](https://www.geeksforgeeks.org/deque-interface-java-example/) :** Elements can be inserted and removed at both ends. Allows

both LIFO and FIFO.

**[Map](https://www.geeksforgeeks.org/map-interface-java-examples/) :** Contains Key value pairs. Doesn't allow duplicates. Example

implementation are [HashMap](http://www.geeksforgeeks.org/java-util-hashmap-in-java/) and TreeMap.

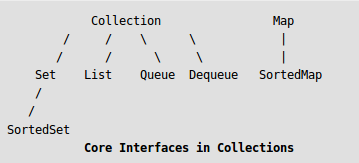
[TreeMap](https://www.geeksforgeeks.org/treemap-in-java/) implements **SortedMap**.

The difference between Set and Map interface is that in Set we

have only keys, whereas in Map, we have key, value pairs.

# List Interface in Java with Examples

The Java.util.List is a child interface of [Collection](https://www.geeksforgeeks.org/collections-in-java-2/). It is an ordered collection of objects in which duplicate values can be stored. Since List preserves the insertion order, it allows positional access and insertion of elements. List Interface is implemented by the classes of [ArrayList](https://www.geeksforgeeks.org/arraylist-in-java/), [LinkedList](https://www.geeksforgeeks.org/linked-list-in-java/), [Vector](https://www.geeksforgeeks.org/java-util-vector-class-java/) and [Stack](https://www.geeksforgeeks.org/stack-class-in-java/).



**Declaration:**

public abstract interface List extends Collection

**Creating List Objects:**  
List is an interface, and the instances of List can be created by implementing various classes in the following ways:

List a = new ArrayList();

List b = new LinkedList();

List c = new Vector();

List d = new Stack();

**Generic List Object:**  
After the introduction of Generics in Java 1.5, it is possible to restrict the type of object that can be stored in the List. The type-safe List can be defined in the following way:

// Obj is the type of object to be stored in List

List<Obj> list = new ArrayList<Obj> ();

**Operations on List:**  
List Interface extends Collection, hence it supports all the operations of Collection Interface, along with following additional operations:

1. **Positional Access:**  
   List allows add, remove, get and set operations based on numerical positions of elements in List. List provides following methods for these operations:
   * **[void add(int index, Object O):](https://www.geeksforgeeks.org/list-addint-index-e-element-method-in-java/" \t "_blank)** This method adds given element at specified index.
   * **[boolean addAll(int index, Collection c):](https://www.geeksforgeeks.org/list-addall-method-in-java-with-examples/" \t "_blank)** This method adds all elements from specified collection to list. First element gets inserted at given index. If there is already an element at that position, that element and other subsequent elements(if any) are shifted to the right by increasing their index.
   * **[Object remove(int index):](https://www.geeksforgeeks.org/list-removeint-index-method-in-java-with-examples/" \t "_blank)** This method removes an element from the specified index. It shifts subsequent elements(if any) to left and decreases their indexes by 1.
   * **[Object get(int index):](https://www.geeksforgeeks.org/list-get-method-in-java-with-examples/" \t "_blank)** This method returns element at the specified index.
   * **[Object set(int index, Object new):](https://www.geeksforgeeks.org/arraylist-set-method-in-java-with-examples/" \t "_blank)** This method replaces element at given index with new element. This function returns the element which was just replaced by new element.

|  |
| --- |
| // Java program to demonstrate positional access  // operations on List interface  import java.util.\*;    public class ListDemo {      public static void main(String[] args)      {          // Creating a list          List<Integer> l1 = new ArrayList<Integer>();          l1.add(0, 1); // adds 1 at 0 index          l1.add(1, 2); // adds 2 at 1 index          System.out.println(l1); // [1, 2]            // Creating another list          List<Integer> l2 = new ArrayList<Integer>();          l2.add(1);          l2.add(2);          l2.add(3);            // Will add list l2 from 1 index          l1.addAll(1, l2);          System.out.println(l1);            // Removes element from index 1          l1.remove(1);          System.out.println(l1); // [1, 2, 3, 2]            // Prints element at index 3          System.out.println(l1.get(3));            // Replace 0th element with 5          l1.set(0, 5);          System.out.println(l1);      }  } |

**Output:**

[1, 2]

[1, 1, 2, 3, 2]

[1, 2, 3, 2]

2

[5, 2, 3, 2]

1. **Search:**  
   List provides methods to search element and returns its numeric position. Following two methods are supported by List for this operation:
   * **[int indexOf(Object o):](https://www.geeksforgeeks.org/list-indexof-method-in-java-with-examples/" \t "_blank)** This method returns first occurrence of given element or -1 if element is not present in list.
   * **[int lastIndexOf(Object o):](https://www.geeksforgeeks.org/list-lastindexof-method-in-java-with-examples/" \t "_blank)**This method returns the last occurrence of given element or -1 if element is not present in list.

|  |
| --- |
| // Java program to demonstrate search  // operations on List interface    import java.util.\*;    public class ListDemo {      public static void main(String[] args)      {          // Type safe array list, stores only string          List<String> l = new ArrayList<String>(5);          l.add("Geeks");          l.add("for");          l.add("Geeks");            // Using indexOf() and lastIndexOf()          System.out.println("first index of Geeks:"                             + l.indexOf("Geeks"));          System.out.println("last index of Geeks:"                             + l.lastIndexOf("Geeks"));          System.out.println("Index of element"                             + " not present : "                             + l.indexOf("Hello"));      }  } |

**Output:**

first index of Geeks:0

last index of Geeks:2

Index of element not present : -1

1. **Iteration:**
   * **[ListIterator(extends Iterator)](https://www.geeksforgeeks.org/list-listiterator-method-in-java-with-examples/" \t "_blank)** is used to iterate over List element. List iterator is bidirectional iterator. For more details about ListIterator refer [Iterators in Java](https://www.geeksforgeeks.org/iterators-in-java/).
2. **Range-view:**  
   List Interface provides a method to get the List view of the portion of given List between two indices. Following is the method supported by List for range view operation.
   * **[List subList(int fromIndex, int toIndex):](https://www.geeksforgeeks.org/list-sublist-method-in-java-with-examples/" \t "_blank)**This method returns List view of specified List between fromIndex(inclusive) and toIndex(exclusive).

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java program to demonstrate subList operation  // on List interface.  import java.util.\*;  public class ListDemo {      public static void main(String[] args)      {          // Type safe array list, stores only string          List<String> l = new ArrayList<String>(5);            l.add("GeeksforGeeks");          l.add("Practice");          l.add("GeeksQuiz");          l.add("IDE");          l.add("Courses");            List<String> range = new ArrayList<String>();            // Return List between 2nd(including)          // and 4th element(excluding)          range = l.subList(2, 4);            System.out.println(range);      }  } |

**Output:**

[GeeksQuiz, IDE]

# LinkedList in Java

Linked List are linear data structures where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node. Due to the dynamicity and ease of insertions and deletions, they are preferred over the arrays. It also has few disadvantages like the nodes cannot be accessed directly instead we need to start from the head and follow through the link to reach to a node we wish to access.  
To store the elements in a linked list we use a doubly linked list which provides a linear data structure and also used to inherit an abstract class and implement list and deque interfaces.

In Java, LinkedList class implements the [list interface](https://www.geeksforgeeks.org/list-interface-java-examples/). The LinkedList class also consists of various constructors and methods like other java collections.

**Constructors for Java LinkedList:**

1. LinkedList(): Used to create an empty linked list.
2. LinkedList(Collection C): Used to create a ordered list which contains all the elements of a specified collection, as returned by the collection’s iterator.

|  |
| --- |
| // Java code for Linked List implementation    import java.util.\*;    public class Test  {      public static void main(String args[])      {          // Creating object of class linked list          LinkedList<String> object = new LinkedList<String>();            // Adding elements to the linked list          object.add("A");          object.add("B");          object.addLast("C");          object.addFirst("D");          object.add(2, "E");          object.add("F");          object.add("G");          System.out.println("Linked list : " + object);            // Removing elements from the linked list          object.remove("B");          object.remove(3);          object.removeFirst();          object.removeLast();          System.out.println("Linked list after deletion: " + object);            // Finding elements in the linked list          boolean status = object.contains("E");            if(status)              System.out.println("List contains the element 'E' ");          else              System.out.println("List doesn't contain the element 'E'");            // Number of elements in the linked list          int size = object.size();          System.out.println("Size of linked list = " + size);            // Get and set elements from linked list          Object element = object.get(2);          System.out.println("Element returned by get() : " + element);          object.set(2, "Y");          System.out.println("Linked list after change : " + object);      }  } |

**Output:**

Linked list : [D, A, E, B, C, F, G]

Linked list after deletion: [A, E, F]

List contains the element 'E'

Size of linked list = 3

Element returned by get() : F

Linked list after change : [A, E, Y]

Set in Java

* Set is an interface which extends Collection. It is an unordered collection of objects in which duplicate values cannot be stored.
* Basically, Set is implemented by HashSet, LinkedHashSet or TreeSet (sorted representation).
* Set has various methods to add, remove clear, size, etc to enhance the usage of this interface

|  |
| --- |
| // Java code for adding elements in Set  import java.util.\*;  public class Set\_example  {      public static void main(String[] args)      {          // Set deonstration using HashSet          Set<String> hash\_Set = new HashSet<String>();          hash\_Set.add("Geeks");          hash\_Set.add("For");          hash\_Set.add("Geeks");          hash\_Set.add("Example");          hash\_Set.add("Set");          System.out.print("Set output without the duplicates");            System.out.println(hash\_Set);            // Set deonstration using TreeSet          System.out.print("Sorted Set after passing into TreeSet");          Set<String> tree\_Set = new TreeSet<String>(hash\_Set);          System.out.println(tree\_Set);      }  } |

(Please note that we have entered a duplicate entity but it is not displayed in the output. Also, we can directly sort the entries by passing the unordered Set in as the parameter of TreeSet).

**Output:**

Set output without the duplicates[Set, Example, Geeks, for]

Sorted Set after passing into TreeSet[Example, For, Geeks, Set]

**Note:** As we can see the duplicate entry “Geeks” is ignored in the final output, Set interface doesn’t allow duplicate entries.

Now we will see some of the basic operations on the Set i.e. Union, Intersection and Difference.

Let’s take an example of two integer Sets:

* [1, 3, 2, 4, 8, 9, 0]
* [1, 3, 7, 5, 4, 0, 7, 5]

**Union**  
In this, we could simply add one Set with other. Since the Set will itself not allow any duplicate entries, we need not take care of the common values.

**Expected Output:**

Union : [0, 1, 2, 3, 4, 5, 7, 8, 9]

**Intersection**  
We just need to retain the common values from both Sets.

**Expected Output:**

Intersection : [0, 1, 3, 4]

**Difference**  
We just need to remove all the values of one Set from the other.  
 **Expected Output:**

Difference : [2, 8, 9]

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java code for demonstrating union, intersection and difference  // on Set  import java.util.\*;  public class Set\_example  {      public static void main(String args[])      {          Set<Integer> a = new HashSet<Integer>();          a.addAll(Arrays.asList(new Integer[] {1, 3, 2, 4, 8, 9, 0}));          Set<Integer> b = new HashSet<Integer>();          b.addAll(Arrays.asList(new Integer[] {1, 3, 7, 5, 4, 0, 7, 5}));            // To find union          Set<Integer> union = new HashSet<Integer>(a);          union.addAll(b);          System.out.print("Union of the two Set");          System.out.println(union);            // To find intersection          Set<Integer> intersection = new HashSet<Integer>(a);          intersection.retainAll(b);          System.out.print("Intersection of the two Set");          System.out.println(intersection);            // To find the symmetric difference          Set<Integer> difference = new HashSet<Integer>(a);          difference.removeAll(b);          System.out.print("Difference of the two Set");          System.out.println(difference);      }  } |

**Output:**

Union of the two Set[0, 1, 2, 3, 4, 5, 7, 8, 9]

Intersection of the two Set[0, 1, 3, 4]

Difference of the two Set[2, 8, 9]

Java HashSet

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.
* HashSet allows null value.
* HashSet class is non synchronized.
* HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
* HashSet is the best approach for search operations.
* The initial default capacity of HashSet is 16, and the load factor is 0.75.

Difference between List and Set

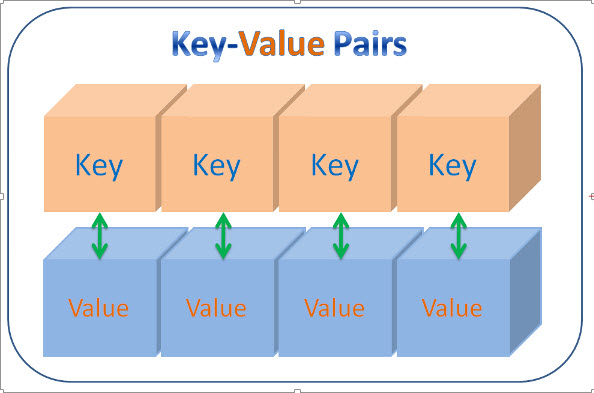
A list can contain duplicate elements whereas Set contains unique elements only.

**What is Hashmap in Java?**

A HashMap basically designates **unique keys** to corresponding **values** that can be retrieved at any given point. 

**Features of Java Hashmap**

a) The **values** can be stored in a map by forming a **key-value** pair. The value can be retrieved using the key by passing it to the correct method.  
  
b) If **no element** exists in the Map, it will throw a ‘**NoSuchElementException’**.  
  
c) HashMap stores only **object** **references**. That is why, it is impossible to use **primitive data types** like double or int. Use wrapper class (like Integer or Double) instead.

[](https://www.guru99.com/images/uploads/2012/12/java11.jpg)

**Using HashMaps in Java Programs:**

Following are the two ways to declare a Hash Map:

HashMap<String, Object> map = new HashMap<String, Object>();

HashMap x = new HashMap();

**Important Hashmap Methods**

* **get(Object KEY)** – This will return the value associated with a specified key in this Java hashmap.
* **put(Object KEY, String VALUE)** – This method stores the specified value and associates it with the specified key in this map.

**Java Hashmap Example**

**Following is a sample implementation of java Hash Map:**

import java.util.HashMap;

import java.util.Map;

public class Sample\_TestMaps{

public static void main(String[] args){

Map<String, String> objMap = new HashMap<String, String>();

objMap.put("Name", "Suzuki");

objMap.put("Power", "220");

objMap.put("Type", "2-wheeler");

objMap.put("Price", "85000");

System.out.println("Elements of the Map:");

System.out.println(objMap);

}

}

**Output:**

Elements of the Map:

{Type=2-wheeler, Price=85000, Power=220, Name=Suzuki}

**Example 2: Remove a value from HashMap based on key**

import java.util.\*;

public class HashMapExample {

public static void main(String args[]) {

// create and populate hash map

HashMap<Integer, String> map = new HashMap<Integer, String>();

map.put(1,"Java");

map.put(2, "Python");

map.put(3, "PHP");

map.put(4, "SQL");

map.put(5, "C++");

System.out.println("Tutorial in Guru99: "+ map);

// Remove value of key 5

map.remove(5);

System.out.println("Tutorial in Guru99 After Remove: "+ map);

}

}

**Java I/O**

**Java I/O** (Input and Output) is used *to process the input* and *produce the output*.

Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

We can perform **file handling in Java** by Java I/O API.

Stream

A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.

In Java, 3 streams are created for us automatically. All these streams are attached with the console.

**1) System.out:**standard output stream

**2) System.in:**standard input stream

**3) System.err:**standard error stream

Let's see the code to print **output and an error** message to the console.

1. System.out.println("simple message");
2. System.err.println("error message");

Let's see the code to get **input** from console.

1. **int** i=System.in.read();//returns ASCII code of 1st character
2. System.out.println((**char**)i);//will print the character

OutputStream vs InputStream

The explanation of OutputStream and InputStream classes are given below:

OutputStream

Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.

InputStream

Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

Let's understand the working of Java OutputStream and InputStream by the figure given below.

OutputStream class

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

Useful methods of OutputStream

|  |  |
| --- | --- |
| **Method** | **Description** |
| 1) public void write(int)throws IOException | is used to write a byte to the current output stream. |
| 2) public void write(byte[])throws IOException | is used to write an array of byte to the current output stream. |
| 3) public void flush()throws IOException | flushes the current output stream. |
| 4) public void close()throws IOException | is used to close the current output stream. |

OutputStream Hierarchy

InputStream class

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

Useful methods of InputStream

|  |  |
| --- | --- |
| **Method** | **Description** |
| 1) public abstract int read()throws IOException | reads the next byte of data from the input stream. It returns -1 at the end of the file. |
| 2) public int available()throws IOException | returns an estimate of the number of bytes that can be read from the current input stream. |
| 3) public void close()throws IOException | is used to close the current input stream. |

## Stream

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

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



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

### Byte Streams

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

**Example**

import java.io.\*;

public class CopyFile {

public static void main(String args[]) throws IOException {

FileInputStream in = null;

FileOutputStream out = null;

try {

in = new FileInputStream("input.txt");

out = new FileOutputStream("output.txt");

int c;

while ((c = in.read()) != -1) {

out.write(c);

}

}finally {

if (in != null) {

in.close();

}

if (out != null) {

out.close();

}

}

}

}

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

This is test for copy file.

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

$javac CopyFile.java

$java CopyFile

### Character Streams

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

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

**Example**

import java.io.\*;

public class CopyFile {

public static void main(String args[]) throws IOException {

FileReader in = null;

FileWriter out = null;

try {

in = new FileReader("input.txt");

out = new FileWriter("output.txt");

int c;

while ((c = in.read()) != -1) {

out.write(c);

}

}finally {

if (in != null) {

in.close();

}

if (out != null) {

out.close();

}

}

}

}

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

This is test for copy file.

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

$javac CopyFile.java

$java CopyFile

## Standard Streams

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

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

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

**Example**

[Live Demo](http://tpcg.io/lVH2u1)

import java.io.\*;

public class ReadConsole {

public static void main(String args[]) throws IOException {

InputStreamReader cin = null;

try {

cin = new InputStreamReader(System.in);

System.out.println("Enter characters, 'q' to quit.");

char c;

do {

c = (char) cin.read();

System.out.print(c);

} while(c != 'q');

}finally {

if (cin != null) {

cin.close();

}

}

}

}

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

$javac ReadConsole.java

$java ReadConsole

Enter characters, 'q' to quit.

1

1

e

e

q

q

## Reading and Writing Files

As described earlier, a stream can be defined as a sequence of data. The **InputStream** is used to read data from a source and the **OutputStream** is used for writing data to a destination.

Here is a hierarchy of classes to deal with Input and Output streams.



The two important streams are **FileInputStream** and **FileOutputStream**, which would be discussed in this tutorial.

### FileInputStream

This stream is used for reading data from the files. Objects can be created using the keyword **new** and there are several types of constructors available.

Following constructor takes a file name as a string to create an input stream object to read the file −

InputStream f = new FileInputStream("C:/java/hello");

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows −

File f = new File("C:/java/hello");

InputStream f = new FileInputStream(f);

Once you have *InputStream* object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **public void close() throws IOException{}**  This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException. |
| 2 | **protected void finalize()throws IOException {}**  This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException. |
| 3 | **public int read(int r)throws IOException{}**  This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's the end of the file. |
| 4 | **public int read(byte[] r) throws IOException{}**  This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If it is the end of the file, -1 will be returned. |
| 5 | **public int available() throws IOException{}**  Gives the number of bytes that can be read from this file input stream. Returns an int. |

There are other important input streams available, for more detail you can refer to the following links −

* [ByteArrayInputStream](https://www.tutorialspoint.com/java/java_bytearrayinputstream.htm)
* [DataInputStream](https://www.tutorialspoint.com/java/java_datainputstream.htm)

## FileOutputStream

FileOutputStream is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output.

Here are two constructors which can be used to create a FileOutputStream object.

Following constructor takes a file name as a string to create an input stream object to write the file −

OutputStream f = new FileOutputStream("C:/java/hello")

Following constructor takes a file object to create an output stream object to write the file. First, we create a file object using File() method as follows −

File f = new File("C:/java/hello");

OutputStream f = new FileOutputStream(f);

Once you have *OutputStream* object in hand, then there is a list of helper methods, which can be used to write to stream or to do other operations on the stream.

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **public void close() throws IOException{}**  This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException. |
| 2 | **protected void finalize()throws IOException {}**  This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException. |
| 3 | **public void write(int w)throws IOException{}**  This methods writes the specified byte to the output stream. |
| 4 | **public void write(byte[] w)**  Writes w.length bytes from the mentioned byte array to the OutputStream. |

There are other important output streams available, for more detail you can refer to the following links −

* [ByteArrayOutputStream](https://www.tutorialspoint.com/java/java_bytearrayoutputstream.htm)
* [DataOutputStream](https://www.tutorialspoint.com/java/java_dataoutputstream.htm)

**Example**

Following is the example to demonstrate InputStream and OutputStream −

import java.io.\*;

public class fileStreamTest {

public static void main(String args[]) {

try {

byte bWrite [] = {11,21,3,40,5};

OutputStream os = new FileOutputStream("test.txt");

for(int x = 0; x < bWrite.length ; x++) {

os.write( bWrite[x] ); // writes the bytes

}

os.close();

InputStream is = new FileInputStream("test.txt");

int size = is.available();

for(int i = 0; i < size; i++) {

System.out.print((char)is.read() + " ");

}

is.close();

} catch (IOException e) {

System.out.print("Exception");

}

}

}

The above code would create file test.txt and would write given numbers in binary format. Same would be the output on the stdout screen.

## File Navigation and I/O

There are several other classes that we would be going through to get to know the basics of File Navigation and I/O.

* [File Class](https://www.tutorialspoint.com/java/java_file_class.htm)
* [FileReader Class](https://www.tutorialspoint.com/java/java_filereader_class.htm)
* [FileWriter Class](https://www.tutorialspoint.com/java/java_filewriter_class.htm)

## Directories in Java

A directory is a File which can contain a list of other files and directories. You use **File** object to create directories, to list down files available in a directory. For complete detail, check a list of all the methods which you can call on File object and what are related to directories.

### Creating Directories

There are two useful **File** utility methods, which can be used to create directories −

* The **mkdir( )** method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the File object already exists, or that the directory cannot be created because the entire path does not exist yet.
* The **mkdirs()** method creates both a directory and all the parents of the directory.

Following example creates "/tmp/user/java/bin" directory −

**Example**

import java.io.File;

public class CreateDir {

public static void main(String args[]) {

String dirname = "/tmp/user/java/bin";

File d = new File(dirname);

// Create directory now.

d.mkdirs();

}

}

Compile and execute the above code to create "/tmp/user/java/bin".

**Note** − Java automatically takes care of path separators on UNIX and Windows as per conventions. If you use a forward slash (/) on a Windows version of Java, the path will still resolve correctly.

## Listing Directories

You can use **list( )** method provided by **File** object to list down all the files and directories available in a directory as follows −

**Example**

import java.io.File;

public class ReadDir {

public static void main(String[] args) {

File file = null;

String[] paths;

try {

// create new file object

file = new File("/tmp");

// array of files and directory

paths = file.list();

// for each name in the path array

for(String path:paths) {

// prints filename and directory name

System.out.println(path);

}

} catch (Exception e) {

// if any error occurs

e.printStackTrace();

}

}

}

This will produce the following result based on the directories and files available in your **/tmp** directory −

**Output**

test1.txt

test2.txt

ReadDir.java

ReadDir.class

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [**next →**](https://www.javatpoint.com/java-fileinputstream-class)[**← prev**](https://www.javatpoint.com/reentrant-monitor-in-java) Java FileOutputStream Class Java FileOutputStream is an output stream used for writing data to a [file](https://www.javatpoint.com/java-file-class).  If you have to write primitive values into a file, use FileOutputStream class. You can write byte-oriented as well as character-oriented data through FileOutputStream class. But, for character-oriented data, it is preferred to use [FileWriter](https://www.javatpoint.com/java-filterwriter-class) than FileOutputStream. FileOutputStream class declaration Let's see the declaration for Java.io.FileOutputStream class:   1. **public** **class** FileOutputStream **extends** OutputStream  FileOutputStream class methods  |  |  | | --- | --- | | **Method** | **Description** | | protected void finalize() | It is used to clean up the connection with the file output stream. | | void write(byte[] ary) | It is used to write **ary.length** bytes from the byte [array](https://www.javatpoint.com/array-in-java) to the file output stream. | | void write(byte[] ary, int off, int len) | It is used to write **len** bytes from the byte array starting at offset **off** to the file output stream. | | void write(int b) | It is used to write the specified byte to the file output stream. | | FileChannel getChannel() | It is used to return the file channel object associated with the file output stream. | | FileDescriptor getFD() | It is used to return the file descriptor associated with the stream. | | void close() | It is used to closes the file output stream. |  Java FileOutputStream Example 1: write byte  1. **import** java.io.FileOutputStream; 2. **public** **class** FileOutputStreamExample { 3. **public** **static** **void** main(String args[]){ 4. **try**{ 5. FileOutputStream fout=**new** FileOutputStream("D:\\testout.txt"); 6. fout.write(65); 7. fout.close(); 8. System.out.println("success..."); 9. }**catch**(Exception e){System.out.println(e);} 10. } 11. }   Output:  Success...  The content of a text file **testout.txt** is set with the data **A**.  testout.txt  A Java FileOutputStream example 2: write string  1. **import** java.io.FileOutputStream; 2. **public** **class** FileOutputStreamExample { 3. **public** **static** **void** main(String args[]){ 4. **try**{ 5. FileOutputStream fout=**new** FileOutputStream("D:\\testout.txt"); 6. String s="Welcome to javaTpoint."; 7. **byte** b[]=s.getBytes();//converting string into byte array 8. fout.write(b); 9. fout.close(); 10. System.out.println("success..."); 11. }**catch**(Exception e){System.out.println(e);} 12. } 13. }   Output:  Success... |

# Java FileInputStream Class

Java FileInputStream class obtains input bytes from a [file](https://www.javatpoint.com/java-file-class). It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc. You can also read character-stream data. But, for reading streams of characters, it is recommended to use [FileReader](https://www.javatpoint.com/java-filereader-class) class.

## Java FileInputStream class declaration

Let's see the declaration for java.io.FileInputStream class:

1. **public** **class** FileInputStream **extends** InputStream

## Java FileInputStream class methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| int available() | It is used to return the estimated number of bytes that can be read from the input stream. |
| int read() | It is used to read the byte of data from the input stream. |
| int read(byte[] b) | It is used to read up to **b.length** bytes of data from the input stream. |
| int read(byte[] b, int off, int len) | It is used to read up to **len** bytes of data from the input stream. |
| long skip(long x) | It is used to skip over and discards x bytes of data from the input stream. |
| FileChannel getChannel() | It is used to return the unique FileChannel object associated with the file input stream. |
| FileDescriptor getFD() | It is used to return the [FileDescriptor](https://www.javatpoint.com/java-filedescriptor-class) object. |
| protected void finalize() | It is used to ensure that the close method is call when there is no more reference to the file input stream. |
| void close() | It is used to closes the [stream](https://www.javatpoint.com/java-8-stream). |

## Java FileInputStream example 1: read single character

1. **import** java.io.FileInputStream;
2. **public** **class** DataStreamExample {
3. **public** **static** **void** main(String args[]){
4. **try**{
5. FileInputStream fin=**new** FileInputStream("D:\\testout.txt");
6. **int** i=fin.read();
7. System.out.print((**char**)i);
9. fin.close();
10. }**catch**(Exception e){System.out.println(e);}
11. }
12. }

**Note:** Before running the code, a text file named as **"testout.txt"**is required to be created. In this file, we are having following content:

Welcome to javatpoint.

After executing the above program, you will get a single character from the file which is 87 (in byte form). To see the text, you need to convert it into character.

Output:

W

## Java FileInputStream example 2: read all characters

1. **package** com.javatpoint;
3. **import** java.io.FileInputStream;
4. **public** **class** DataStreamExample {
5. **public** **static** **void** main(String args[]){
6. **try**{
7. FileInputStream fin=**new** FileInputStream("D:\\testout.txt");
8. **int** i=0;
9. **while**((i=fin.read())!=-1){
10. System.out.print((**char**)i);
11. }
12. fin.close();
13. }**catch**(Exception e){System.out.println(e);}
14. }
15. }

Output:

Welcome to javaTpoint

# Java BufferedOutputStream Class

Java BufferedOutputStream [class](https://www.javatpoint.com/object-and-class-in-java) is used for buffering an output stream. It internally uses buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.

For adding the buffer in an OutputStream, use the BufferedOutputStream class. Let's see the syntax for adding the buffer in an OutputStream:

1. OutputStream os= **new** BufferedOutputStream(**new** FileOutputStream("D:\\IO Package\\testout.txt"));

## Java BufferedOutputStream class declaration

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

1. **public** **class** BufferedOutputStream **extends** FilterOutputStream

## Java BufferedOutputStream class constructors

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| BufferedOutputStream(OutputStream os) | It creates the new buffered output stream which is used for writing the data to the specified output stream. |
| BufferedOutputStream(OutputStream os, int size) | It creates the new buffered output stream which is used for writing the data to the specified output stream with a specified buffer size. |

## Java BufferedOutputStream class methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| void write(int b) | It writes the specified byte to the buffered output stream. |
| void write(byte[] b, int off, int len) | It write the bytes from the specified byte-input stream into a specified byte [array](https://www.javatpoint.com/array-in-java), starting with the given offset |
| void flush() | It flushes the buffered output stream. |

## Example of BufferedOutputStream class:

In this example, we are writing the textual information in the BufferedOutputStream object which is connected to the [FileOutputStream](https://www.javatpoint.com/java-fileoutputstream-class) [object](https://www.javatpoint.com/object-and-class-in-java). The flush() flushes the data of one stream and send it into another. It is required if you have connected the one stream with another.

1. **package** com.javatpoint;
2. **import** java.io.\*;
3. **public** **class** BufferedOutputStreamExample{
4. **public** **static** **void** main(String args[])**throws** Exception{
5. FileOutputStream fout=**new** FileOutputStream("D:\\testout.txt");
6. BufferedOutputStream bout=**new** BufferedOutputStream(fout);
7. String s="Welcome to javaTpoint.";
8. **byte** b[]=s.getBytes();
9. bout.write(b);
10. bout.flush();
11. bout.close();
12. fout.close();
13. System.out.println("success");
14. }
15. }

Output:

Success

# Java BufferedInputStream Class

Java BufferedInputStream [class](https://www.javatpoint.com/object-and-class-in-java) is used to read information from [stream](https://www.javatpoint.com/java-8-stream). It internally uses buffer mechanism to make the performance fast.

The important points about BufferedInputStream are:

* When the bytes from the stream are skipped or read, the internal buffer automatically refilled from the contained input stream, many bytes at a time.
* When a BufferedInputStream is created, an internal buffer [array](https://www.javatpoint.com/array-in-java) is created.

## Java BufferedInputStream class declaration

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

1. **public** **class** BufferedInputStream **extends** FilterInputStream

## Java BufferedInputStream class constructors

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| BufferedInputStream(InputStream IS) | It creates the BufferedInputStream and saves it argument, the input stream IS, for later use. |
| BufferedInputStream(InputStream IS, int size) | It creates the BufferedInputStream with a specified buffer size and saves it argument, the input stream IS, for later use. |

## Java BufferedInputStream class methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| int available() | It returns an estimate number of bytes that can be read from the input stream without blocking by the next invocation method for the input stream. |
| int read() | It read the next byte of data from the input stream. |
| int read(byte[] b, int off, int ln) | It read the bytes from the specified byte-input stream into a specified byte array, starting with the given offset. |
| void close() | It closes the input stream and releases any of the system resources associated with the stream. |
| void reset() | It repositions the stream at a position the mark method was last called on this input stream. |
| void mark(int readlimit) | It sees the general contract of the mark method for the input stream. |
| long skip(long x) | It skips over and discards x bytes of data from the input stream. |
| boolean markSupported() | It tests for the input stream to support the mark and reset methods. |

### Example of Java BufferedInputStream

Let's see the simple example to read data of [file](https://www.javatpoint.com/java-file-class) using BufferedInputStream:

1. **package** com.javatpoint;
3. **import** java.io.\*;
4. **public** **class** BufferedInputStreamExample{
5. **public** **static** **void** main(String args[]){
6. **try**{
7. FileInputStream fin=**new** FileInputStream("D:\\testout.txt");
8. BufferedInputStream bin=**new** BufferedInputStream(fin);
9. **int** i;
10. **while**((i=bin.read())!=-1){
11. System.out.print((**char**)i);
12. }
13. bin.close();
14. fin.close();
15. }**catch**(Exception e){System.out.println(e);}
16. }
17. }

# Java SequenceInputStream Class

[Java](https://www.javatpoint.com/java-tutorial) SequenceInputStream [class](https://www.javatpoint.com/object-class) is used to read data from multiple [streams](https://www.javatpoint.com/java-8-stream). It reads data sequentially (one by one).

## Java SequenceInputStream Class declaration

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

1. **public** **class** SequenceInputStream **extends** InputStream

## Constructors of SequenceInputStream class

|  |  |
| --- | --- |
| [**Constructor**](https://www.javatpoint.com/java-constructor) | **Description** |
| SequenceInputStream(InputStream s1, InputStream s2) | creates a new input stream by reading the data of two input stream in order, first s1 and then s2. |
| SequenceInputStream(Enumeration e) | creates a new input stream by reading the data of an enumeration whose type is InputStream. |

## Methods of SequenceInputStream class

|  |  |
| --- | --- |
| **Method** | **Description** |
| int read() | It is used to read the next byte of data from the input stream. |
| int read(byte[] ary, int off, int len) | It is used to read len bytes of data from the input stream into the [array](https://www.javatpoint.com/array-in-java) of bytes. |
| int available() | It is used to return the maximum number of byte that can be read from an input stream. |
| void close() | It is used to close the input stream. |

## Java SequenceInputStream Example

In this example, we are printing the data of two files testin.txt and testout.txt.

1. **package** com.javatpoint;
3. **import** java.io.\*;
4. **class** InputStreamExample {
5. **public** **static** **void** main(String args[])**throws** Exception{
6. FileInputStream input1=**new** FileInputStream("D:\\testin.txt");
7. FileInputStream input2=**new** FileInputStream("D:\\testout.txt");
8. SequenceInputStream inst=**new** SequenceInputStream(input1, input2);
9. **int** j;
10. **while**((j=inst.read())!=-1){
11. System.out.print((**char**)j);
12. }
13. inst.close();
14. input1.close();
15. input2.close();
16. }
17. }

Here, we are assuming that you have two files: testin.txt and testout.txt which have following information:

testin.txt:

Welcome to Java IO Programming.

testout.txt:

It is the example of Java SequenceInputStream class.

After executing the program, you will get following output:

Output:

Welcome to Java IO Programming. It is the example of Java SequenceInputStream class.

## Example that reads the data from two files and writes into another file

In this example, we are writing the data of two files **testin1.txt** and **testin2.txt** into another file named **testout.txt.**

1. **package** com.javatpoint;
3. **import** java.io.\*;
4. **class** Input1{
5. **public** **static** **void** main(String args[])**throws** Exception{
6. FileInputStream fin1=**new** FileInputStream("D:\\testin1.txt");
7. FileInputStream fin2=**new** FileInputStream("D:\\testin2.txt");
8. FileOutputStream fout=**new** FileOutputStream("D:\\testout.txt");
9. SequenceInputStream sis=**new** SequenceInputStream(fin1,fin2);
10. **int** i;
11. **while**((i=sis.read())!=-1)
12. {
13. fout.write(i);
14. }
15. sis.close();
16. fout.close();
17. fin1.close();
18. fin2.close();
19. System.out.println("Success..");
20. }
21. }

Output:

Succeess...

testout.txt:

1. Welcome to Java IO Programming. It is the example of Java SequenceInputStream **class**.

## SequenceInputStream example that reads data using enumeration

If we need to read the data from more than two files, we need to use [Enumeration](https://www.javatpoint.com/enum-in-java). Enumeration object can be obtained by calling elements() method of the Vector class. Let's see the simple example where we are reading the data from 4 files: a.txt, b.txt, c.txt and d.txt.

1. **package** com.javatpoint;
2. **import** java.io.\*;
3. **import** java.util.\*;
4. **class** Input2{
5. **public** **static** **void** main(String args[])**throws** IOException{
6. //creating the FileInputStream objects for all the files
7. FileInputStream fin=**new** FileInputStream("D:\\a.txt");
8. FileInputStream fin2=**new** FileInputStream("D:\\b.txt");
9. FileInputStream fin3=**new** FileInputStream("D:\\c.txt");
10. FileInputStream fin4=**new** FileInputStream("D:\\d.txt");
11. //creating Vector object to all the stream
12. Vector v=**new** Vector();
13. v.add(fin);
14. v.add(fin2);
15. v.add(fin3);
16. v.add(fin4);
17. //creating enumeration object by calling the elements method
18. Enumeration e=v.elements();
19. //passing the enumeration object in the constructor
20. SequenceInputStream bin=**new** SequenceInputStream(e);
21. **int** i=0;
22. **while**((i=bin.read())!=-1){
23. System.out.print((**char**)i);
24. }
25. bin.close();
26. fin.close();
27. fin2.close();
28. }
29. }

The a.txt, b.txt, c.txt and d.txt have following information:

a.txt:

Welcome

b.txt:

to

c.txt:

java

d.txt:

programming

Output:

Welcometojavaprogramming

# Java SequenceInputStream Class

[Java](https://www.javatpoint.com/java-tutorial) SequenceInputStream [class](https://www.javatpoint.com/object-class) is used to read data from multiple [streams](https://www.javatpoint.com/java-8-stream). It reads data sequentially (one by one).

## Java SequenceInputStream Class declaration

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

1. **public** **class** SequenceInputStream **extends** InputStream

## Constructors of SequenceInputStream class

|  |  |
| --- | --- |
| [**Constructor**](https://www.javatpoint.com/java-constructor) | **Description** |
| SequenceInputStream(InputStream s1, InputStream s2) | creates a new input stream by reading the data of two input stream in order, first s1 and then s2. |
| SequenceInputStream(Enumeration e) | creates a new input stream by reading the data of an enumeration whose type is InputStream. |

## Methods of SequenceInputStream class

|  |  |
| --- | --- |
| **Method** | **Description** |
| int read() | It is used to read the next byte of data from the input stream. |
| int read(byte[] ary, int off, int len) | It is used to read len bytes of data from the input stream into the [array](https://www.javatpoint.com/array-in-java) of bytes. |
| int available() | It is used to return the maximum number of byte that can be read from an input stream. |
| void close() | It is used to close the input stream. |

## Java SequenceInputStream Example

In this example, we are printing the data of two files testin.txt and testout.txt.

1. **package** com.javatpoint;
3. **import** java.io.\*;
4. **class** InputStreamExample {
5. **public** **static** **void** main(String args[])**throws** Exception{
6. FileInputStream input1=**new** FileInputStream("D:\\testin.txt");
7. FileInputStream input2=**new** FileInputStream("D:\\testout.txt");
8. SequenceInputStream inst=**new** SequenceInputStream(input1, input2);
9. **int** j;
10. **while**((j=inst.read())!=-1){
11. System.out.print((**char**)j);
12. }
13. inst.close();
14. input1.close();
15. input2.close();
16. }
17. }

Here, we are assuming that you have two files: testin.txt and testout.txt which have following information:

testin.txt:

Welcome to Java IO Programming.

testout.txt:

It is the example of Java SequenceInputStream class.

After executing the program, you will get following output:

Output:

Welcome to Java IO Programming. It is the example of Java SequenceInputStream class.

## Example that reads the data from two files and writes into another file

In this example, we are writing the data of two files **testin1.txt** and **testin2.txt** into another file named **testout.txt.**

1. **package** com.javatpoint;
3. **import** java.io.\*;
4. **class** Input1{
5. **public** **static** **void** main(String args[])**throws** Exception{
6. FileInputStream fin1=**new** FileInputStream("D:\\testin1.txt");
7. FileInputStream fin2=**new** FileInputStream("D:\\testin2.txt");
8. FileOutputStream fout=**new** FileOutputStream("D:\\testout.txt");
9. SequenceInputStream sis=**new** SequenceInputStream(fin1,fin2);
10. **int** i;
11. **while**((i=sis.read())!=-1)
12. {
13. fout.write(i);
14. }
15. sis.close();
16. fout.close();
17. fin1.close();
18. fin2.close();
19. System.out.println("Success..");
20. }
21. }

Output:

Succeess...

testout.txt:

1. Welcome to Java IO Programming. It is the example of Java SequenceInputStream **class**.

## SequenceInputStream example that reads data using enumeration

If we need to read the data from more than two files, we need to use [Enumeration](https://www.javatpoint.com/enum-in-java). Enumeration object can be obtained by calling elements() method of the Vector class. Let's see the simple example where we are reading the data from 4 files: a.txt, b.txt, c.txt and d.txt.

1. **package** com.javatpoint;
2. **import** java.io.\*;
3. **import** java.util.\*;
4. **class** Input2{
5. **public** **static** **void** main(String args[])**throws** IOException{
6. //creating the FileInputStream objects for all the files
7. FileInputStream fin=**new** FileInputStream("D:\\a.txt");
8. FileInputStream fin2=**new** FileInputStream("D:\\b.txt");
9. FileInputStream fin3=**new** FileInputStream("D:\\c.txt");
10. FileInputStream fin4=**new** FileInputStream("D:\\d.txt");
11. //creating Vector object to all the stream
12. Vector v=**new** Vector();
13. v.add(fin);
14. v.add(fin2);
15. v.add(fin3);
16. v.add(fin4);
17. //creating enumeration object by calling the elements method
18. Enumeration e=v.elements();
19. //passing the enumeration object in the constructor
20. SequenceInputStream bin=**new** SequenceInputStream(e);
21. **int** i=0;
22. **while**((i=bin.read())!=-1){
23. System.out.print((**char**)i);
24. }
25. bin.close();
26. fin.close();
27. fin2.close();
28. }
29. }

The a.txt, b.txt, c.txt and d.txt have following information:

a.txt:

Welcome

b.txt:

to

c.txt:

java

d.txt:

programming

Output:

Welcometojavaprogramming

# Java ByteArrayOutputStream Class

Java ByteArrayOutputStream class is used to **write common data** into multiple files. In this stream, the data is written into a byte [array](https://www.javatpoint.com/array-in-java) which can be written to multiple streams later.

The ByteArrayOutputStream holds a copy of data and forwards it to multiple streams.

The buffer of ByteArrayOutputStream automatically grows according to data.

## Java ByteArrayOutputStream class declaration

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

1. **public** **class** ByteArrayOutputStream **extends** OutputStream

## Java ByteArrayOutputStream class constructors

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| ByteArrayOutputStream() | Creates a new byte array output [stream](https://www.javatpoint.com/java-8-stream) with the initial capacity of 32 bytes, though its size increases if necessary. |
| ByteArrayOutputStream(int size) | Creates a new byte array output stream, with a buffer capacity of the specified size, in bytes. |

## Java ByteArrayOutputStream class methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| int size() | It is used to returns the current size of a buffer. |
| byte[] toByteArray() | It is used to create a newly allocated byte array. |
| String toString() | It is used for converting the content into a [string](https://www.javatpoint.com/java-string) decoding bytes using a platform default character set. |
| String toString(String charsetName) | It is used for converting the content into a string decoding bytes using a specified charsetName. |
| void write(int b) | It is used for writing the byte specified to the byte array output stream. |
| void write(byte[] b, int off, int len | It is used for writing **len** bytes from specified byte array starting from the offset **off** to the byte array output stream. |
| void writeTo(OutputStream out) | It is used for writing the complete content of a byte array output stream to the specified output stream. |
| void reset() | It is used to reset the count field of a byte array output stream to zero value. |
| void close() | It is used to close the ByteArrayOutputStream. |

## Example of Java ByteArrayOutputStream

Let's see a simple example of [java](https://www.javatpoint.com/java-tutorial) ByteArrayOutputStream class to write common data into 2 files: f1.txt and f2.txt.

1. **package** com.javatpoint;
2. **import** java.io.\*;
3. **public** **class** DataStreamExample {
4. **public** **static** **void** main(String args[])**throws** Exception{
5. FileOutputStream fout1=**new** FileOutputStream("D:\\f1.txt");
6. FileOutputStream fout2=**new** FileOutputStream("D:\\f2.txt");
8. ByteArrayOutputStream bout=**new** ByteArrayOutputStream();
9. bout.write(65);
10. bout.writeTo(fout1);
11. bout.writeTo(fout2);
13. bout.flush();
14. bout.close();//has no effect
15. System.out.println("Success...");
16. }
17. }

Output:

Success...

f1.txt:

A

f2.txt:

A

# Java ByteArrayInputStream Class

The ByteArrayInputStream is composed of two words: ByteArray and InputStream. As the name suggests, it can be used to read byte [array](https://www.javatpoint.com/array-in-java) as input stream.

Java ByteArrayInputStream [class](https://www.javatpoint.com/object-and-class-in-java) contains an internal buffer which is used to **read byte array** as stream. In this stream, the data is read from a byte array.

The buffer of ByteArrayInputStream automatically grows according to data.

## Java ByteArrayInputStream class declaration

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

1. **public** **class** ByteArrayInputStream **extends** InputStream

## Java ByteArrayInputStream class constructors

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| ByteArrayInputStream(byte[] ary) | Creates a new byte array input stream which uses **ary** as its buffer array. |
| ByteArrayInputStream(byte[] ary, int offset, int len) | Creates a new byte array input stream which uses **ary** as its buffer array that can read up to specified **len** bytes of data from an array. |

## Java ByteArrayInputStream class methods

|  |  |
| --- | --- |
| **Methods** | **Description** |
| int available() | It is used to return the number of remaining bytes that can be read from the input stream. |
| int read() | It is used to read the next byte of data from the input stream. |
| int read(byte[] ary, int off, int len) | It is used to read up to len bytes of data from an array of bytes in the input stream. |
| boolean markSupported() | It is used to test the input stream for mark and reset method. |
| long skip(long x) | It is used to skip the x bytes of input from the input stream. |
| void mark(int readAheadLimit) | It is used to set the current marked position in the stream. |
| void reset() | It is used to reset the buffer of a byte array. |
| void close() | It is used for closing a ByteArrayInputStream. |

## Example of Java ByteArrayInputStream

Let's see a simple example of [java](https://www.javatpoint.com/java-tutorial) ByteArrayInputStream class to read byte array as input stream.

1. **package** com.javatpoint;
2. **import** java.io.\*;
3. **public** **class** ReadExample {
4. **public** **static** **void** main(String[] args) **throws** IOException {
5. **byte**[] buf = { 35, 36, 37, 38 };
6. // Create the new byte array input stream
7. ByteArrayInputStream byt = **new** ByteArrayInputStream(buf);
8. **int** k = 0;
9. **while** ((k = byt.read()) != -1) {
10. //Conversion of a byte into character
11. **char** ch = (**char**) k;
12. System.out.println("ASCII value of Character is:" + k + "; Special character is: " + ch);
13. }
14. }
15. }

Output:

ASCII value of Character is:35; Special character is: #

ASCII value of Character is:36; Special character is: $

ASCII value of Character is:37; Special character is: %

ASCII value of Character is:38; Special character is: &