**JAVA NOTESs**

* 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).

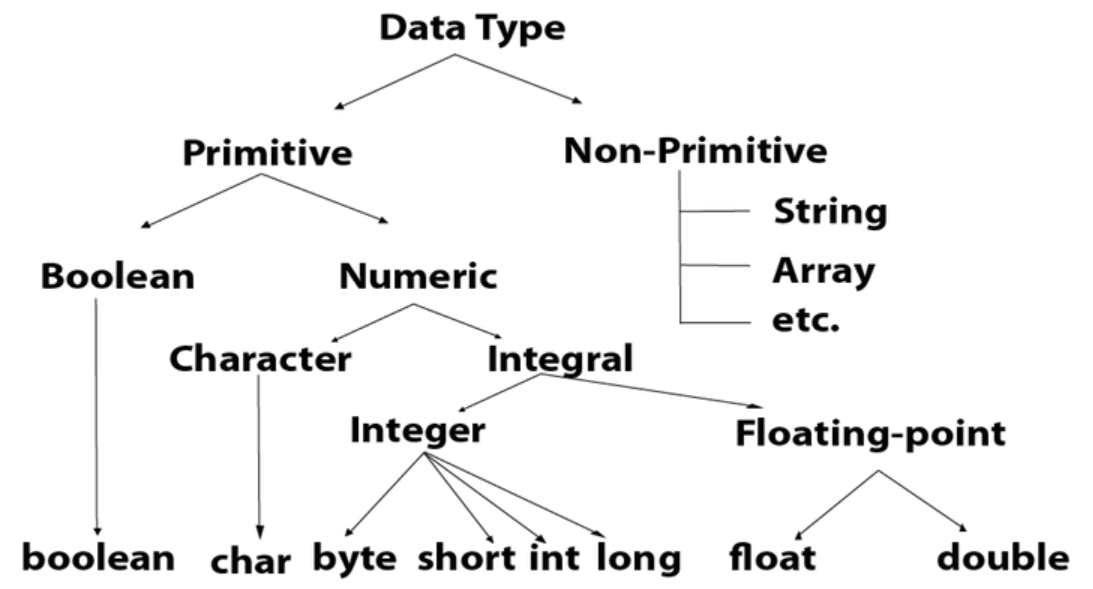
The bytecode generated is a non-executable code and needs an interpreter to execute on a machine. This interpreter is the JVM and thus the Bytecode is executed by the JVM.

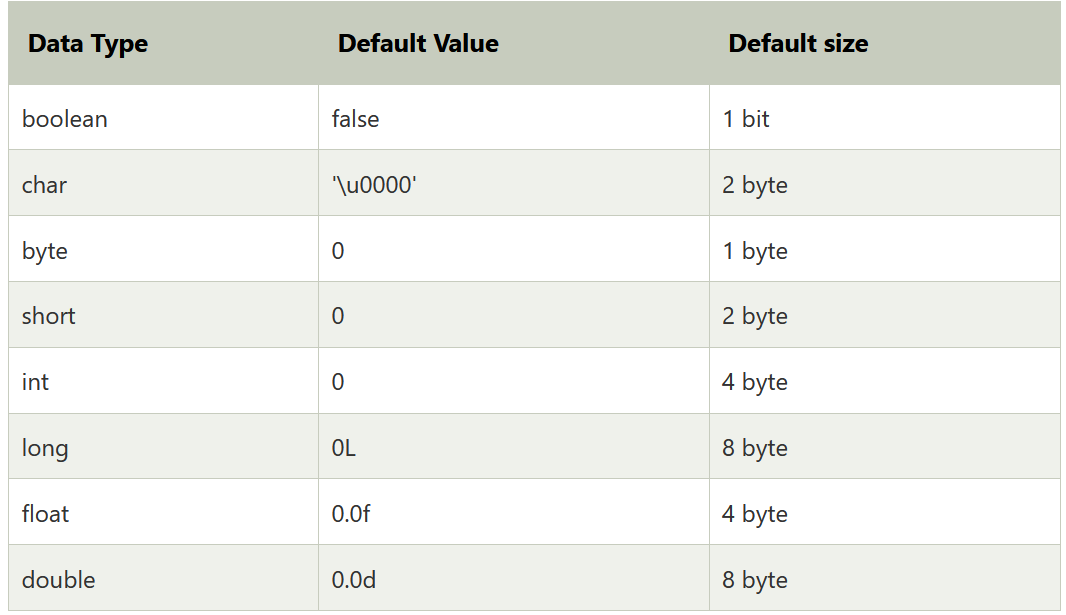
And finally program runs to give the desired output.

**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

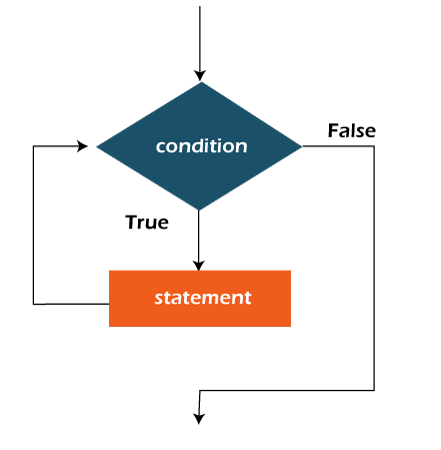
2. Update expression: Every time the loop body is executed, this expression increments or decrements loop variable.

**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**

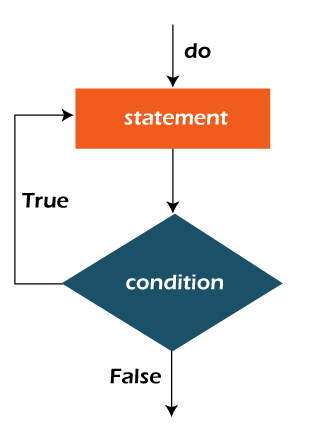
2. Update expression: Every time the loop body is executed, the this expression increments or decrements loop variable.

**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.