**JAVA NOTES**

* The history of Java is very interesting. Java was originally designed for interactive television, but it was too advanced technology for the digital cable television industry at the time. The history of Java starts with the Green Team. Java team members (also known as Green Team), initiated this project to develop a language for digital devices such as set-top boxes, televisions, etc. However, it was best suited for internet programming. Later, Java technology was incorporated by Netscape.
* The principles for creating Java programming were "Simple, Robust, Portable, Platform-independent, Secured, High Performance, Multithreaded, Architecture Neutral, Object-Oriented, Interpreted, and Dynamic". [Java](https://www.javatpoint.com/java-tutorial) was developed by James Gosling, who is known as the father of Java, in 1995. James Gosling and his team members started the project in the early '90s.
* According to Sun, 3 billion devices run Java. There are many devices where Java is currently used. Some of them are as follows:

1. Desktop Applications such as acrobat reader, media player, antivirus, etc.
2. Web Applications such as irctc.co.in, javatpoint.com, etc.
3. Enterprise Applications such as banking applications.
4. Mobile
5. Embedded System
6. Smart Card
7. Robotics
8. Games, etc.
9. **Standalone Application**

Standalone applications are also known as desktop applications or window-based applications. These are traditional software that we need to install on every machine.

**Web Application**

An application that runs on the server side and creates a dynamic page is called a web application. Currently, [Servlet](https://www.javatpoint.com/servlet-tutorial), [JSP](https://www.javatpoint.com/jsp-tutorial), [Struts](https://www.javatpoint.com/struts-2-tutorial), [Spring](https://www.javatpoint.com/spring-tutorial), [Hibernate](https://www.javatpoint.com/hibernate-tutorial), [JSF](https://www.javatpoint.com/jsf-tutorial), etc. technologies are used for creating web applications in Java.

**Enterprise Application**

An application that is distributed in nature, such as banking applications, etc. is called an enterprise application. It has advantages like high-level security, load balancing, and clustering. In Java, [EJB](https://www.javatpoint.com/ejb-tutorial) is used for creating enterprise applications.

**Mobile Application**

An application which is created for mobile devices is called a mobile application. Currently, Android and Java ME are used for creating mobile applications.

**Object and Classes**

* An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.
* An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.
* **State:** represents the data (value) of an object.
* **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
* **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

**Class in Java**

* A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.
* A class in Java can contain:
* **Fields**
* **Methods**
* **Constructors**
* **Blocks**
* **Nested class and interface**

Discussion on Public Static Void Main

* **1. Public**
* It is an *Access modifier*, which specifies from where and who can access the method. Making the *main()* method public makes it globally available. It is made public so that JVM can invoke it from outside the class as it is not present in the current class.
* **2. Static**
* It is a *keyword* that is when associated with a method, making it a class-related method. The *main()* method is static so that JVM can invoke it without instantiating the class. This also saves the unnecessary wastage of memory which would have been used by the object declared only for calling the *main()* method by the JVM.
* **3. Void**
* It is a keyword and is used to specify that a method doesn’t return anything. As the *main()* method doesn’t return anything, its return type is *void*. As soon as the *main()* method terminates, the java program terminates too. Hence, it doesn’t make any sense to return from the *main()* method as JVM can’t do anything with the return value of it.
* **4. main**
* It is the name of the Java main method. It is the identifier that the JVM looks for as the starting point of the java program. It’s not a keyword.

**Can we execute a java program without main method?**

* Yes, we can execute a java program without a main method by using a static block.
* A **static block** in Java is a group of statements that gets executed only once when the class is loaded into the memory by ClassLoader, It is also known as a static initialization block, and it goes into the stack memory.

**JVM**

* Java program runs as a ‘main thread’ in [JVM](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/). The Java program is not even a process of Operating System directly. There is no direct interaction between the Java program and Operating System. There is no direct allocation of resources to the Java program directly, or the Java program does not occupy any place in the process table. Whom should it return an exit status to, then? This is why the main method of Java is designed not to return an int or exit status.
* But JVM is a process of an operating system, and JVM can be terminated with a certain exit status. With help of java.lang.Runtime.exit(int status) or System.exit(int status).

**Why Java is Platform Independent**

* Whenever, a program is written in JAVA, the javac compiles it.

The result of the JAVA compiler is the **.class file or the bytecode** and not the machine native code (unlike C compiler).

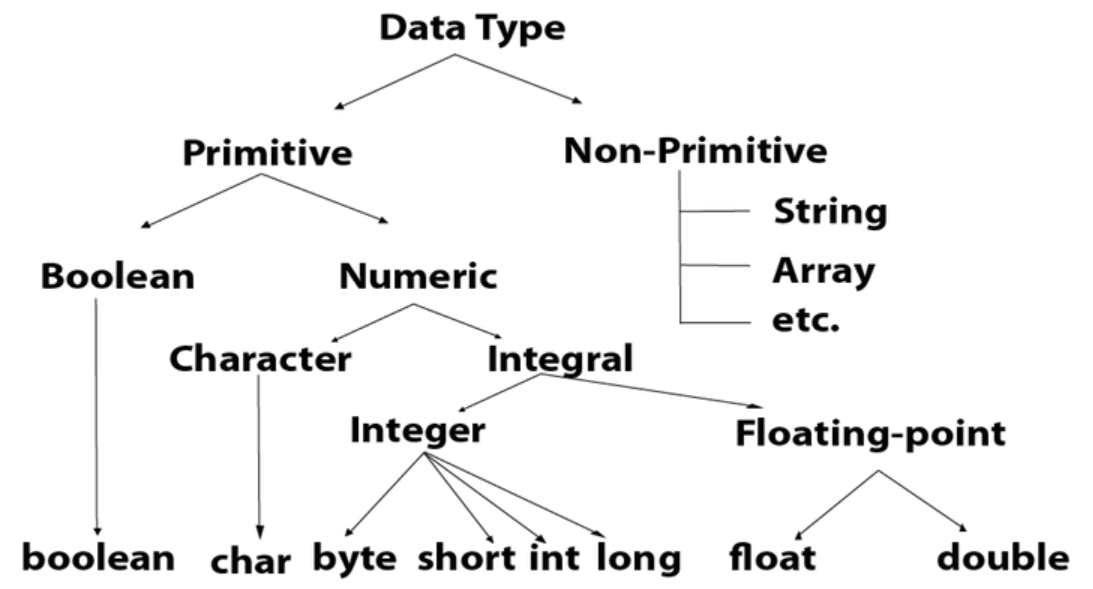
The bytecode generated is a non-executable code and needs an interpreter to execute on a machine. This interpreter is the JVM and thus the Bytecode is executed by the JVM.

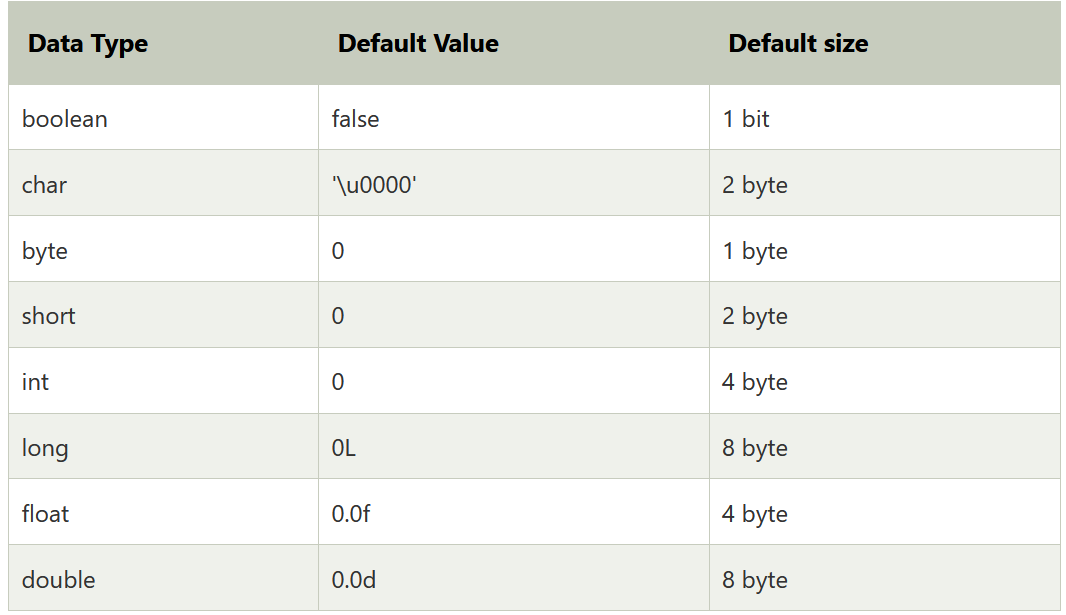
And finally program runs to give the desired output.

**Data Types in JAVA**

There are 8 types of primitive data types:

* boolean data type
* byte data type
* char data type
* short data type
* int data type
* long data type
* float data type
* double data type





**Operators in Java**

**Operator** in [Java](https://www.javatpoint.com/java-tutorial) is a symbol that is used to perform operations. For example: +, -, \*, / etc.

There are many types of operators in Java which are given below:

* Unary Operator,
* Arithmetic Operator,
* Shift Operator,
* Relational Operator,
* Bitwise Operator,
* Logical Operator,
* Ternary Operator and
* Assignment Operator.

## Java Operator Precedence

|  |  |  |
| --- | --- | --- |
| **Operator Type** | **Category** | **Precedence** |
| Unary | postfix | expr++ expr-- |
| prefix | ++expr --expr +expr -expr ~ ! |
| Arithmetic | multiplicative | \* / % |
| additive | + - |
| Shift | shift | << >> >>> |
| Relational | comparison | < > <= >= instanceof |
| equality | == != |
| Bitwise | bitwise AND | & |
| bitwise exclusive OR | ^ |
| bitwise inclusive OR | | |
| Logical | logical AND | && |
| logical OR | || |
| Ternary | ternary | ? : |
| Assignment | assignment | = += -= \*= /= %= &= ^= |= <<= >>= >>>= |

### **Java Unary Operator**

The Java unary operators require only one operand. Unary operators are used to perform various operations i.e.:

* incrementing/decrementing a value by one
* negating an expression
* inverting the value of a boolean

### **Java Unary Operator Example: ++ and --**

1. **public** **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** x=10;
4. System.out.println(x++);//10 (11)
5. System.out.println(++x);//12
6. System.out.println(x--);//12 (11)
7. System.out.println(--x);//10
8. }}

**Output:**

10

12

12

10

### **Java Unary Operator Example 2: ++ and --**

1. **public** **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** a=10;
4. **int** b=10;
5. System.out.println(a++ + ++a);//10+12=22
6. System.out.println(b++ + b++);//10+11=21
8. }}

**Output:**

22

21

### **Java Unary Operator Example: ~ and !**

1. **public** **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** a=10;
4. **int** b=-10;
5. **boolean** c=**true**;
6. **boolean** d=**false**;
7. System.out.println(~a);//-11 (minus of total positive value which starts from 0)
8. System.out.println(~b);//9 (positive of total minus, positive starts from 0)
9. System.out.println(!c);//false (opposite of boolean value)
10. System.out.println(!d);//true
11. }}

**Output:**

-11

9

false

true

### **Java Arithmetic Operators**

Java arithmetic operators are used to perform addition, subtraction, multiplication, and division. They act as basic mathematical operations.

### **Java Arithmetic Operator Example**

1. **public** **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** a=10;
4. **int** b=5;
5. System.out.println(a+b);//15
6. System.out.println(a-b);//5
7. System.out.println(a\*b);//50
8. System.out.println(a/b);//2
9. System.out.println(a%b);//0
10. }}

**Output:**

15

5

50

2

0

### **Java Arithmetic Operator Example: Expression**

1. **public** **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. System.out.println(10\*10/5+3-1\*4/2);
4. }}

**Output:**

21

### **Java Left Shift Operator**

The Java left shift operator << is used to shift all of the bits in a value to the left side of a specified number of times.

### **Java Left Shift Operator Example**

1. **public** **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. System.out.println(10<<2);//10\*2^2=10\*4=40
4. System.out.println(10<<3);//10\*2^3=10\*8=80
5. System.out.println(20<<2);//20\*2^2=20\*4=80
6. System.out.println(15<<4);//15\*2^4=15\*16=240
7. }}

**Output:**

40

80

80

240

### **Java Right Shift Operator**

The Java right shift operator >> is used to move the value of the left operand to right by the number of bits specified by the right operand.

### **Java Right Shift Operator Example**

1. **public** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. System.out.println(10>>2);//10/2^2=10/4=2
4. System.out.println(20>>2);//20/2^2=20/4=5
5. System.out.println(20>>3);//20/2^3=20/8=2
6. }}

**Output:**

2

5

2

### **Java Shift Operator Example: >> vs >>>**

1. **public** **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. //For positive number, >> and >>> works same
4. System.out.println(20>>2);
5. System.out.println(20>>>2);
6. //For negative number, >>> changes parity bit (MSB) to 0
7. System.out.println(-20>>2);
8. System.out.println(-20>>>2);
9. }}

**Output:**

5

5

-5

1073741819

### **Java AND Operator Example: Logical && and Bitwise &**

The logical && operator doesn't check the second condition if the first condition is false. It checks the second condition only if the first one is true.

The bitwise & operator always checks both conditions whether first condition is true or false.

1. **public** **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** a=10;
4. **int** b=5;
5. **int** c=20;
6. System.out.println(a<b&&a<c);//false && true = false
7. System.out.println(a<b&a<c);//false & true = false
8. }}

**Output:**

false

false

### **Java AND Operator Example: Logical && vs Bitwise &**

1. **public** **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** a=10;
4. **int** b=5;
5. **int** c=20;
6. System.out.println(a<b&&a++<c);//false && true = false
7. System.out.println(a);//10 because second condition is not checked
8. System.out.println(a<b&a++<c);//false && true = false
9. System.out.println(a);//11 because second condition is checked
10. }}

**Output:**

false

10

false

11

### **Java OR Operator Example: Logical || and Bitwise |**

The logical || operator doesn't check the second condition if the first condition is true. It checks the second condition only if the first one is false.

The bitwise | operator always checks both conditions whether first condition is true or false.

1. **public** **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** a=10;
4. **int** b=5;
5. **int** c=20;
6. System.out.println(a>b||a<c);//true || true = true
7. System.out.println(a>b|a<c);//true | true = true
8. //|| vs |
9. System.out.println(a>b||a++<c);//true || true = true
10. System.out.println(a);//10 because second condition is not checked
11. System.out.println(a>b|a++<c);//true | true = true
12. System.out.println(a);//11 because second condition is checked
13. }}

**Output:**

true

true

true

10

true

11

### **Java Ternary Operator**

Java Ternary operator is used as one line replacement for if-then-else statement and used a lot in Java programming. It is the only conditional operator which takes three operands.

### **Java Ternary Operator Example**

1. **public** **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** a=2;
4. **int** b=5;
5. **int** min=(a<b)?a:b;
6. System.out.println(min);
7. }}

**Output:**

2

Another Example:

1. **public** **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** a=10;
4. **int** b=5;
5. **int** min=(a<b)?a:b;
6. System.out.println(min);
7. }}

**Output:**

5

### **Java Assignment Operator**

Java assignment operator is one of the most common operators. It is used to assign the value on its right to the operand on its left.

### **Java Assignment Operator Example**

1. **public** **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** a=10;
4. **int** b=20;
5. a+=4;//a=a+4 (a=10+4)
6. b-=4;//b=b-4 (b=20-4)
7. System.out.println(a);
8. System.out.println(b);
9. }}

**Output:**

14

16

### **Java Assignment Operator Example**

1. **public** **class** OperatorExample{
2. **public** **static** **void** main(String[] args){
3. **int** a=10;
4. a+=3;//10+3
5. System.out.println(a);
6. a-=4;//13-4
7. System.out.println(a);
8. a\*=2;//9\*2
9. System.out.println(a);
10. a/=2;//18/2
11. System.out.println(a);
12. }}

**Output:**

13

9

18

9

### **Java Assignment Operator Example: Adding short**

1. **public** **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **short** a=10;
4. **short** b=10;
5. //a+=b;//a=a+b internally so fine
6. a=a+b;//Compile time error because 10+10=20 now int
7. System.out.println(a);
8. }}

**Output:**

Compile time error

After type cast:

1. **public** **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **short** a=10;
4. **short** b=10;
5. a=(**short**)(a+b);//20 which is int now converted to short
6. System.out.println(a);
7. }}

**Output:**

20



* Simple for Loop
* [For-each](https://www.javatpoint.com/for-each-loop) or Enhanced for Loop
* Labeled for Loop

## Java Simple for Loop

A simple for loop is the same as [C](https://www.javatpoint.com/c-programming-language-tutorial)/[C++](https://www.javatpoint.com/cpp-tutorial). We can initialize the [variable](https://www.javatpoint.com/java-variables), check condition and increment/decrement value. It consists of four parts:

1. **Initialization**: It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.
2. **Condition**: It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return boolean value either true or false. It is an optional condition.
3. **Increment/Decrement**: It increments or decrements the variable value. It is an optional condition.
4. **Statement**: The statement of the loop is executed each time until the second condition is false.

**Syntax:**

1. **for**(initialization; condition; increment/decrement){
2. //statement or code to be executed
3. }

**Flowchart:**



**Example:**

**ForExample.java**

1. //Java Program to demonstrate the example of for loop
2. //which prints table of 1
3. **public** **class** ForExample {
4. **public** **static** **void** main(String[] args) {
5. //Code of Java for loop
6. **for**(**int** i=1;i<=10;i++){
7. System.out.println(i);
8. }
9. }
10. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=ForExample)

**Output:**

1

2

3

4

5

6

7

8

9

10

## Java Nested for Loop

If we have a for loop inside the another loop, it is known as nested for loop. The inner loop executes completely whenever outer loop executes.

**Example:**

**NestedForExample.java**

1. **public** **class** NestedForExample {
2. **public** **static** **void** main(String[] args) {
3. //loop of i
4. **for**(**int** i=1;i<=3;i++){
5. //loop of j
6. **for**(**int** j=1;j<=3;j++){
7. System.out.println(i+" "+j);
8. }//end of i
9. }//end of j
10. }
11. }

**Output:**

1 1

1 2

1 3

2 1

2 2

2 3

3 1

3 2

3 3

**Pyramid Example 1:**

**PyramidExample.java**

1. **public** **class** PyramidExample {
2. **public** **static** **void** main(String[] args) {
3. **for**(**int** i=1;i<=5;i++){
4. **for**(**int** j=1;j<=i;j++){
5. System.out.print("\* ");
6. }
7. System.out.println();//new line
8. }
9. }
10. }

**Output:**

\*

\* \*

\* \* \*

\* \* \* \*

\* \* \* \* \*

**Pyramid Example 2:**

**PyramidExample2.java**

1. **public** **class** PyramidExample2 {
2. **public** **static** **void** main(String[] args) {
3. **int** term=6;
4. **for**(**int** i=1;i<=term;i++){
5. **for**(**int** j=term;j>=i;j--){
6. System.out.print("\* ");
7. }
8. System.out.println();//new line
9. }
10. }
11. }

**Output:**

\* \* \* \* \* \*

\* \* \* \* \*

\* \* \* \*

\* \* \*

\* \*

\*

## Java for-each Loop

The for-each loop is used to traverse array or collection in Java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation.

It works on the basis of elements and not the index. It returns element one by one in the defined variable.

**Syntax:**

1. **for**(data\_type variable : array\_name){
2. //code to be executed
3. }

**Example:**

**ForEachExample.java**

1. //Java For-each loop example which prints the
2. //elements of the array
3. **public** **class** ForEachExample {
4. **public** **static** **void** main(String[] args) {
5. //Declaring an array
6. **int** arr[]={12,23,44,56,78};
7. //Printing array using for-each loop
8. **for**(**int** i:arr){
9. System.out.println(i);
10. }
11. }
12. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=ForEachExample)

**Output:**

12

23

44

56

78

## Java Labeled For Loop

We can have a name of each Java for loop. To do so, we use label before the for loop. It is useful while using the nested for loop as we can break/continue specific for loop.

#### Note: The break and continue keywords breaks or continues the innermost for loop respectively.

**Syntax:**

1. labelname:
2. **for**(initialization; condition; increment/decrement){
3. //code to be executed
4. }

**Example:**

**LabeledForExample.java**

1. //A Java program to demonstrate the use of labeled for loop
2. **public** **class** LabeledForExample {
3. **public** **static** **void** main(String[] args) {
4. //Using Label for outer and for loop
5. aa:
6. **for**(**int** i=1;i<=3;i++){
7. bb:
8. **for**(**int** j=1;j<=3;j++){
9. **if**(i==2&&j==2){
10. **break** aa;
11. }
12. System.out.println(i+" "+j);
13. }
14. }
15. }
16. }

**Output:**

1 1

1 2

1 3

2 1

If you use **break bb;**, it will break inner loop only which is the default behaviour of any loop.

**LabeledForExample2.java**

1. **public** **class** LabeledForExample2 {
2. **public** **static** **void** main(String[] args) {
3. aa:
4. **for**(**int** i=1;i<=3;i++){
5. bb:
6. **for**(**int** j=1;j<=3;j++){
7. **if**(i==2&&j==2){
8. **break** bb;
9. }
10. System.out.println(i+" "+j);
11. }
12. }
13. }
14. }

**Java While Loop**

The [Java](https://www.javatpoint.com/java-tutorial) while loop is used to iterate a part of the [program](https://www.javatpoint.com/programs-list) repeatedly until the specified Boolean condition is true. As soon as the Boolean condition becomes false, the loop automatically stops.

The while loop is considered as a repeating if statement. If the number of iteration is not fixed, it is recommended to use the while [loop](https://www.javatpoint.com/java-for-loop).

**Syntax:**

1. **while** (condition){
2. //code to be executed
3. I ncrement / decrement statement
4. }

**The different parts of do-while loop:**

1. Condition: It is an expression which is tested. If the condition is true, the loop body is executed and control goes to update expression. When the condition becomes false, we exit the while loop.

**Example**:

i <=100

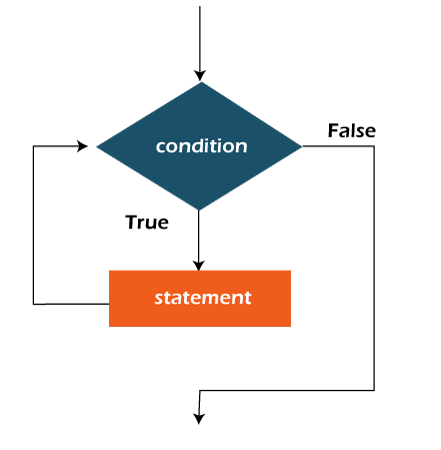
2. Update expression: Every time the loop body is executed, this expression increments or decrements loop variable.

**Example:**

**i++;**

**Flowchart of Java While Loop**

Here, the important thing about while loop is that, sometimes it may not even execute. If the condition to be tested results into false, the loop body is skipped and first statement after the while loop will be executed.



**Example:**

In the below example, we print integer values from 1 to 10. Unlike the for loop, we separately need to initialize and increment the variable used in the condition (here, i). Otherwise, the loop will execute infinitely.

**WhileExample.java**

1. **public** **class** WhileExample {
2. **public** **static** **void** main(String[] args) {
3. **int** i=1;
4. **while**(i<=10){
5. System.out.println(i);
6. i++;
7. }
8. }
9. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=WhileExample)

**Output:**

1

2

3

4

5

6

7

8

9

10

## Java Infinitive While Loop

If you pass **true** in the while loop, it will be infinitive while loop.

**Syntax:**

1. **while**(**true**){
2. //code to be executed
3. }

**Example:**

**WhileExample2.java**

1. **public** **class** WhileExample2 {
2. **public** **static** **void** main(String[] args) {
3. // setting the infinite while loop by passing true to the condition
4. **while**(**true**){
5. System.out.println("infinitive while loop");
6. }
7. }
8. }

**Output:**

infinitive while loop

infinitive while loop

infinitive while loop

infinitive while loop

infinitive while loop

ctrl+c

**DO WHILE LOOP**

The Java do-while loop is used to iterate a part of the program repeatedly, until the specified condition is true. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use a do-while loop.

Java do-while loop is called an **exit control loop**. Therefore, unlike while loop and for loop, the do-while check the condition at the end of loop body. The Java do-while loop is executed at least once because condition is checked after loop body.

**Syntax:**

1. **do**{
2. //code to be executed / loop body
3. //update statement
4. }**while** (condition);

**The different parts of do-while loop:**

1. Condition: It is an expression which is tested. If the condition is true, the loop body is executed and control goes to update expression. As soon as the condition becomes false, loop breaks automatically.

**Example:**

**i <=100**

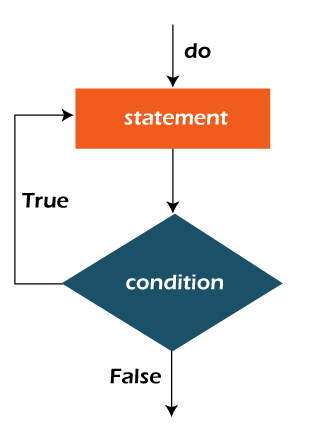
2. Update expression: Every time the loop body is executed, the this expression increments or decrements loop variable.

**Example:**

**i++;**

#### Note: The do block is executed at least once, even if the condition is false.

**Flowchart of do-while loop:**



**Example:**

In the below example, we print integer values from 1 to 10. Unlike the for loop, we separately need to initialize and increment the variable used in the condition (here, i). Otherwise, the loop will execute infinitely.

**DoWhileExample.java**

1. **public** **class** DoWhileExample {
2. **public** **static** **void** main(String[] args) {
3. **int** i=1;
4. **do**{
5. System.out.println(i);
6. i++;
7. }**while**(i<=10);
8. }
9. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=DoWhileExample)

**Output:**

1

2

3

4

5

6

7

8

9

10

## Java Infinitive do-while Loop

If you pass **true** in the do-while loop, it will be infinitive do-while loop.

**Syntax:**

1. **do**{
2. //code to be executed
3. }**while**(**true**);

**Example:**

**DoWhileExample2.java**

1. **public** **class** DoWhileExample2 {
2. **public** **static** **void** main(String[] args) {
3. **do**{
4. System.out.println("infinitive do while loop");
5. }**while**(**true**);
6. }
7. }

**Constructors in Java**

In [Java](https://www.javatpoint.com/java-tutorial), a constructor is a block of codes similar to the method. It is called when an instance of the [class](https://www.javatpoint.com/object-and-class-in-java) is created. At the time of calling constructor, memory for the object is allocated in the memory.

It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

There are two types of constructors in Java: no-arg constructor, and parameterized constructor.

**Note:** It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

### **Rules for creating Java constructor**

There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

#### Note: We can use [access modifiers](https://www.javatpoint.com/access-modifiers) while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.

## Types of Java constructors

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor



## Java Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.

### **Syntax of default constructor:**

1. <class\_name>(){}

## Example of default constructor

|  |
| --- |
| In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation. |

1. //Java Program to create and call a default constructor
2. **class** Bike1{
3. //creating a default constructor
4. Bike1(){System.out.println("Bike is created");}
5. //main method
6. **public** **static** **void** main(String args[]){
7. //calling a default constructor
8. Bike1 b=**new** Bike1();
9. }
10. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Bike1)

Output:

Bike is created

#### Rule: If there is no constructor in a class, compiler automatically creates a default constructor.



### **Q) What is the purpose of a default constructor?**

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

### **Example of default constructor that displays the default values**

1. //Let us see another example of default constructor
2. //which displays the default values
3. **class** Student3{
4. **int** id;
5. String name;
6. //method to display the value of id and name
7. **void** display(){System.out.println(id+" "+name);}
9. **public** **static** **void** main(String args[]){
10. //creating objects
11. Student3 s1=**new** Student3();
12. Student3 s2=**new** Student3();
13. //displaying values of the object
14. s1.display();
15. s2.display();
16. }
17. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Student3)

Output:

0 null

0 null

**Explanation:**In the above class,you are not creating any constructor so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.

### **Java Parameterized Constructor**

A constructor which has a specific number of parameters is called a parameterized constructor.

### **Why use the parameterized constructor?**

The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

### **Example of parameterized constructor**

In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

1. //Java Program to demonstrate the use of the parameterized constructor.
2. **class** Student4{
3. **int** id;
4. String name;
5. //creating a parameterized constructor
6. Student4(**int** i,String n){
7. id = i;
8. name = n;
9. }
10. //method to display the values
11. **void** display(){System.out.println(id+" "+name);}
13. **public** **static** **void** main(String args[]){
14. //creating objects and passing values
15. Student4 s1 = **new** Student4(111,"Karan");
16. Student4 s2 = **new** Student4(222,"Aryan");
17. //calling method to display the values of object
18. s1.display();
19. s2.display();
20. }
21. }

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

Output:

111 Karan

222 Aryan

## Constructor Overloading in Java

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

Constructor [overloading in Java](https://www.javatpoint.com/method-overloading-in-java) is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

### **Example of Constructor Overloading**

1. //Java program to overload constructors
2. **class** Student5{
3. **int** id;
4. String name;
5. **int** age;
6. //creating two arg constructor
7. Student5(**int** i,String n){
8. id = i;
9. name = n;
10. }
11. //creating three arg constructor
12. Student5(**int** i,String n,**int** a){
13. id = i;
14. name = n;
15. age=a;
16. }
17. **void** display(){System.out.println(id+" "+name+" "+age);}
19. **public** **static** **void** main(String args[]){
20. Student5 s1 = **new** Student5(111,"Karan");
21. Student5 s2 = **new** Student5(222,"Aryan",25);
22. s1.display();
23. s2.display();
24. }
25. }

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

Output:

111 Karan 0

222 Aryan 25

## Difference between constructor and method in Java

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

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



**Static keyword in Java**

The **static keyword** in [Java](https://www.javatpoint.com/java-tutorial) is used for memory management mainly. We can apply static keyword with [variables](https://www.javatpoint.com/java-variables), methods, blocks and [nested classes](https://www.javatpoint.com/java-inner-class). The static keyword belongs to the class than an instance of the class.

The static can be:

1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block
4. Nested class



## 1) Java static variable

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

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

### **Advantages of static variable**

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

#### Understanding the problem without static variable

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

Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all [objects](https://www.javatpoint.com/object-and-class-in-java). If we make it static, this field will get the memory only once.

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

### **Example of static variable**

1. //Java Program to demonstrate the use of static variable
2. **class** Student{
3. **int** rollno;//instance variable
4. String name;
5. **static** String college ="ITS";//static variable
6. //constructor
7. Student(**int** r, String n){
8. rollno = r;
9. name = n;
10. }
11. //method to display the values
12. **void** display (){System.out.println(rollno+" "+name+" "+college);}
13. }
14. //Test class to show the values of objects
15. **public** **class** TestStaticVariable1{
16. **public** **static** **void** main(String args[]){
17. Student s1 = **new** Student(111,"Karan");
18. Student s2 = **new** Student(222,"Aryan");
19. //we can change the college of all objects by the single line of code
20. //Student.college="BBDIT";
21. s1.display();
22. s2.display();
23. }
24. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestStaticVariable1)

Output:

111 Karan ITS

222 Aryan ITS



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

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

1. //Java Program to demonstrate the use of an instance variable
2. //which get memory each time when we create an object of the class.
3. **class** Counter{
4. **int** count=0;//will get memory each time when the instance is created
6. Counter(){
7. count++;//incrementing value
8. System.out.println(count);
9. }
11. **public** **static** **void** main(String args[]){
12. //Creating objects
13. Counter c1=**new** Counter();
14. Counter c2=**new** Counter();
15. Counter c3=**new** Counter();
16. }
17. }

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

Output:

1

1

1

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

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

1. //Java Program to illustrate the use of static variable which
2. //is shared with all objects.
3. **class** Counter2{
4. **static** **int** count=0;//will get memory only once and retain its value
6. Counter2(){
7. count++;//incrementing the value of static variable
8. System.out.println(count);
9. }
11. **public** **static** **void** main(String args[]){
12. //creating objects
13. Counter2 c1=**new** Counter2();
14. Counter2 c2=**new** Counter2();
15. Counter2 c3=**new** Counter2();
16. }
17. }

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

Output:

1

2

3

## 2) Java static method

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

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

### **Example of static method**

1. //Java Program to demonstrate the use of a static method.
2. **class** Student{
3. **int** rollno;
4. String name;
5. **static** String college = "ITS";
6. //static method to change the value of static variable
7. **static** **void** change(){
8. college = "BBDIT";
9. }
10. //constructor to initialize the variable
11. Student(**int** r, String n){
12. rollno = r;
13. name = n;
14. }
15. //method to display values
16. **void** display(){System.out.println(rollno+" "+name+" "+college);}
17. }
18. //Test class to create and display the values of object
19. **public** **class** TestStaticMethod{
20. **public** **static** **void** main(String args[]){
21. Student.change();//calling change method
22. //creating objects
23. Student s1 = **new** Student(111,"Karan");
24. Student s2 = **new** Student(222,"Aryan");
25. Student s3 = **new** Student(333,"Sonoo");
26. //calling display method
27. s1.display();
28. s2.display();
29. s3.display();
30. }
31. }

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

Output:111 Karan BBDIT

222 Aryan BBDIT

333 Sonoo BBDIT

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

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

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

Output:125

### **Restrictions for the static method**

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

1. The static method can not use non static data member or call non-static method directly.
2. this and super cannot be used in static context.
3. **class** A{
4. **int** a=40;//non static
6. **public** **static** **void** main(String args[]){
7. System.out.println(a);
8. }
9. }

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

Output:Compile Time Error

### **Q) Why is the Java main method static?**

Ans) It is because the object is not required to call a static method. If it were a non-static method, [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) creates an object first then call main() method that will lead the problem of extra memory allocation.

## 3) Java static block

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

### **Example of static block**

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

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

Output:static block is invoked

Hello main

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

Ans) No, one of the ways was the static block, but it was possible till JDK 1.6. Since JDK 1.7, it is not possible to execute a Java class without the [main method](https://www.javatpoint.com/java-main-method).

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

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

Output:

static block is invoked

Since JDK 1.7 and above, output would be:

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

public static void main(String[] args)

or a JavaFX application class must extend javafx.application.Application

**Inheritance in Java**

**Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of [OOPs](https://www.javatpoint.com/java-oops-concepts) (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new [classes](https://www.javatpoint.com/object-and-class-in-java) that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a parent-child relationship.

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

* For [Method Overriding](https://www.javatpoint.com/method-overriding-in-java) (so [runtime polymorphism](https://www.javatpoint.com/runtime-polymorphism-in-java) can be achieved).
* For Code Reusability.

### **Terms used in Inheritance**

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

### **The syntax of Java Inheritance**

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

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

### **Java Inheritance Example**



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

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

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

Programmer salary is:40000.0

Bonus of programmer is:10000

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

## Types of inheritance in java

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

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



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

When one class inherits multiple classes, it is known as multiple inheritance. For Example:



## Single Inheritance Example

When a class inherits another class, it is known as a single inheritance. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

*File: TestInheritance.java*

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

Output:

barking...

eating...

## Multilevel Inheritance Example

When there is a chain of inheritance, it is known as multilevel inheritance. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

*File: TestInheritance2.java*

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

Output:

weeping...

barking...

eating...

## Hierarchical Inheritance Example

When two or more classes inherits a single class, it is known as hierarchical inheritance. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

*File: TestInheritance3.java*

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

Output:

meowing...

eating...

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

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

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

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

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

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

Compile Time Error

**Polymorphism**

If one task is performed in different ways, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.

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

Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.

# **Method Overloading in Java**

If a [class](https://www.javatpoint.com/object-and-class-in-java) has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the [program](https://www.javatpoint.com/java-programs).

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

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

## Advantage of method overloading

Method overloading increases the readability of the program.

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

There are two ways to overload the method in java

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

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

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

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

In this example, we are creating [static methods](https://www.javatpoint.com/static-keyword-in-java) so that we don't need to create instance for calling methods.

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

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

Output:

22

33

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

In this example, we have created two methods that differs in [data type](https://www.javatpoint.com/java-data-types). The first add method receives two integer arguments and second add method receives two double arguments.

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

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

Output:

22

24.9

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

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

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

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

Output:

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

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

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

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

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) calls main() method which receives string array as arguments only. Let's see the simple example:

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

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

Output:

main with String[]

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

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

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

Output:40

60

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

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

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

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

Output:int arg method invoked

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

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

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

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

Output:Compile Time Error

# **Method Overriding in Java**

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

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

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

* Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
* Method overriding is used for runtime polymorphism

#### Rules for Java Method Overriding

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



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

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

1. //Java Program to demonstrate why we need method overriding
2. //Here, we are calling the method of parent class with child
3. //class object.
4. //Creating a parent class
5. **class** Vehicle{
6. **void** run(){System.out.println("Vehicle is running");}
7. }
8. //Creating a child class
9. **class** Bike **extends** Vehicle{
10. **public** **static** **void** main(String args[]){
11. //creating an instance of child class
12. Bike obj = **new** Bike();
13. //calling the method with child class instance
14. obj.run();
15. }
16. }

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

Output:

Vehicle is running

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

### **Example of method overriding**

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

1. //Java Program to illustrate the use of Java Method Overriding
2. //Creating a parent class.
3. **class** Vehicle{
4. //defining a method
5. **void** run(){System.out.println("Vehicle is running");}
6. }
7. //Creating a child class
8. **class** Bike2 **extends** Vehicle{
9. //defining the same method as in the parent class
10. **void** run(){System.out.println("Bike is running safely");}
12. **public** **static** **void** main(String args[]){
13. Bike2 obj = **new** Bike2();//creating object
14. obj.run();//calling method
15. }
16. }

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

Output:

Bike is running safely

### **A real example of Java Method Overriding**

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



#### Java method overriding is mostly used in Runtime Polymorphism which we will learn in next pages.

1. //Java Program to demonstrate the real scenario of Java Method Overriding
2. //where three classes are overriding the method of a parent class.
3. //Creating a parent class.
4. **class** Bank{
5. **int** getRateOfInterest(){**return** 0;}
6. }
7. //Creating child classes.
8. **class** SBI **extends** Bank{
9. **int** getRateOfInterest(){**return** 8;}
10. }
12. **class** ICICI **extends** Bank{
13. **int** getRateOfInterest(){**return** 7;}
14. }
15. **class** AXIS **extends** Bank{
16. **int** getRateOfInterest(){**return** 9;}
17. }
18. //Test class to create objects and call the methods
19. **class** Test2{
20. **public** **static** **void** main(String args[]){
21. SBI s=**new** SBI();
22. ICICI i=**new** ICICI();
23. AXIS a=**new** AXIS();
24. System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
25. System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
26. System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
27. }
28. }

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

Output:

SBI Rate of Interest: 8

ICICI Rate of Interest: 7

AXIS Rate of Interest: 9

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

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

### **Why can we not override static method?**

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

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

No, because the main is a static method.

**Super Keyword in Java**

The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

## Usage of Java super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.



## 1) super is used to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

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

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

Output:

black

white

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

## 2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

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

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

Output:

eating...

barking...

In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.

To call the parent class method, we need to use super keyword.

## 3) super is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

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

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

Output:

animal is created

dog is created

#### Note: super() is added in each class constructor automatically by compiler if there is no super() or this().



As we know well that default constructor is provided by compiler automatically if there is no constructor. But, it also adds super() as the first statement.

**Another example of super keyword where super() is provided by the compiler implicitly.**

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

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

Output:

animal is created

dog is created

## super example: real use

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

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

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

Output:

1 ankit 45000

**This Keyword in Java**

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



Here is given the 3 usage of java this keyword.

1. [this can be used to refer current class instance variable.](https://www.javatpoint.com/this1)
2. [this can be used to invoke current class method (implicitly)](https://www.javatpoint.com/this2)
3. [this() can be used to invoke current class constructor.](https://www.javatpoint.com/this3)

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

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

#### Understanding the problem without this keyword

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

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

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

**Output:**

0 null 0.0

0 null 0.0

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

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

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

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

**Output:**

111 ankit 5000.0

112 sumit 6000.0

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

#### Program where this keyword is not required

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

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

**Output:**

111 ankit 5000.0

112 sumit 6000.0

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

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

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



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

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

**Output:**

hello n

hello m

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

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

**Calling default constructor from parameterized constructor:**

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

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

**Output:**

hello a

10

**Final Keyword in Java**

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

## 1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

### **Example of final variable**

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

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

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

Output:Compile Time Error

## 2) Java final method

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

### **Example of final method**

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

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

Output:Compile Time Error

## 3) Java final class

If you make any class as final, you cannot extend it.

### **Example of final class**

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

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

Output:Compile Time Error

### **Q) Is final method inherited?**

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

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

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

Output:running...

**Abstract Class in Java**

A class which is declared with the abstract keyword is known as an abstract class in [Java](https://www.javatpoint.com/java-tutorial). It can have abstract and non-abstract methods (method with the body)

### **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](https://www.javatpoint.com/object-and-class-in-java) 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%)

#### 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](https://www.javatpoint.com/java-constructor) 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**](https://www.javatpoint.com/opr/test.jsp?filename=Honda4)

running safely

### **Understanding the real scenario of Abstract class**

In this example, Shape is the abstract class, 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**](https://www.javatpoint.com/opr/test.jsp?filename=TestAbstraction1)

drawing circle

### **Another example of Abstract class in java**

*File: TestBank.java*

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

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

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**](https://www.javatpoint.com/opr/test.jsp?filename=TestAbstraction2)

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**](https://www.javatpoint.com/opr/test.jsp?filename=Bike12)

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](https://www.javatpoint.com/interface-in-java). 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**

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*](https://www.javatpoint.com/abstract-class-in-java). There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple [inheritance in Java](https://www.javatpoint.com/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](https://www.javatpoint.com/java-8-features), 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**](https://www.javatpoint.com/opr/test.jsp?filename=A6)

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**](https://www.javatpoint.com/opr/test.jsp?filename=TestInterface1)

Output:

drawing circle

## Java Interface Example: Bank

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

*File: TestInterface2.java*

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

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

Output:

ROI: 9.15

## Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, 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**](https://www.javatpoint.com/opr/test.jsp?filename=A7)

Output:Hello

Welcome

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

As we have explained in the inheritance chapter, multiple inheritance is not supported in the case of [class](https://www.javatpoint.com/object-and-class-in-java) 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**](https://www.javatpoint.com/opr/test.jsp?filename=TestInterface3)

Output:

Hello

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

## Interface inheritance

A class implements 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**](https://www.javatpoint.com/opr/test.jsp?filename=TestInterface4)

Output:

Hello

Welcome

## Java 8 Default Method in Interface

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

*File: TestInterfaceDefault.java*

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

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

Output:

drawing rectangle

default method

## Java 8 Static Method in Interface

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

*File: TestInterfaceStatic.java*

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

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

Output:

drawing rectangle

27

## Q) What is marker or tagged interface?

An interface which has no member is known as a marker or tagged interface, for example, [Serializable](https://www.javatpoint.com/serialization-in-java), 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. }

**Encapsulation in Java**

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



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

### **Advantage of Encapsulation in Java**

By providing only a setter or getter method, you can make the class **read-only or write-only**. In other words, you can skip the getter or setter methods.

It provides you the **control over the data**. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.

It is a way to achieve **data hiding** in Java because other class will not be able to access the data through the private data members.

The encapsulate class is **easy to test**. So, it is better for unit testing.

The standard IDE's are providing the facility to generate the getters and setters. So, it is **easy and fast to create an encapsulated class** in Java.

### **Simple Example of Encapsulation in Java**

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

*File: Student.java*

1. //A Java class which is a fully encapsulated class.
2. //It has a private data member and getter and setter methods.
3. **public** **class** Student{
4. //private data member
5. **private** String name;
6. //getter method for name
7. **public** String getName(){
8. **return** name;
9. }
10. //setter method for name
11. **public** **void** setName(String name){
12. **this**.name=name
13. }
14. }
15. **class** Test{
16. **public** **static** **void** main(String[] args){
17. //creating instance of the encapsulated class
18. Student s=**new** Student();
19. //setting value in the name member
20. s.setName("vijay");
21. //getting value of the name member
22. System.out.println(s.getName());
23. }
24. }

**Access Modifiers in Java**

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

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

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

### **Understanding Java Access Modifiers**

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

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

### **1) Private**

The private access modifier is accessible only within the class.

**Simple example of private access modifier**

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

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

### **Role of Private Constructor**

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

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

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

### **2) Default**

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

**Example of default access modifier**

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

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

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

### **3) Protected**

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

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

It provides more accessibility than the default modifer.

**Example of protected access modifier**

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

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

Output:Hello

### **4) Public**

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

**Example of public access modifier**

1. //save by A.java
3. **package** pack;
4. **public** **class** A{
5. **public** **void** msg(){System.out.println("Hello");}
6. }
7. //save by B.java
9. **package** mypack;
10. **import** pack.\*;
12. **class** B{
13. **public** **static** **void** main(String args[]){
14. A obj = **new** A();
15. obj.msg();
16. }
17. }

Output:Hello

**Arrays in Java**

**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**](https://www.javatpoint.com/opr/test.jsp?filename=Testarray)

Output:

10

20

70

40

50

## Declaration, Instantiation and Initialization of Java Array

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

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

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

1. //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**](https://www.javatpoint.com/opr/test.jsp?filename=Testarray1)

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**](https://www.javatpoint.com/opr/test.jsp?filename=Testarray2)

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**](https://www.javatpoint.com/opr/test.jsp?filename=TestAnonymousArray)

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**](https://www.javatpoint.com/opr/test.jsp?filename=TestReturnArray)

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**](https://www.javatpoint.com/opr/test.jsp?filename=TestArrayException)

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**](https://www.javatpoint.com/opr/test.jsp?filename=Testarray3)

Output:

1 2 3

2 4 5

4 4 5

## What is the class name of Java array?

In Java, 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**](https://www.javatpoint.com/opr/test.jsp?filename=Testarray4)

Output:

I

## Copying a Java Array

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

**Syntax of arraycopy method**

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

### **Example of 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**](https://www.javatpoint.com/opr/test.jsp?filename=TestArrayCopyDemo)

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**](https://www.javatpoint.com/opr/test.jsp?filename=Testarray5)

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)

Output:

6 6 6

12 12 12

18 18 18

**String in Java**

In [Java](https://www.javatpoint.com/java-tutorial), string is basically an object that represents sequence of char values. An [array](https://www.javatpoint.com/array-in-java) of characters works same as Java string. For example:

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

is same as:

1. String s="javatpoint";

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

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

The java.lang.String class implements Serializable, Comparable and CharSequence [interfaces](https://www.javatpoint.com/interface-in-java).



## CharSequence Interface

The CharSequence interface is used to represent the sequence of characters. String, [StringBuffer](https://www.javatpoint.com/StringBuffer-class) and [StringBuilder](https://www.javatpoint.com/StringBuilder-class) classes implement it. It means, we can create strings in Java by using these three classes.



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

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

### **What is String in Java?**

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

### **How to create a string object?**

There are two ways to create String object:

1. By string literal
2. By new keyword

### **1) String Literal**

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

1. String s="welcome";

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

1. String s1="Welcome";
2. String s2="Welcome";//It doesn't create a new instance



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

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

### **Why Java uses the concept of String literal?**

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

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

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

In such case, [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

### **Java String Example**

**StringExample.java**

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

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

**Output:**

java

strings

example

The above code, converts a **char** array into a **String** object. And displays the String objects **s1, s2**, and **s3** on console using **println()** method.

****

****

**String Buffer**

Java StringBuffer class is used to create mutable (modifiable) String objects. The StringBuffer class in Java is the 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() | It creates an empty String buffer with the initial capacity of  16. |
| StringBuffer(String str) | It creates a String buffer with the specified string.. |
| StringBuffer(int capacity) | It creates an empty String buffer with the specified  capacity as length. |

### **1) StringBuffer Class append() Method**

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

**StringBufferExample.java**

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. }

**Output:**

Hello Java

### **2) StringBuffer insert() Method**

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

**StringBufferExample2.java**

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. }

**Output:**

HJavaello

### **3) StringBuffer replace() Method**

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

**StringBufferExample3.java**

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. }

**Output:**

HJavalo

### **4) StringBuffer delete() Method**

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

**StringBufferExample4.java**

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. }

**Output:**

Hlo

### **5) StringBuffer reverse() Method**

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

**StringBufferExample5.java**

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. }

**Output:**

olleH

### **6) StringBuffer capacity() Method**

The capacity() method of the 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.

**StringBufferExample6.java**

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. }

**Output:**

16

16

34

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

The ensureCapacity() method of the 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.

**StringBufferExample7.java**

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. }

**Output:**

16

16

34

34

70

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

## Important methods of StringBuilder class

|  |  |
| --- | --- |
| **Method** | **Description** |
| public StringBuilder append(String s) | It 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) | It 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) | It is used to replace the string from specified startIndex and endIndex. |
| public StringBuilder delete(int startIndex, int endIndex) | It is used to delete the string from specified startIndex and endIndex. |
| public StringBuilder reverse() | It is used to reverse the string. |
| public int capacity() | It is used to return the current capacity. |
| public void ensureCapacity(int minimumCapacity) | It is used to ensure the capacity at least equal to the given minimum. |
| public char charAt(int index) | It is used to return the character at the specified position. |
| public int length() | It is used to return the length of the string i.e. total number of characters. |
| public String substring(int beginIndex) | It is used to return the substring from the specified beginIndex. |
| public String substring(int beginIndex, int endIndex) | It 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.

**StringBuilderExample.java**

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. }

**Output:**

Hello Java

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

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

**StringBuilderExample2.java**

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. }

**Output:**

HJavaello

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

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

**StringBuilderExample3.java**

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. }

**Output:**

HJavalo

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

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

**StringBuilderExample4.java**

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. }

**Output:**

Hlo

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

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

**StringBuilderExample5.java**

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. }

**Output:**

olleH

### **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.

**StringBuilderExample6.java**

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. }

**Output:**

16

16

34

### **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.

**StringBuilderExample7.java**

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. }

**Output:**

16

16

34

34

70

**Exception Handling in Java**

The **Exception Handling in Java** is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.

In this tutorial, we will learn about Java exceptions, it's types, 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 need to handle exceptions. Let's consider 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 a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in [Java](https://www.javatpoint.com/java-tutorial).

## Hierarchy of Java Exception classes

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



### **Types of Java Exceptions**

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error



## Difference between Checked and Unchecked Exceptions

### **1) Checked Exception**

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

### **2) Unchecked Exception**

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

### **3) Error**

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

## Java Exception Keywords

Java provides five keywords that are used to handle the exception. The following table describes each.

|  |  |
| --- | --- |
| **Keyword** | **Description** |
| try | The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally. |
| 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 necessary 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 specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature. |

## Java Exception Handling Example

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

**JavaExceptionExample.java**

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**](https://www.javatpoint.com/opr/test.jsp?filename=JavaExceptionExample)

**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](https://www.javatpoint.com/java-variables), 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**

If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a [string](https://www.javatpoint.com/java-string) variable that has characters; converting this variable into digit will cause NumberFormatException.

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

### **4) A scenario where ArrayIndexOutOfBoundsException occurs**

When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

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

## 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 in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

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

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

### **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.

## 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 the application programmer handles the exception, the normal flow of the application is maintained, i.e., rest of the code is executed.

## Problem without exception handling

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

### **Example 1**

**TryCatchExample1.java**

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**](https://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample1)

**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 might be 100 lines of code after the exception. If the exception is not handled, all the code below the exception won't be executed.

## Solution by exception handling

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

### **Example 2**

**TryCatchExample2.java**

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**](https://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample2)

**Output:**

java.lang.ArithmeticException: / by zero

rest of the code

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.

**TryCatchExample3.java**

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**](https://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample3)

**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.

**TryCatchExample4.java**

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**](https://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample4)

**Output:**

java.lang.ArithmeticException: / by zero

rest of the code

### **Example 5**

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

**TryCatchExample5.java**

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**](https://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample5)

**Output:**

Can't divided by zero

### **Example 6**

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

**TryCatchExample6.java**

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**](https://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample6)

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

We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception. We will discuss custom exceptions later in this section.

We can also define our own set of conditions and throw an exception explicitly using throw keyword. For example, we can throw ArithmeticException if we divide a number by another number. Here, we just need to set the condition and throw exception using throw keyword.

The syntax of the Java throw keyword is given below.

throw Instance i.e.,

1. **throw** **new** exception\_class("error message");

Let's see the example of throw IOException.

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

Where the Instance must be of type Throwable or subclass of Throwable. For example, Exception is the sub class of Throwable and the user-defined exceptions usually extend the Exception class.

## Java throw keyword Example

### **Example 1: Throwing Unchecked Exception**

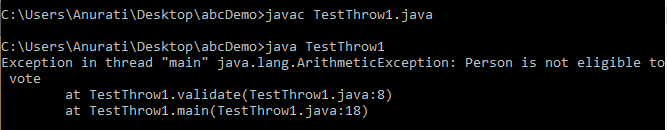
In this example, we have created a method named validate() that accepts an integer as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

**TestThrow1.java**

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. //function to check if person is eligible to vote or not
3. **public** **static** **void** validate(**int** age) {
4. **if**(age<18) {
5. //throw Arithmetic exception if not eligible to vote
6. **throw** **new** ArithmeticException("Person is not eligible to vote");
7. }
8. **else** {
9. System.out.println("Person is eligible to vote!!");
10. }
11. }
12. //main method
13. **public** **static** **void** main(String args[]){
14. //calling the function
15. validate(13);
16. System.out.println("rest of the code...");
17. }
18. }

**Output:**



The above code throw an unchecked exception. Similarly, we can also throw unchecked and user defined exceptions.

#### Note: If we throw unchecked exception from a method, it is must to handle the exception or declare in throws clause.

If we throw a checked exception using throw keyword, it is must to handle the exception using catch block or the method must declare it using throws declaration.

### **Example 2: Throwing Checked Exception**

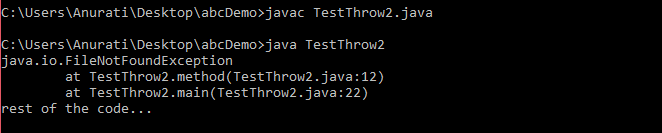
#### Note: Every subclass of Error and RuntimeException is an unchecked exception in Java. A checked exception is everything else under the Throwable class.

**TestThrow2.java**

1. **import** java.io.\*;
3. **public** **class** TestThrow2 {
5. //function to check if person is eligible to vote or not
6. **public** **static** **void** method() **throws** FileNotFoundException {
8. FileReader file = **new** FileReader("C:\\Users\\Anurati\\Desktop\\abc.txt");
9. BufferedReader fileInput = **new** BufferedReader(file);

12. **throw** **new** FileNotFoundException();
14. }
15. //main method
16. **public** **static** **void** main(String args[]){
17. **try**
18. {
19. method();
20. }
21. **catch** (FileNotFoundException e)
22. {
23. e.printStackTrace();
24. }
25. System.out.println("rest of the code...");
26. }
27. }

**Output:**



### **Example 3: Throwing User-defined Exception**

exception is everything else under the Throwable class.

**TestThrow3.java**

1. // class represents user-defined exception
2. **class** UserDefinedException **extends** Exception
3. {
4. **public** UserDefinedException(String str)
5. {
6. // Calling constructor of parent Exception
7. **super**(str);
8. }
9. }
10. // Class that uses above MyException
11. **public** **class** TestThrow3
12. {
13. **public** **static** **void** main(String args[])
14. {
15. **try**
16. {
17. // throw an object of user defined exception
18. **throw** **new** UserDefinedException("This is user-defined exception");
19. }
20. **catch** (UserDefinedException ude)
21. {
22. System.out.println("Caught the exception");
23. // Print the message from MyException object
24. System.out.println(ude.getMessage());
25. }
26. }
27. }

**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 the normal flow of the program 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 checking the code before it 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 our control so we can correct our code.
* **error:** beyond our control. For example, we 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.

**Testthrows1.java**

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**](https://www.javatpoint.com/opr/test.jsp?filename=Testthrows1)

**Output:**

exception handled

normal flow...

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

**There are two cases:**

1. **Case 1:** We have caught the exception i.e. we have handled the exception using try/catch block.
2. **Case 2:** We have declared the exception i.e. specified throws keyword with the method.

### **Case 1: Handle Exception Using try-catch block**

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

**Testthrows2.java**

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**](https://www.javatpoint.com/opr/test.jsp?filename=Testthrows2)

**Output:**

exception handled

normal flow...

### **Case 2: Declare Exception**

* In case we declare the exception, if exception does not occur, the code will be executed fine.
* In case we declare the exception and the exception occurs, it will be thrown at runtime because **throws** does not handle the exception.

Let's see examples for both the scenario.

**A) If exception does not occur**

**Testthrows3.java**

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**](https://www.javatpoint.com/opr/test.jsp?filename=Testthrows3)

**Output:**

device operation performed

normal flow...

**B) If exception occurs**

**Testthrows4.java**

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. }

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

Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

The finally block follows the try-catch block.

### **Flowchart of finally block**



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

## Why use Java finally block?

* finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
* The important statements to be printed can be placed in the finally block.

## Usage of Java finally

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

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

### **Case 1: When an exception does not occur**

Let's see the below example where the Java program does not throw any exception, and the finally block is executed after the try block.

**TestFinallyBlock.java**

1. **class** TestFinallyBlock {
2. **public** **static** **void** main(String args[]){
3. **try**{
4. //below code do not throw any exception
5. **int** data=25/5;
6. System.out.println(data);
7. }
8. //catch won't be executed
9. **catch**(NullPointerException e){
10. System.out.println(e);
11. }
12. //executed regardless of exception occurred or not
13. **finally** {
14. System.out.println("finally block is always executed");
15. }
17. System.out.println("rest of phe code...");
18. }
19. }

**Output:**



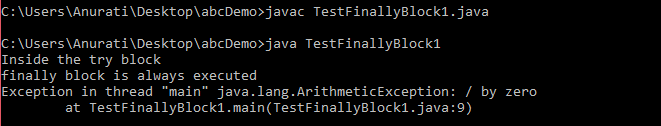
### **Case 2: When an exception occurr but not handled by the catch block**

Let's see the the fillowing example. Here, the code throws an exception however the catch block cannot handle it. Despite this, the finally block is executed after the try block and then the program terminates abnormally.

**TestFinallyBlock1.java**

1. **public** **class** TestFinallyBlock1{
2. **public** **static** **void** main(String args[]){
4. **try** {
6. System.out.println("Inside the try block");
8. //below code throws divide by zero exception
9. **int** data=25/0;
10. System.out.println(data);
11. }
12. //cannot handle Arithmetic type exception
13. //can only accept Null Pointer type exception
14. **catch**(NullPointerException e){
15. System.out.println(e);
16. }
18. //executes regardless of exception occured or not
19. **finally** {
20. System.out.println("finally block is always executed");
21. }
23. System.out.println("rest of the code...");
24. }
25. }

**Output:**



### **Case 3: When an exception occurs and is handled by the catch block**

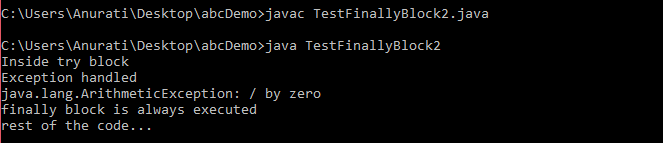
**Example:**

Let's see the following example where the Java code throws an exception and the catch block handles the exception. Later the finally block is executed after the try-catch block. Further, the rest of the code is also executed normally.

**TestFinallyBlock2.java**

1. **public** **class** TestFinallyBlock2{
2. **public** **static** **void** main(String args[]){
4. **try** {
6. System.out.println("Inside try block");
8. //below code throws divide by zero exception
9. **int** data=25/0;
10. System.out.println(data);
11. }
13. //handles the Arithmetic Exception / Divide by zero exception
14. **catch**(ArithmeticException e){
15. System.out.println("Exception handled");
16. System.out.println(e);
17. }
19. //executes regardless of exception occured or not
20. **finally** {
21. System.out.println("finally block is always executed");
22. }
24. System.out.println("rest of the code...");
25. }
26. }

**Output:**



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

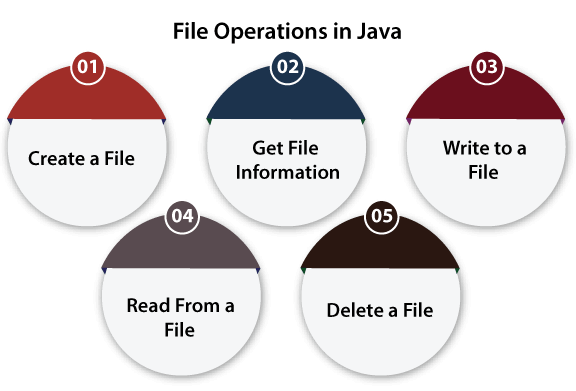
#### Note: The finally block will not be executed if the program exits (either by calling System.exit() or by causing a fatal error that causes the process to abort).

# **File Operations in Java**

In Java, a **File** is an abstract data type. A named location used to store related information is known as a **File**. There are several **File Operations** like **creating a new File, getting information about File, writing into a File, reading from a File** and **deleting a File**.

We can perform the following operation on a file:

* Create a File
* Get File Information
* Write to a File
* Read from a File
* Delete a File



### **Create a File**

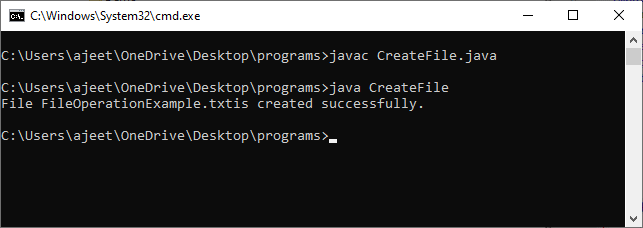
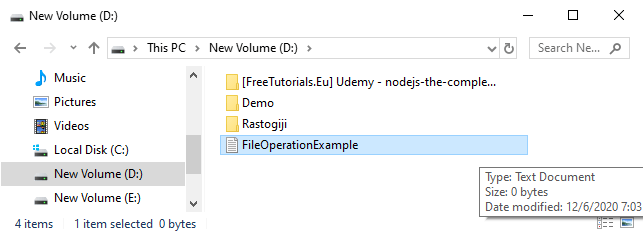
**Create a File** operation is performed to create a new file. We use the **createNewFile()** method of file. The **createNewFile()** method returns true when it successfully creates a new file and returns false when the file already exists.

Let's take an example of creating a file to understand how we can use the **createNewFile()** method to perform this operation.

**CreateFile.java**

1. // Importing File class
2. **import** java.io.File;
3. // Importing the IOException class for handling errors
4. **import** java.io.IOException;
5. **class** CreateFile {
6. **public** **static** **void** main(String args[]) {
7. **try** {
8. // Creating an object of a file
9. File f0 = **new** File("D:FileOperationExample.txt");
10. **if** (f0.createNewFile()) {
11. System.out.println("File " + f0.getName() + " is created successfully.");
12. } **else** {
13. System.out.println("File is already exist in the directory.");
14. }
15. } **catch** (IOException exception) {
16. System.out.println("An unexpected error is occurred.");
17. exception.printStackTrace();
18. }
19. }
20. }

**Output:**

**Explanation:**

In the above code, we import the File and IOException class for performing file operation and handling errors, respectively. We create the **f0** object of the File class and specify the location of the directory where we want to create a file. In the try block, we call the **createNewFile()** method through the **f0** object to create a new file in the specified location. If the method returns false, it will jump to the else section. If there is any error, it gets handled in the catch block.

### **Get File Information**

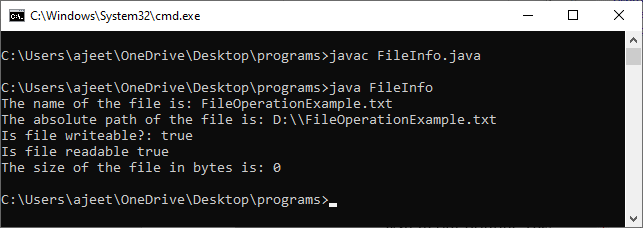
The operation is performed to get the file information. We use several methods to get the information about the file like name, absolute path, is readable, is writable and length.

Let's take an example to understand how to use file methods to get the information of the file.

**FileInfo.java**

1. // Import the File class
2. **import** java.io.File;
3. **class** FileInfo {
4. **public** **static** **void** main(String[] args) {
5. // Creating file object
6. File f0 = **new** File("D:FileOperationExample.txt");
7. **if** (f0.exists()) {
8. // Getting file name
9. System.out.println("The name of the file is: " + f0.getName());
11. // Getting path of the file
12. System.out.println("The absolute path of the file is: " + f0.getAbsolutePath());
14. // Checking whether the file is writable or not
15. System.out.println("Is file writeable?: " + f0.canWrite());
17. // Checking whether the file is readable or not
18. System.out.println("Is file readable " + f0.canRead());
20. // Getting the length of the file in bytes
21. System.out.println("The size of the file in bytes is: " + f0.length());
22. } **else** {
23. System.out.println("The file does not exist.");
24. }
25. }
26. }

**Output:**



**Description:**

In the above code, we import the **java.io.File** package and create a class **FileInfo**. In the main method, we create an object of the text file which we have created in our previous example. We check the existence of the file using a conditional statement, and if it is present, we get the following information about that file:

1. We get the name of the file using the **getName()**
2. We get the absolute path of the file using the **getAbsolutePath()** method of the file.
3. We check whether we can write data into a file or not using the **canWrite()**
4. We check whether we can read the data of the file or not using the **canRead()**
5. We get the length of the file by using the **length()**

If the file doesn't exist, we show a custom message.

### **Write to a File**

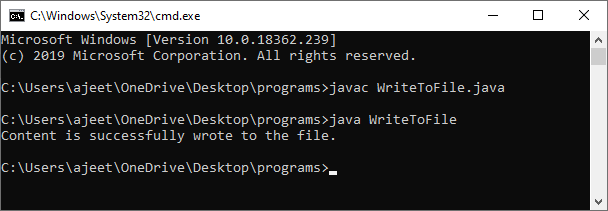
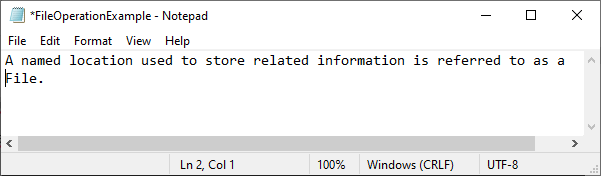
The next operation which we can perform on a file is **"writing into a file"**. In order to write data into a file, we will use the **FileWriter** class and its **write()** method together. We need to close the stream using the **close()** method to retrieve the allocated resources.

Let's take an example to understand how we can write data into a file.

**WriteToFile.java**

1. // Importing the FileWriter class
2. **import** java.io.FileWriter;
4. // Importing the IOException class for handling errors
5. **import** java.io.IOException;
7. **class** WriteToFile {
8. **public** **static** **void** main(String[] args) {
10. **try** {
11. FileWriter fwrite = **new** FileWriter("D:FileOperationExample.txt");
12. // writing the content into the FileOperationExample.txt file
13. fwrite.write("A named location used to store related information is referred to as a File.");
15. // Closing the stream
16. fwrite.close();
17. System.out.println("Content is successfully wrote to the file.");
18. } **catch** (IOException e) {
19. System.out.println("Unexpected error occurred");
20. e.printStackTrace();
21. }
22. }
23. }

**Output:**

**Explanation:**

In the above code, we import the **java.io.FileWriter** and **java.io.IOException** classes. We create a class WriteToFile, and in its main method, we use the **try-catch** block. In the try section, we create an instance of the FileWriter class, i.e., **fwrite**. We call the **write** method of the FileWriter class and pass the content to that function which we want to write. After that, we call the **close()** method of the FileWriter class to close the file stream. After writing the content and closing the stream, we print a custom message.

If we get any error in the try section, it jumps to the catch block. In the catch block, we handle the **IOException** and print a custom message.

### **Read from a File**

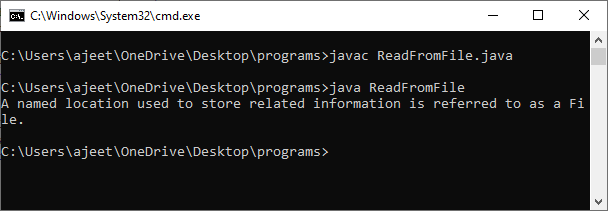
The next operation which we can perform on a file is **"read from a file"**. In order to write data into a file, we will use the **Scanner** class. Here, we need to close the stream using the **close()** method. We will create an instance of the [Scanner class](https://www.javatpoint.com/Scanner-class) and use the **[hasNextLine()](https://www.javatpoint.com/post/java-scanner-hasnextline-method)**[method](https://www.javatpoint.com/post/java-scanner-hasnextline-method) **[nextLine()](https://www.javatpoint.com/post/java-scanner-nextline-method)**[method](https://www.javatpoint.com/post/java-scanner-nextline-method) to get data from the file.

Let's take an example to understand how we can read data from a file.

**ReadFromFile.java**

1. // Importing the File class
2. **import** java.io.File;
3. // Importing FileNotFoundException class for handling errors
4. **import** java.io.FileNotFoundException;
5. // Importing the Scanner class for reading text files
6. **import** java.util.Scanner;
8. **class** ReadFromFile {
9. **public** **static** **void** main(String[] args) {
10. **try** {
11. // Create f1 object of the file to read data
12. File f1 = **new** File("D:FileOperationExample.txt");
13. Scanner dataReader = **new** Scanner(f1);
14. **while** (dataReader.hasNextLine()) {
15. String fileData = dataReader.nextLine();
16. System.out.println(fileData);
17. }
18. dataReader.close();
19. } **catch** (FileNotFoundException exception) {
20. System.out.println("Unexcpected error occurred!");
21. exception.printStackTrace();
22. }
23. }
24. }

**Output:**



**Expalnation:**

In the above code, we import the **"java.util.Scannner", "java.io.File"** and **"java.io.IOException"** classes. We create a class **ReadFromFile**, and in its main method, we use the **try-catch block**. In the try section, we create an instance of both the **Scanner** and the **File** classes. We pass the **File** class object to the **Scanner** class object and then iterate the scanner class object using the **"While"** loop and print each line of the file. We also need to close the scanner class object, so we use the close() function. If we get any error in the try section, it jumps to the catch block. In the catch block, we handle the IOException and print a custom message.

### **Delete a File**

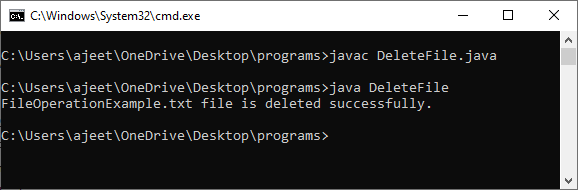
The next operation which we can perform on a file is **"deleting a file"**. In order to delete a file, we will use the **delete()** method of the file. We don't need to close the stream using the **close()** method because for deleting a file, we neither use the FileWriter class nor the Scanner class.

Let's take an example to understand how we can write data into a file.

**DeleteFile.java**

1. // Importing the File class
2. **import** java.io.File;
3. **class** DeleteFile {
4. **public** **static** **void** main(String[] args) {
5. File f0 = **new** File("D:FileOperationExample.txt");
6. **if** (f0.delete()) {
7. System.out.println(f0.getName()+ " file is deleted successfully.");
8. } **else** {
9. System.out.println("Unexpected error found in deletion of the file.");
10. }
11. }
12. }

**Output:**



**Explanation:**

In the above code, we import the **File** class and create a class **DeleteFile**. In the main() method of the class, we create **f0** object of the file which we want to delete. In the **if** statement, we call the **delete()** method of the file using the f0 object. If the delete() method returns true, we print the success custom message. Otherwise, it jumps to the else section where we print the unsuccessful custom message.

All the above-mentioned operations are used to read, write, delete, and create file programmatically.

# **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**](https://www.javatpoint.com/java-tutorial) 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.

## Java Thread Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **S.N.** | **Modifier and Type** | **Method** | **Description** |
| 1) | void | [start()](https://www.javatpoint.com/java-thread-start-method) | It is used to start the execution of the thread. |
| 2) | void | [run()](https://www.javatpoint.com/java-thread-run-method) | It is used to do an action for a thread. |
| 3) | static void | [sleep()](https://www.javatpoint.com/java-thread-sleep-method) | It sleeps a thread for the specified amount of time. |
| 4) | static Thread | [currentThread()](https://www.javatpoint.com/java-thread-currentthread-method) | It returns a reference to the currently executing thread object. |

# **Java Threads | How to create a thread in Java**

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:**

The **start() method** of Thread class is used to start a newly created thread. It performs the following tasks:

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

* 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**

**FileName:** Multi.java

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**

**FileName:** Multi3.java

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);   // Using the constructor Thread(Runnable r)
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 explicitly create the Thread class object. We are passing the object of your class that implements Runnable so that your class run() method may execute.

### **3) Using the Thread Class: Thread(String Name)**

We can directly use the Thread class to spawn new threads using the constructors defined above.

**FileName:** MyThread1.java

1. **public** **class** MyThread1
2. {
3. // Main method
4. **public** **static** **void** main(String argvs[])
5. {
6. // creating an object of the Thread class using the constructor Thread(String name)
7. Thread t= **new** Thread("My first thread");
9. // the start() method moves the thread to the active state
10. t.start();
11. // getting the thread name by invoking the getName() method
12. String str = t.getName();
13. System.out.println(str);
14. }
15. }

**Output:**

My first thread

### **4) Using the Thread Class: Thread(Runnable r, String name)**

Observe the following program.

**FileName:** MyThread2.java

1. **public** **class** MyThread2 **implements** Runnable
2. {
3. **public** **void** run()
4. {
5. System.out.println("Now the thread is running ...");
6. }
8. // main method
9. **public** **static** **void** main(String argvs[])
10. {
11. // creating an object of the class MyThread2
12. Runnable r1 = **new** MyThread2();
14. // creating an object of the class Thread using Thread(Runnable r, String name)
15. Thread th1 = **new** Thread(r1, "My new thread");
17. // the start() method moves the thread to the active state
18. th1.start();
20. // getting the thread name by invoking the getName() method
21. String str = th1.getName();
22. System.out.println(str);
23. }
24. }

**Output:**

My new thread

Now the thread is running ...

# **Collections in Java**

1. [Java Collection Framework](https://www.javatpoint.com/collections-in-java)
2. [Hierarchy of Collection Framework](https://www.javatpoint.com/collections-in-java#collectionhierarchy)
3. [Collection interface](https://www.javatpoint.com/collections-in-java#collectionmethods)
4. [Iterator interface](https://www.javatpoint.com/collections-in-java#collectioniterator)

The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes ([ArrayList](https://www.javatpoint.com/java-arraylist), Vector, [LinkedList](https://www.javatpoint.com/java-linkedlist), [PriorityQueue](https://www.javatpoint.com/java-priorityqueue), HashSet, LinkedHashSet, TreeSet).

#### What is Collection in Java

A Collection represents a single unit of objects, i.e., a group.

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

#### What is a framework in Java

* It provides readymade architecture.
* It represents a set of classes and interfaces.
* It is optional.

#### What is Collection framework

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

1. Interfaces and its implementations, i.e., classes
2. Algorithm

### **Hierarchy of Collection Framework**

Let us see the hierarchy of Collection framework. The **java.util** package contains all the [classes](https://www.javatpoint.com/object-and-class-in-java) and [interfaces](https://www.javatpoint.com/interface-in-java) for the Collection framework.



### **Methods of Collection interface**

There are many methods declared in the Collection interface. They are as follows:

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | public boolean add(E e) | It is used to insert an element in this collection. |
| 2 | public boolean addAll(Collection<? extends E> c) | It is used to insert the specified collection elements in the invoking collection. |
| 3 | public boolean remove(Object element) | It is used to delete an element from the collection. |
| 4 | public boolean removeAll(Collection<?> c) | It is used to delete all the elements of the specified collection from the invoking collection. |
| 5 | default boolean removeIf(Predicate<? super E> filter) | It is used to delete all the elements of the collection that satisfy the specified predicate. |
| 6 | public boolean retainAll(Collection<?> c) | It is used to delete all the elements of invoking collection except the specified collection. |
| 7 | public int size() | It returns the total number of elements in the collection. |
| 8 | public void clear() | It removes the total number of elements from the collection. |
| 9 | public boolean contains(Object element) | It is used to search an element. |
| 10 | public boolean containsAll(Collection<?> c) | It is used to search the specified collection in the collection. |
| 11 | public Iterator iterator() | It returns an iterator. |
| 12 | public Object[] toArray() | It converts collection into array. |
| 13 | public <T> T[] toArray(T[] a) | It converts collection into array. Here, the runtime type of the returned array is that of the specified array. |
| 14 | public boolean isEmpty() | It checks if collection is empty. |
| 15 | default Stream<E> parallelStream() | It returns a possibly parallel Stream with the collection as its source. |
| 16 | default Stream<E> stream() | It returns a sequential Stream with the collection as its source. |
| 17 | default Spliterator<E> spliterator() | It generates a Spliterator over the specified elements in the collection. |
| 18 | public boolean equals(Object element) | It matches two collections. |
| 19 | public int hashCode() | It returns the hash code number of the collection. |

### **Iterator interface**

|  |
| --- |
| Iterator interface provides the facility of iterating the elements in a forward direction only. |

#### Methods of Iterator interface

There are only three methods in the Iterator interface. They are:

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | public boolean hasNext() | It returns true if the iterator has more elements otherwise it returns false. |
| 2 | public Object next() | It returns the element and moves the cursor pointer to the next element. |
| 3 | public void remove() | It removes the last elements returned by the iterator. It is less used. |

## Iterable Interface

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

It contains only one abstract method. i.e.,

1. Iterator<T> iterator()

It returns the iterator over the elements of type T.

## Collection Interface

The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.

Some of the methods of Collection interface are Boolean add ( Object obj), Boolean addAll ( Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

## List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface, we must use :

1. List <data-type> list1= **new** ArrayList();
2. List <data-type> list2 = **new** LinkedList();
3. List <data-type> list3 = **new** Vector();
4. List <data-type> list4 = **new** Stack();

There are various methods in List interface that can be used to insert, delete, and access the elements from the list.

The classes that implement the List interface are given below.

## ArrayList

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

1. **import** java.util.\*;
2. **class** TestJavaCollection1{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> list=**new** ArrayList<String>();//Creating arraylist
5. list.add("Ravi");//Adding object in arraylist
6. list.add("Vijay");
7. list.add("Ravi");
8. list.add("Ajay");
9. //Traversing list through Iterator
10. Iterator itr=list.iterator();
11. **while**(itr.hasNext()){
12. System.out.println(itr.next());
13. }
14. }
15. }

Output:

Ravi

Vijay

Ravi

Ajay

## LinkedList

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection2{
3. **public** **static** **void** main(String args[]){
4. LinkedList<String> al=**new** LinkedList<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ravi");
8. al.add("Ajay");
9. Iterator<String> itr=al.iterator();
10. **while**(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }

Output:

Ravi

Vijay

Ravi

Ajay

## Vector

Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection3{
3. **public** **static** **void** main(String args[]){
4. Vector<String> v=**new** Vector<String>();
5. v.add("Ayush");
6. v.add("Amit");
7. v.add("Ashish");
8. v.add("Garima");
9. Iterator<String> itr=v.iterator();
10. **while**(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }

Output:

Ayush

Amit

Ashish

Garima

## Stack

The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection4{
3. **public** **static** **void** main(String args[]){
4. Stack<String> stack = **new** Stack<String>();
5. stack.push("Ayush");
6. stack.push("Garvit");
7. stack.push("Amit");
8. stack.push("Ashish");
9. stack.push("Garima");
10. stack.pop();
11. Iterator<String> itr=stack.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

Output:

Ayush

Garvit

Amit

Ashish

## Queue Interface

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

Queue interface can be instantiated as:

1. Queue<String> q1 = **new** PriorityQueue();
2. Queue<String> q2 = **new** ArrayDeque();

There are various classes that implement the Queue interface, some of them are given below.

## PriorityQueue

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection5{
3. **public** **static** **void** main(String args[]){
4. PriorityQueue<String> queue=**new** PriorityQueue<String>();
5. queue.add("Amit Sharma");
6. queue.add("Vijay Raj");
7. queue.add("JaiShankar");
8. queue.add("Raj");
9. System.out.println("head:"+queue.element());
10. System.out.println("head:"+queue.peek());
11. System.out.println("iterating the queue elements:");
12. Iterator itr=queue.iterator();
13. **while**(itr.hasNext()){
14. System.out.println(itr.next());
15. }
16. queue.remove();
17. queue.poll();
18. System.out.println("after removing two elements:");
19. Iterator<String> itr2=queue.iterator();
20. **while**(itr2.hasNext()){
21. System.out.println(itr2.next());
22. }
23. }
24. }

Output:

head:Amit Sharma

head:Amit Sharma

iterating the queue elements:

Amit Sharma

Raj

JaiShankar

Vijay Raj

after removing two elements:

Raj

Vijay Raj

## Deque Interface

Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

Deque can be instantiated as:

1. Deque d = **new** ArrayDeque();

## ArrayDeque

ArrayDeque class implements the Deque interface. It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.

ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection6{
3. **public** **static** **void** main(String[] args) {
4. //Creating Deque and adding elements
5. Deque<String> deque = **new** ArrayDeque<String>();
6. deque.add("Gautam");
7. deque.add("Karan");
8. deque.add("Ajay");
9. //Traversing elements
10. **for** (String str : deque) {
11. System.out.println(str);
12. }
13. }
14. }

Output:

Gautam

Karan

Ajay

## Set Interface

Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

1. Set<data-type> s1 = **new** HashSet<data-type>();
2. Set<data-type> s2 = **new** LinkedHashSet<data-type>();
3. Set<data-type> s3 = **new** TreeSet<data-type>();

## HashSet

HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection7{
3. **public** **static** **void** main(String args[]){
4. //Creating HashSet and adding elements
5. HashSet<String> set=**new** HashSet<String>();
6. set.add("Ravi");
7. set.add("Vijay");
8. set.add("Ravi");
9. set.add("Ajay");
10. //Traversing elements
11. Iterator<String> itr=set.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

Output:

Vijay

Ravi

Ajay

## LinkedHashSet

LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection8{
3. **public** **static** **void** main(String args[]){
4. LinkedHashSet<String> set=**new** LinkedHashSet<String>();
5. set.add("Ravi");
6. set.add("Vijay");
7. set.add("Ravi");
8. set.add("Ajay");
9. Iterator<String> itr=set.iterator();
10. **while**(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }

Output:

Ravi

Vijay

Ajay

## SortedSet Interface

SortedSet is the alternate of Set interface that provides a total ordering on its elements. The elements of the SortedSet are arranged in the increasing (ascending) order. The SortedSet provides the additional methods that inhibit the natural ordering of the elements.

The SortedSet can be instantiated as:

1. SortedSet<data-type> set = **new** TreeSet();

## TreeSet

Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.

Consider the following example:

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection9{
3. **public** **static** **void** main(String args[]){
4. //Creating and adding elements
5. TreeSet<String> set=**new** TreeSet<String>();
6. set.add("Ravi");
7. set.add("Vijay");
8. set.add("Ravi");
9. set.add("Ajay");
10. //traversing elements
11. Iterator<String> itr=set.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

Output:

Ajay

Ravi

Vijay

### ArrayList add

**I**

**Prototype:** boolean add (E e)  
**Parameters:** e=> Element to be added to the ArrayList.  
**Return Value:** true=> Element successfully added.  
**Description:** Adds the given element e to the end of the list.

**II.**

**Prototype:** void add (int index, E element)

**Parameters:**

index=> Position at which the element is to be added.  
Element=> Element to be added to the ArrayList.

**Return Value:** void

**Description:** Adds given element ‘element’ at the specified position ‘index’ by shifting the element at that position and subsequent elements to the right.

**Exceptions:** IndexOutOfBoundsException => If the specified index is out of the range.

### ArrayList addAll

**I**

**Prototype:** boolean addAll (Collection<? extends E> c)  
**Parameters:** c=> Collection whose elements are to be added to the ArrayList.  
**Return Value:** true=> If the operation has altered the ArrayList.

**Description:** Adds all the elements in the given collection c to the end of the list. The result of the operation is undefined if the collection is altered when the operation is in progress.

**Exceptions:** NullPointerException => If given collection c is null.

**II**

**Prototype:** boolean addAll (int index, Collection<? extends E> c)  
**Parameters:** index=> Position at which the elements in the given collection are to be added.  
**Return Value:** true=> If the list has changed as a result of the operation.

**Description:** Adds all the elements in the given collection c at the position specified by the ‘index’ in the list. The element at the specified index and subsequent elements are shifted to the right. The result of the operation is undefined if the collection being added is altered when the operation is in progress.

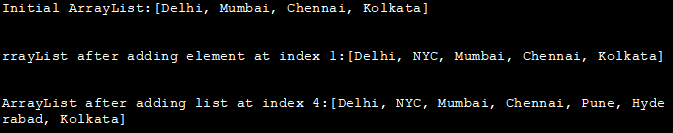
**Exceptions:** IndexOutOfBoundsException: if the index where the collection is to be added is out of bounds  
NullPointerException: if the given collection c is null.

**The following Java program demonstrates the usage of add and addAll methods.**

|  |
| --- |
| **import** java.util.\*;  **class** Main{  **public** **static** **void** main(String args[]){        //create an ArrayList        ArrayList<String> city\_List=**new** ArrayList<String>();        //add elements to the ArrayList using add method        city\_List.add("Delhi");        city\_List.add("Mumbai");        city\_List.add("Chennai");        city\_List.add("Kolkata");        //print the list        System.out.println("Initial ArrayList:" + city\_List);        //add an element at index 1 using add method overload        city\_List.add(1, "NYC");        //print the list        System.out.println("\nrrayList after adding element at index 1:" + city\_List);        //define a second list        ArrayList<String> more\_Cities = **new** ArrayList<String>(Arrays.asList("Pune", "Hyderabad"));        //use addAll method to add the list to ArrayList at index 4        city\_List.addAll(4,more\_Cities);        //print the list        System.out.println("\nArrayList after adding list at index 4:" + city\_List);     }  } |

**Output:**

Initial ArrayList:[Delhi, Mumbai, Chennai, Kolkata]  
rrayList after adding element at index 1:[Delhi, NYC, Mumbai, Chennai, Kolkata]  
ArrayList after adding list at index 4:[Delhi, NYC, Mumbai, Chennai, Pune, Hyderabad, Kolkata]

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/03/1-7.png)

The above program uses both the versions of the add method to add elements to the list. It also adds a collection to the list at the specified index. Note the shifting of elements to the right of the ArrayList as evident from the output of the program.

### ArrayList Add To The Front

As already mentioned, the first version of the add method adds the elements to the end of the list. If you want to add the elements at the beginning of the ArrayList, then you have to make use of the second version of the add method. This add method takes an index as a parameter. This index is the position at which the element is to be added.

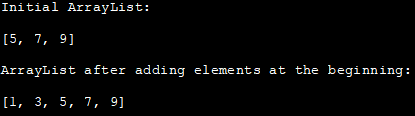
Thus to add the element at the beginning of the list, you have to specify the index as 0 which is the start of the list.

**The following program adds an element to the front of the ArrayList.**

|  |
| --- |
| **import** java.util.ArrayList;  **public** **class** Main {  **public** **static** **void** main(String[] args) {          //define new ArrayList and initialize it          ArrayList<Integer> numList = **new** ArrayList<Integer>();          numList.add(5);          numList.add(7);          numList.add(9);          //print the ArrayList          System.out.println("Initial ArrayList:");          System.out.println(numList);            //use add method with index=0 to add elements to the beginning of the list          numList.add(0, 3);          numList.add(0, 1);            System.out.println("ArrayList after adding elements at the beginning:");          //print ArrayList          System.out.println(numList);      }  } |

**Output:**

Initial ArrayList:  
[5, 7, 9]  
ArrayList after adding elements at the beginning:  
[1, 3, 5, 7, 9]

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/03/2-6.png)

### ArrayList remove

**I.**

**Prototype:** E remove (int index)  
**Parameters:** index=> Position at which the element is to be removed from the ArrayList.  
**Return Value:** E=> Element that is deleted  
**Description:** Deletes element at the ‘index’ in the ArrayList and moves subsequent elements to the left.  
**Exceptions:** IndexOutOfBoundsException => Index specified is out of range.

**II.**

**Prototype:** boolean remove (Object o)  
**Parameters:** o=> Element that is to be removed from the ArrayList.  
**Return Value:** true=> If the element is present in the list.

**Description:** Deletes the first occurrence of element o from the list. If the element is not present in the list, then there is no effect of this operation. Once the element is deleted, the subsequent elements are shifted to the left.

### ArrayList removeAll

**Prototype:** boolean removeAll (Collection<?> c)  
**Parameters:** c=> Collection whose elements match with those of ArrayList and are to be removed.  
**Return Value:** true=> If the ArrayList is altered by the operation.

**Description:** Removes all the elements from the list that match the elements in the given collection c. As a result, the elements remaining are shifted to the left of the list.

**Exceptions:** ClassCastException => Class is not the same as that of the specified collection which implies class is incompatible.  
NullPointerException => If the given collection c is null; or if c has a null element and it is not allowed by the collection.

### ArrayList removeRange

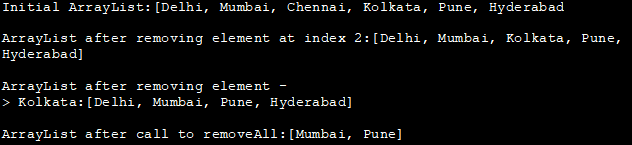
**Prototype:** protected void removeRange (int fromIndex, int toIndex)  
**Parameters:** fromIndex=> Index of the starting element of the range to be removed.  
toIndex=> Index of the element after the last element in the range to be removed.  
**Return Value:** void  
**Description:** Removes elements specified in the given range, fromIndex (inclusive) to toIndex (exclusive) from the list. This operation shortens the length of the list by (toIndex-fromIndex). This operation has no effect in case fromIndex = toIndex.  
**Exceptions:** IndexOutOfBoundsException=> If any of the indices (fromIndex or toIndex) is out of bounds.

**Let us implement a Java program to demonstrate some of these remove methods that we discussed above.**

|  |
| --- |
| **import** java.util.\*;  **class** Main{  **public** **static** **void** main(String args[]){        //create an ArrayList        ArrayList<String> city\_List=**new** ArrayList<String>(Arrays.asList("Delhi","Mumbai","Chennai",                             "Kolkata", "Pune", "Hyderabad"));        //print the list        System.out.println("Initial ArrayList:" + city\_List);          //remove element at index 2        city\_List.remove(2);        //print the list        System.out.println("\nArrayList after removing element at index 2:" + city\_List);          //remove the element "Kolkata"        city\_List.remove("Kolkata");        //print the list        System.out.println("\nArrayList after removing element -> Kolkata:" + city\_List);          //create new list        ArrayList<String> newCities=**new** ArrayList<String>(Arrays.asList("Delhi","Hyderabad"));        //call removeAll to remove elements contained in newCities list.        city\_List.removeAll(newCities);        //print the list        System.out.println("\nArrayList after call to removeAll:" + city\_List);     }  } |

**Output:**

Initial ArrayList:[Delhi, Mumbai, Chennai, Kolkata, Pune, Hyderabad  
ArrayList after removing element at index 2:[Delhi, Mumbai, Kolkata, Pune, Hyderabad]  
ArrayList after removing element -> Kolkata:[Delhi, Mumbai, Pune, Hyderabad]  
ArrayList after call to removeAll:[Mumbai, Pune]

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/03/3-6.png)

### ArrayList size (Length)

**Prototype:** int size ()  
**Parameters:** NIL  
**Return Value:** int=> Number of elements in the ArrayList.  
**Description:** Returns the total number of elements or the length of the ArrayList.

**EnsureCapacity**

**Prototype:** void ensureCapacity (int minCapacity)  
**Parameters:** minCapacity=> The minimum capacity desired for the ArrayList.  
**Return Value:** void  
**Description:** Increases the capacity of the ArrayList to ensure that it has the minCapacity.

**trimToSize**

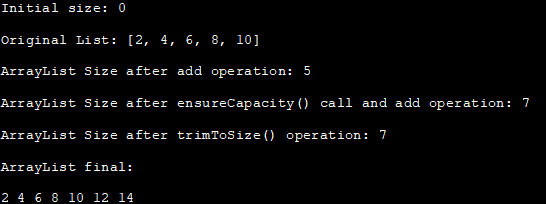
**Prototype:** void trimToSize()  
**Parameters:** NIL  
**Return Value:** void  
**Description:** Trims the ArrayList capacity to the size or number of elements present in the list.

**The below programming example demonstrates the methods size (), ensureCapacity () and trimToSize ().**

|  |
| --- |
| **import** java.util.ArrayList;  **public** **class** Main  {  **public** **static** **void** main(String [] args)      {          //Create and initialize Arraylist          ArrayList<Integer> evenList=**new** ArrayList<Integer>(5);          System.out.println("Initial size: "+evenList.size());          evenList.add(2);          evenList.add(4);          evenList.add(6);          evenList.add(8);          evenList.add(10);          //print the list and size          System.out.println("Original List: " + evenList);          System.out.println("ArrayList Size after add operation: "+evenList.size());          //call ensureCapacity () with minimum capacity =10          evenList.ensureCapacity(10);          //add two more elements          evenList.add(12);          evenList.add(14);          //print the size again          System.out.println("ArrayList Size after ensureCapacity() call and add operation: "+evenList.size());          //call trimToSize()          evenList.trimToSize();          //print the size and the ArrayList          System.out.println("ArrayList Size after trimToSize() operation: "+evenList.size());          System.out.println("ArrayList final: ");  **for**(**int** num: evenList){              System.out.print(num + " ");          }        }  } |

**Output:**

Initial size: 0  
Original List: [2, 4, 6, 8, 10]  
ArrayList Size after add operation: 5  
ArrayList Size after ensureCapacity() call and add operation: 7  
ArrayList Size after trimToSize() operation: 7  
ArrayList final:  
2 4 6 8 10 12 14

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/03/4-5.png)

### ArrayList contains

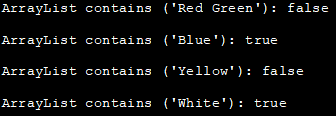
**Prototype:** boolean contains (Object o)  
**Parameters:** o=> Element which is to be checked if present in the ArrayList.  
**Return Value:** true=> If the ArrayList contains element o.  
**Description:** Checks if the list contains the given element ‘o’. Returns true if the element is present.

**We make use of the ‘contains’ method in the following program.**

|  |
| --- |
| **import** java.util.ArrayList;  **public** **class** Main {  **public** **static** **void** main(String[] args) {          //create and initialize colorsList        ArrayList<String> colorsList = **new** ArrayList<String>();        colorsList.add("Red");        colorsList.add("Green");        colorsList.add("Blue");        colorsList.add("White");          //call contains method to check if different strings are present in ArrayList        System.out.println("ArrayList contains ('Red Green'): "                                             +colorsList.contains("Red Green"));        System.out.println("ArrayList contains ('Blue'): "                                               +colorsList.contains("Blue"));        System.out.println("ArrayList contains ('Yellow'): "                                            +colorsList.contains("Yellow"));        System.out.println("ArrayList contains ('White'): "                                             +colorsList.contains("White"));     }  } |

**Output:**

ArrayList contains (‘Red Green’): false  
ArrayList contains (‘Blue’): true  
ArrayList contains (‘Yellow’): false  
ArrayList contains (‘White’): true

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/03/5-5.png)

As shown in the above output, the ‘contains’ method checks if the argument provided is present in the ArrayList and returns true or false.

### ArrayList get

**Prototype:** E get (int index)  
**Parameters:** index=> Index at which element is to be retrieved from the ArrayList.  
**Return Value:** E=> Element value at the given index in the ArrayList.  
**Description:** Returns the element in the list present at the position specified by ‘index’.  
**Exceptions:** IndexOutOfBoundsException => If index is out of bounds.

### ArrayList set (Replace element)

**Prototype:** E set (int index, E element)  
**Parameters:** index=> Index at which the element is to be replaced.  
Element=> New element to be set at the index specified.  
**Return Value:** E => Element that is replaced by the set operation.  
**Description:**Sets the element value at the given ‘index’ to the new value given by ‘element’.  
**Exceptions:** IndexOutOfBoundsException => If index is out of bounds

**The Java program below uses get () and set () method to retrieve and replace values in the ArrayList.**

|  |
| --- |
| **import** java.util.ArrayList;  **public** **class** Main {  **public** **static** **void** main(String[] args) {        //create and initialize colorsList        ArrayList<String> colorsList = **new** ArrayList<String>();        colorsList.add("Red");        colorsList.add("Green");        colorsList.add("Blue");        colorsList.add("White");        //call get () method to retrieve value at index 2        System.out.println("Entry at index 2 before call to set: " + colorsList.get(2));          //replace the value at index 2 with new value        colorsList.set(2,"Yellow");          //print the value at index 2 again        System.out.println("Entry at index 2 after call to set: " + colorsList.get(2));     }  } |

**Output:**

Entry at index 2 before call to set: Blue  
Entry at index 2 after call to set: Yellow

[Output](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/03/6-4.png)

### ArrayList clear

**Prototype:** void clear ()  
**Parameters:** NIL  
**Return Value:** void  
**Description:** Clears the list by removing all the elements from the list.

### ArrayList isEmpty

**Prototype:** boolean isEmpty ()  
**Parameters:** NIL  
**Return Value:** true=> if list is empty  
**Description:** Checks if the given list is empty.

**Clear () and isEmpty () functions are demonstrated below.**

|  |
| --- |
| **import** java.util.ArrayList;  **public** **class** Main {  **public** **static** **void** main(String[] args) {          //create and initialize colorsList        ArrayList<String> colorsList = **new** ArrayList<String>();        colorsList.add("Red");        colorsList.add("Green");        colorsList.add("Blue");        colorsList.add("White");        //print the ArrayList        System.out.println("The ArrayList: " + colorsList);        //call clear() nethod on ArrayList        colorsList.clear();        //check if ArrayList is empty using isEmpty() method        System.out.println("Is ArrayList empty after clear ()? :" + colorsList.isEmpty());     }  } |

**Output:**

The ArrayList: [Red, Green, Blue, White]  
Is ArrayList empty after clear ()? :true

[Output - isEmpty ArrayList](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/03/7-3.png)

### ArrayList indexOf

**Prototype:** int indexOf (Object o)  
**Parameters:** o=> Element whose index is to be found in the ArrayList.  
**Return Value:** int => Index of the first occurrence of the element in the list.  
**Description:** Returns the index of the first occurrence of the element o in the list. -1 if the element o is not present in the list.

### ArrayList lastIndexOf

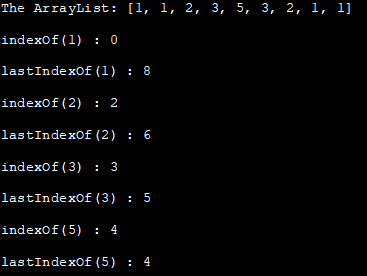
**Prototype:** int lastIndexOf (Object o)  
**Parameters:** o=> The element to be searched for.  
**Return Value:** int=> Index of the last occurrence of the element in the list.  
**Description:** Returns the index of the last occurrence of the specified element o in the list. -1 if the element is not present in the list.

**The below Java program demonstrates the indexOf and lastIndexOf methods of ArrayList.**

|  |
| --- |
| **import** java.util.ArrayList;  **public** **class** Main {  **public** **static** **void** main(String[] args) {          //create and initialize intList        ArrayList<Integer> intList = **new** ArrayList<Integer>();        intList.add(1);        intList.add(1);        intList.add(2);        intList.add(3);        intList.add(5);        intList.add(3);        intList.add(2);        intList.add(1);        intList.add(1);          //print the ArrayList        System.out.println("The ArrayList: " + intList);        //call indexOf() and lastIndexOf() methods to check the indices of specified elements        System.out.println("indexOf(1) : " + intList.indexOf(1));        System.out.println("lastIndexOf(1) : " + intList.lastIndexOf(1));        System.out.println("indexOf(2) : " + intList.indexOf(2));        System.out.println("lastIndexOf(2) : " + intList.lastIndexOf(2));         System.out.println("indexOf(3) : " + intList.indexOf(3));        System.out.println("lastIndexOf(3) : " + intList.lastIndexOf(3));         System.out.println("indexOf(5) : " + intList.indexOf(5));        System.out.println("lastIndexOf(5) : " + intList.lastIndexOf(5));     }  } |

**Output:**

The ArrayList: [1, 1, 2, 3, 5, 3, 2, 1, 1]  
indexOf(1) : 0  
lastIndexOf(1) : 8  
indexOf(2) : 2  
lastIndexOf(2) : 6  
indexOf(3) : 3  
lastIndexOf(3) : 5  
indexOf(5) : 4  
lastIndexOf(5) : 4

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/03/8-2.png)

### ArrayList toArray

**Prototype:** Object [] toArray ()  
**Parameters:** NIL  
**Return Value:** Object [] =>an array. This returned array contains all the elements of the list in a proper sequence.  
**Description:** Converts the given list into an array.

**Prototype:** <T> T[] toArray (T[] a)

**Parameters:** a=> Array to store elements of the list. If the size of the array is not enough for list elements, another array with the same type as a is created for storing elements.

**Return Value:** T[] => Array that contains all the list elements.  
**Description:** Converts the given list into an array of the type given by a.

**Exceptions:** ArrayStoreException => If there is a mismatch in runtime type of the array and runtime type or supertype of its elements.  
NullPointerException => The given array is null

**The Java program below demonstrates the toArray method of ArrayList.**

|  |
| --- |
| **import** java.util.\*;  **public** **class** Main {  **public** **static** **void** main(String[] args) {          // define and initialize ArrayList          ArrayList<Integer> intList = **new** ArrayList<Integer>();          intList.add(10);          intList.add(20);          intList.add(30);          intList.add(40);          intList.add(50);            // print ArrayList          System.out.println("ArrayList: "  + intList);            //declare array          Integer myArray[] = **new** Integer[intList.size()];          //use toArray method to convert ArrayList to Array          myArray = intList.toArray(myArray);          //print the Array          System.out.println("Array from ArrayList:" + Arrays.toString(myArray));      }  } |

**Output:**

ArrayList: [10, 20, 30, 40, 50]  
Array from ArrayList:[10, 20, 30, 40, 50]

### ArrayList clone

**Prototype:** Object clone ()  
**Parameters:** NIL  
**Return Value:** Object=> Clone of the ArrayList instance.  
**Description:** Makes a shallow copy of the given ArrayList.

|  |
| --- |
| **import** java.util.ArrayList;  **public** **class** Main {  **public** **static** **void** main(String a[]){       ArrayList<String> fruitsList = **new** ArrayList<String>();         //Adding elements to the ArrayList       fruitsList.add("Apple");       fruitsList.add("Orange");       fruitsList.add("Melon");       fruitsList.add("Grapes");       System.out.println("Original ArrayList: "+fruitsList);         ArrayList<String> clone\_list = (ArrayList<String>)fruitsList.clone();       System.out.println("Cloned ArrayList: "+ clone\_list);         //add one elmeent & remove one element from original arraylist       fruitsList.add("Mango");       fruitsList.remove("Orange");         //print original and cloned ArrayList again       System.out.println("\nOriginal ArrayList after add & remove:"+fruitsList);       System.out.println("Cloned ArrayList after original changed:"+clone\_list);     }  } |

**Output:**

Original ArrayList: [Apple, Orange, Melon, Grapes]  
Cloned ArrayList: [Apple, Orange, Melon, Grapes]  
Original ArrayList after add & remove:[Apple, Melon, Grapes, Mango]  
Cloned ArrayList after original changed:[Apple, Orange, Melon, Grapes]

From the above program output, you can see that the cloned ArrayList is a shallow copy of the original ArrayList. This means that when the original ArrayList is changed, these changes do not reflect in the cloned ArrayList as they do not share the memory locations of each element.

For making a deep copy of Array, the original ArrayList needs to be traversed and each of its elements needs to be copied to the destination ArrayList.

### ArrayList subList

**Prototype:** List<E> subList (int fromIndex, int toIndex)  
**Parameters:** fromIndex=> Starting index of the range (inclusive)  
toIndex=> End index of the range (exclusive)  
**Return Value:** List<E> => Sublist of the list in the given range.

**Description:** Returns a subList between a given range, fromIndex to index for the given list. Note that this sublist or the view of the list in the given range supports all the operations supported by the list. No view is returned if fromIndex = toIndex.

**Exceptions:** IndexOutOfBoundsException => Thrown when toIndex is out of range.  
IllegalArgumentException=> If fromIndex > toIndex i.e. indices are out of order.

**Let us see an example of the subList method.**

|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.List;  **class** Main{  **public** **static** **void** main(String a[]){       //create and initialize the ArrayList       ArrayList<Integer> intList = **new** ArrayList<Integer>();       intList.add(5);       intList.add(10);       intList.add(15);       intList.add(20);       intList.add(25);       intList.add(30);       intList.add(35);       intList.add(40);       intList.add(45);       intList.add(50);         //print the ArrayList       System.out.println("Original ArrayList: "+intList);         //create a sublist for the given ArrayList       ArrayList<Integer> sub\_ArrayList = **new** ArrayList<Integer>(intList.subList(2, 6));       //print the sublist       System.out.println("Sublist of given ArrayList: "+sub\_ArrayList);    }  } |

**Output:**

Original ArrayList: [5, 10, 15, 20, 25, 30, 35, 40, 45, 50]  
Sublist of given ArrayList: [15, 20, 25, 30]

[Output](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/03/11-1.png)

### ArrayList retainAll

**Prototype:** boolean retainAll (Collection<?> c)  
**Parameters:** c=> Collection with elements that are to be retained in the list.  
**Return Value:** true=> If the ArrayList changed as a result of the operation.  
**Description:** Retains those elements in the list that match the elements in the given collection c.

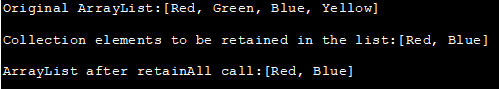
**Exceptions:** ClassCastException => The collection type and list type do not match  
NullPointerException => Given collection is null or the list contains null element and collection does not permit nulls.

**The following program demonstrates the retainAll method.**

|  |
| --- |
| **import** java.util.\*;  **class** Main{  **public** **static** **void** main(String args[]){    //create and initialize ArrayList    ArrayList<String> colorsList=**new** ArrayList<String>();    colorsList.add("Red");    colorsList.add("Green");    colorsList.add("Blue");    colorsList.add("Yellow");    //print the ArrayList    System.out.println("Original ArrayList:" + colorsList);      //define another collection    ArrayList<String> color\_collection=**new** ArrayList<String>();    color\_collection.add("Red");    color\_collection.add("Blue");    System.out.println("Collection elements to be retained in the list:" + color\_collection);      //call retainAll method with above collection as an argument    colorsList.retainAll(color\_collection);    //print the ArrayList after retainAll call.    System.out.println("ArrayList after retainAll call:" + colorsList);   }  } |

**Output:**

Original ArrayList:[Red, Green, Blue, Yellow]  
Collection elements to be retained in the list:[Red, Blue]  
ArrayList after retainAll call:[Red, Blue]

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/03/12-1.png)

### ArrayList Iterator

**Prototype:** Iterator<E> iterator ()  
**Parameters:** NIL  
**Return Value:** Iterator <E> => iterator over the list elements.  
**Description:** Returns an iterator to traverse over the list elements in the proper sequence.

### ArrayList listIterator

**I.**

**Prototype:** ListIterator<E> listIterator ()  
**Parameters:** NIL  
**Return Value:** ListIterator <E> => listIterator over the list elements.  
**Description:** Returns list iterator to traverse over the elements of the given list.

**II.**

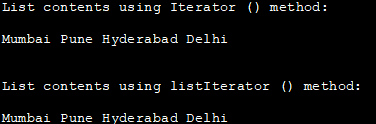
**Prototype:** ListIterator<E> listIterator (int index)  
**Parameters:** index=> Position of the first element in the listIterator.  
**Return Value:** ListIterator<E> => ListIterator for the list from specified index.  
**Description:** Returns the list iterator starting from the specified position ‘index’ to traverse over the elements of the given list.  
**Exceptions:** IndexOutOfBoundsException => Given index is out of range.

**Example of iterator () and listIterator () methods.**

|  |
| --- |
| **import** java.util.\*;  **class** Main{  **public** **static** **void** main(String args[]){          //create ArrayList and initialize it          ArrayList<String> cities=**new** ArrayList<String>();          cities.add("Mumbai");          cities.add("Pune");          cities.add("Hyderabad");          cities.add("Delhi");            //use iterator() method to traverse through the list          System.out.println("List contents using Iterator () method:");          Iterator iter=cities.iterator();  **while**(iter.hasNext()){              System.out.print(iter.next() + " ");          }            //use listIterator() method to traverse through the list          System.out.println("\n\nList contents using listIterator () method:");          ListIterator<String> list\_iter=cities.listIterator();  **while**(list\_iter.hasNext())  {            System.out.print(list\_iter.next() + " ");          }    }  } |

**Output:**

List contents using Iterator () method:  
Mumbai Pune Hyderabad Delhi  
List contents using listIterator () method:  
Mumbai Pune Hyderabad Delhi

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/03/13-1.png)

### Add Array To ArrayList In Java

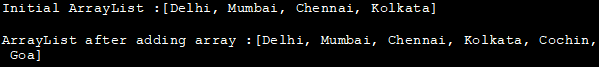
ArrayList supports the addAll method to add elements of the collection to the ArrayList. In a similar manner, you can also add an Array to the ArrayList. This is done using the ‘Collections.addAll’ method.

**Example of adding an Array to the ArrayList.**

|  |
| --- |
| **import** java.util.\*;  **class** Main{  **public** **static** **void** main(String args[]){        //create an ArrayList        ArrayList<String> city\_List=**new** ArrayList<String>();        //add elements to the ArrayList using add method        city\_List.add("Delhi");        city\_List.add("Mumbai");        city\_List.add("Chennai");        city\_List.add("Kolkata");        //print ArrayList        System.out.println("\nInitial ArrayList :" + city\_List);          //define an array.        String[] myArray = **new** String[]{"Cochin", "Goa"};        //add the array to the ArrayList        Collections.addAll(city\_List,myArray);          //print the ArrayList        System.out.println("\nArrayList after adding array :" + city\_List);     }  } |

**Output:**

Initial ArrayList :[Delhi, Mumbai, Chennai, Kolkata]  
ArrayList after adding array :[Delhi, Mumbai, Chennai, Kolkata, Cochin, Goa]

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/03/14.1-1.png)

### Sort ArrayList In Java

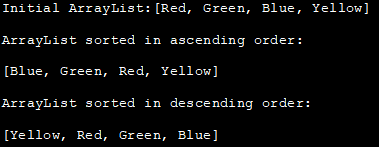
ArrayList uses the Collections.sort method to sort its elements. By default, the list is sorted in ascending order by the Collections.sort method. If the ArrayList is to be sorted in descending order, then you have to provide ‘Collections.reverseOrder()’ a parameter to the sort method.

**Given below is a program to sort an ArrayList in ascending and descending order:**

|  |
| --- |
| **import** java.util.\*;  **public** **class** Main  {  **public** **static** **void** main(String args[]){          //Create and initialize an ArrayList         ArrayList<String> colorsList = **new** ArrayList<String>();         colorsList.add("Red");         colorsList.add("Green");         colorsList.add("Blue");         colorsList.add("Yellow");           //print initial ArrayList         System.out.println("Initial ArrayList:" + colorsList);           //sort ArrayList in ascending order         Collections.sort(colorsList);           //print sorted ArrayList         System.out.println("\nArrayList sorted in ascending order:");         System.out.println(colorsList);            //sort ArrayList in reverse(desending) order          Collections.sort(colorsList, Collections.reverseOrder());            //print sorted list          System.out.println("\nArrayList sorted in descending order:");          System.out.println(colorsList);      }  } |

**Output:**

Initial ArrayList:[Red, Green, Blue, Yellow]  
ArrayList sorted in ascending order:  
[Blue, Green, Red, Yellow]  
ArrayList sorted in descending order:  
[Yellow, Red, Green, Blue]

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/03/15.png)

In case the ArrayList contains other class objects as elements, then you can make use of Comparable and Comparator interfaces. More details about interfaces will be covered in our later tutorials.

### Reverse An ArrayList In Java

You can also reverse an ArrayList in Java. One method to do this is to use the traditional method of traversing the ArrayList in the reverse order and copy each element to a new ArrayList.

Another method is using the Collections class which provides the ‘reverse’ method that is used to reverse a collection.

**The program to reverse an ArrayList using the Collections class is given below.**

|  |
| --- |
| **import** java.io.\*;  **import** java.util.\*;    **public** **class** Main {  **public** **static** **void** main(String[] args)      {          // create and initialize an ArrayList          ArrayList<Integer> oddList = **new** ArrayList<Integer>();          oddList.add(1);          oddList.add(3);          oddList.add(5);          oddList.add(7);          oddList.add(9);          System.out.print("Initial ArrayList: " + oddList);            // use Collections.reverse method to reverse the ArrayList          Collections.reverse(oddList);          //print the ArrayList          System.out.print("\nReversed ArrayList: " + oddList);     }  } |

**Output:**

Initial ArrayList: [1, 3, 5, 7, 9]  
Reversed ArrayList: [9, 7, 5, 3, 1]

[Output - Reverse and ArrayList](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/03/16.png)

### Remove Duplicates From An ArrayList In Java

To remove duplicates from the ArrayList, you can once again resort to the traditional method of using an iterator to traverse through the ArrayList and store only the first occurrence of the element into a different ArrayList.

Yet another method is by using the ‘distinct ()’ method of stream () class. This method returns a stream of distinct elements. The stream () feature is available in Java from Java 8 onwards.

**The implementation of stream ().distinct () method is given below:**

|  |
| --- |
| **import** java.util.\*;  **import** java.util.stream.Collectors;    **public** **class** Main {  **public** **static** **void** main(String[] args)  {          // Create an ArrayList of numbers          ArrayList<Integer> numList = **new** ArrayList<>          (Arrays.asList(1, 2, 3, 1, 3, 5, 5, 6, 6, 7, 7, 8, 8));            //print the original ArrayList          System.out.println("Original ArrayList:" + numList);            //Use Java 8 stream().distinct()  method to remove duplicates from the list          List<Integer> distinctList = numList.stream().distinct().collect(Collectors.toList());          //print the new list          System.out.println("ArrayList without duplicates:" + distinctList);      }  } |

**Output:**

Original ArrayList:[1, 2, 3, 1, 3, 5, 5, 6, 6, 7, 7, 8, 8]  
ArrayList without duplicates:[1, 2, 3, 5, 6, 7, 8]

[Output - Remove duplicates from ArrayList](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/03/17.png)

### Shuffle (Randomize) An ArrayList In Java

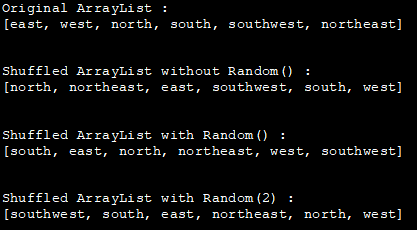
You can also ‘shuffle’ or randomize the ArrayList elements. This is done using the Collections.shuffle () method. Using this method, either you can shuffle the ArrayList with default settings or provide a random () function that will randomize the elements according to the random value provided.

**A Java program to achieve this is given below.**

|  |
| --- |
| **import** java.util.\*;    **public** **class** Main {  **public** **static** **void** main(String[] args)      {          //create and initialize a String ArrayList          ArrayList<String> strlist = **new** ArrayList<String>();          strlist.add("east");          strlist.add("west");          strlist.add("north");          strlist.add("south");          strlist.add("southwest");          strlist.add("northeast");          //print the original list          System.out.println("Original ArrayList : \n" + strlist);            //shuffle the ArrayList without random function          Collections.shuffle(strlist);          System.out.println("\nShuffled ArrayList without Random() : \n"                             + strlist);            // shuffle the ArrayList with random() function          Collections.shuffle(strlist, **new** Random());          System.out.println("\nShuffled ArrayList with Random() : \n" + strlist);            // use random (2) to shuffle the ArrayList          Collections.shuffle(strlist, **new** Random(2));          System.out.println("\nShuffled ArrayList with Random(2) : \n" + strlist);      }  } |

**Output:**

Original ArrayList :[east, west, north, south, southwest, northeast] Shuffled ArrayList without Random() :[north, northeast, east, southwest, south, west]  
Shuffled ArrayList with Random() :[south, east, north, northeast, west, southwest]  
Shuffled ArrayList with Random(2) :[southwest, south, east, northeast, north, west]

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/03/18.png)