What Are Features or Properties of Java Language

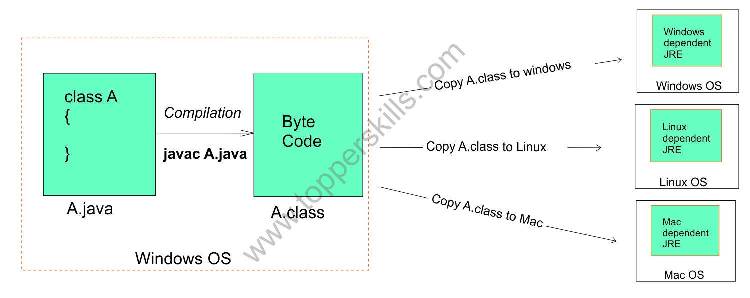
**Simple:-**

The Java is simple because there are not complex syntaxes, any C++ or other object oriented programmer can easily understand the Java syntaxes. There are no explicit pointer and operator overloading concepts in java. Read [Java vs C++.](https://www.topperskills.com/tutorials/java/difference-between-java-c++.html)

**Object Oriented:-**

Java follows the [Object Oriented Programming Model;](https://www.topperskills.com/tutorials/oop/object-oriented-programming-basic-concepts.html) we cannot write a java program without a class therefore java is an Object Oriented programming language. Java implements the Object Oriented concepts like [Abstraction,](https://www.topperskills.com/tutorials/oop/object-oriented-programming-abstraction-concepts.html) [Encapsulation,](https://www.topperskills.com/tutorials/oop/object-oriented-programming-encapsulation-concepts.html) [Inheritance,](https://www.topperskills.com/tutorials/oop/object-oriented-programming-inheritance-concepts.html) and [Polymorphism.](https://www.topperskills.com/tutorials/oop/object-oriented-programming-polymorphism-concepts.html)

**Platform Independent and Architecture Neutral:-**



Java Compiler converts the source code into the byte code, that byte code is platform independent and architecture neutral, means the java compiler does not consider the underlying operating system and processor architecture while creating the byte code. Once we compiled the java source code we do not need to recompile the same code on another machine, this makes java Write Once Run Anywhere (WORA).

For instance, if we developed and compiled a program on the Windows operating system, then the created byte code (.class file) can be run on other operating systems like Linux or Mac without recompiling the same program.

**Portability:-**

The byte code of the java programs is platform independent and architecture neutral, therefore we can easily run the same byte code on different platforms. For instance, we can develop and compile a java program on Linux operating systems and the compiled byte code can be easily run on any operating systems like Windows, Macintosh, etc. because of this we can easily port the java program on different computers.

***Portability= Platform Independent + Architecture Neutral.***

**High Performance:-**

Java JVM uses the [Just In Time compiler (JIT)](https://www.topperskills.com/tutorials/java/java-jvm-internal-architecture-structure.html)which converts the similar statements from byte code to processor dependent machine code at a time, this reduces the processor time consumption to run the program and achieves high performance.

**Multithreaded:-**

Java program can implement the multithreading concept. In fact every java program has two threads, one is main thread which is responsible to run the program and there is a background daemon thread which is called Garbage Collector which is responsible for destroying the unused objects to free the memory.

**Secure:-**

Java does not support the explicit pointer and pointer arithmetic.

Ex. Explicit pointer :- int \*ptr= &a;

Ex. Pointer Arithmetic:- ptr++, ptr--

Using pointer arithmetic one can easily access the value of other memory locations which can be the security loop hole.

Other reason of java being secure is, to run the program Java does not use runtime environment of underlying operating system like C++ uses, and instead it uses its own runtime environment that is JRE – Java Runtime Environment.

**Robust:-**

The java is more robust than other programming languages because java has inbuilt Garbage Collection feature implemented, therefore the possibility of the java program running out of memory is very low.

Java has strong error handling mechanism; java force the programmer to handle the exceptions at compile time, so the chances of raising exceptions at run time is less.

**Distributed:-**

The Java can be used to develop the distributed applications using the concepts like RMI, EJB, etc. A distributed application is one in which there can be multiple clients and multiple servers to serve the requests of these clients. If a server faces any technical problem then the remaining servers will serve the clients request; therefore client will never face problem of server unavailability.

**Interpreted:-**

The java is highly interpreted programming language means that it does not require operating system`s runtime to convert byte code to machine code. The Java has its own interpreter to convert the byte code into processor dependent machine language.

# C++ vs Java

There are many differences and similarities between the [C++ programming](https://www.javatpoint.com/cpp-tutorial) language and [Java](https://www.javatpoint.com/java-tutorial). A list of top differences between C++ and Java are given below:

|  |  |  |
| --- | --- | --- |
| **Comparison Index** | **C++** | **Java** |
| **Platform-independent** | C++ is platform-dependent. | Java is platform-independent. |
| **Mainly used for** | C++ is mainly used for system programming. | Java is mainly used for application programming. It is widely used in Windows-based, web-based, enterprise, and mobile applications. |
| **Design Goal** | C++ was designed for systems and applications programming. It was an extension of the [C programming language](https://www.javatpoint.com/c-programming-language-tutorial). | Java was designed and created as an interpreter for printing systems but later extended as a support network computing. It was designed to be easy to use and accessible to a broader audience. |
| **Goto** | C++ supports the [goto](https://www.javatpoint.com/cpp-goto-statement) statement. | Java doesn't support the goto statement. |
| **Multiple inheritance** | C++ supports multiple inheritance. | Java doesn't support multiple inheritance through class. It can be achieved by using [interfaces in java](https://www.javatpoint.com/interface-in-java). |
| **Operator Overloading** | C++ supports [operator overloading](https://www.javatpoint.com/cpp-overloading). | Java doesn't support operator overloading. |
| **Pointers** | C++ supports [pointers](https://www.javatpoint.com/cpp-pointers). You can write a pointer program in C++. | Java supports pointer internally. However, you can't write the pointer program in java. It means java has restricted pointer support in java. |
| **Compiler and Interpreter** | C++ uses compiler only. C++ is compiled and run using the compiler which converts source code into machine code so, C++ is platform dependent. | Java uses both compiler and interpreter. Java source code is converted into bytecode at compilation time. The interpreter executes this bytecode at runtime and produces output. Java is interpreted that is why it is platform-independent. |
| **Call by Value and Call by reference** | C++ supports both call by value and call by reference. | Java supports call by value only. There is no call by reference in java. |
| **Structure and Union** | C++ supports structures and unions. | Java doesn't support structures and unions. |
| **Thread Support** | C++ doesn't have built-in support for threads. It relies on third-party libraries for thread support. | Java has built-in [thread](https://www.javatpoint.com/multithreading-in-java) support. |
| **Documentation comment** | C++ doesn't support documentation comments. | Java supports documentation comment (/\*\* ... \*/) to create documentation for java source code. |
| **Virtual Keyword** | C++ supports virtual keyword so that we can decide whether or not to override a function. | Java has no virtual keyword. We can override all non-static methods by default. In other words, non-static methods are virtual by default. |
| **unsigned right shift >>>** | C++ doesn't support >>> operator. | Java supports unsigned right shift >>> operator that fills zero at the top for the negative numbers. For positive numbers, it works same like >> operator. |
| **Inheritance Tree** | C++ always creates a new inheritance tree. | Java always uses a single inheritance tree because all classes are the child of the Object class in Java. The Object class is the root of the [inheritance](https://www.javatpoint.com/inheritance-in-java) tree in java. |
| **Hardware** | C++ is nearer to hardware. | Java is not so interactive with hardware. |
| **Object-oriented** | C++ is an object-oriented language. However, in the C language, a single root hierarchy is not possible. | Java is also an [object-oriented](https://www.javatpoint.com/java-oops-concepts) language. However, everything (except fundamental types) is an object in Java. It is a single root hierarchy as everything gets derived from java.lang.Object. |

**Note**

* Java doesn't support default arguments like C++.
* Java does not support header files like C++. Java uses the import keyword to include different classes and methods.

## C++ Program Example

File: main.cpp

1. #include <iostream>
2. **int** main() {
3. cout << "Hello C++ Programming";
4. **return** 0;
5. }

**Output:**

Hello C++ Programming

## Java Program Example

File: Simple.java

1. **class** Simple{
2. **public** **static** **void** main(String args[]){
3. System.out.println("Hello Java");
4. }
5. }

**Output:**

Hello Java

# Java Interpreter

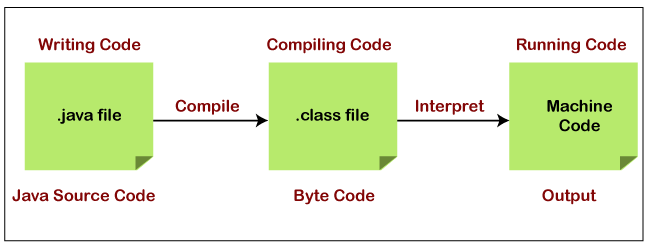
Java is a platform-independent programming language. It means that we can run Java on the platforms that have a **Java interpreter**. It is the reason that makes the Java platform-independent. The Java interpreter converts the Java bytecode (.class file) into the code understand by the operating system.

## What is an interpreter in Java?

**Java interpreter** is a computer program (system software) that implements the JVM. It is responsible for reading and executing the program. It is designed in such a way that it can read the source program and translate the source code instruction by instruction. **It converts the high-level program into assembly language** (machine language).

## How does the Java interpreter work?

To convert the byte code into machine code, we deploy the .class file on the [Java Virtual Machine (JVM)](https://www.javatpoint.com/jvm-java-virtual-machine). The JVM converts that code into machine code using the Java interpreter. The JVM uses the interpreter at runtime, after that it executes the code on the host machine.



As the Java compiler compiles the source code into the [Java bytecode](https://www.javatpoint.com/java-bytecode). In the same way, the Java interpreter converts or translates the bytecode into the machine-understandable format i.e. machine code, after that the machine code interacts with the operating system.

If the JVM is installed on any system it means that the platform is JVM enabled. The platform performs all the tasks of the [Java](https://www.javatpoint.com/java-tutorial) run-time system. It loads the Java class file and interprets the compiled byte-code.

The [browsers](https://www.javatpoint.com/browsers), like [Google Chrome](https://www.javatpoint.com/google-chrome), Netscape, etc. are the popular example that contains the Java interpreter. It means these are Java-enabled browsers. It is used to run the Applet in the browser. The interpreter also serves as a specialized compiler in an implementation that supports dynamic or **just-in-time (JIT)** compilation which turns the Java bytecode into native machine instructions.

First, we specify the class by using the **java** command followed by the class name and options available for the interpreter, and command-line arguments if required. We use the following command to load the class:

***% java [interpreter options] class name [arguments]***

In the above command, the class name should be a fully qualified name (the name of the class that includes the package name, if any). Remember that, we do not write the .class extension at the end of the class name. For example:

***java Product***

***java com.javatoint.product.Mobile***

In the first command, **Product** is the class name. In the second command, **com.javatpint.product** is the name of the package in which the **Mobile** class is stored.

Once the class is loaded, Java follows a convention and searches for the class that contains the main() method. When the JVM founds the main() method, the interpreter starts the application by invoking the main() method. After executing the main() method, additional threads, and references other classes.

## Features of Interpreter

It converts the source code into machine language, line by line at run time, without changing the sequence.

* An interpreter does not generate an intermediate machine code
* Each error of every line is displayed one by one
* When compared to a compiler, the program execution speed is slower
* Less amount of time is spent on analyzing and processing the program

## Difference between Interpreter and Compiler

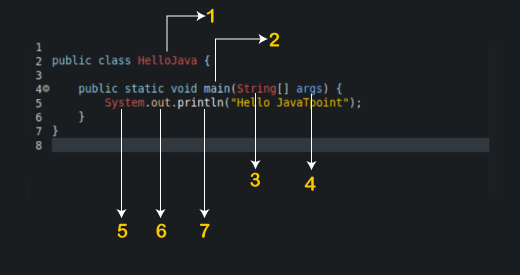
|  |  |
| --- | --- |
| **Interpreter** | **Compiler** |
| It translates the code instruction by instruction. | It translates the entire program at once. |
| Its execution is slower. | Its execution is faster. |
| Its compile time is less. | It takes more time to compile the code. |
| It does not generate the intermediate object code. | It generates the intermediate object code. |
| It compiles the program until an error is found. | All the errors show once at the end of the compilation. |
| Python, PHP, Ruby, and Perl use an interpreter. | Java, C++, Scala, and C uses a compiler. |

# Identifiers in Java

Identifiers in Java are symbolic names used for identification. They can be a class name, variable name, method name, package name, constant name, and more. However, In [Java](https://www.javatpoint.com/java-tutorial), There are some reserved words that cannot be used as an identifier.

For every identifier there are some conventions that should be used before declaring them. Let's understand it with a simple Java program:

1. **public** **class** HelloJava {
2. **public** **static** **void** main(String[] args) {
3. System.out.println("Hello JavaTpoint");
4. }
5. }



From the above example, we have the following Java identifiers:

1. HelloJava (Class name)
2. main (main method)
3. String (Predefined Class name)
4. args (String variables)
5. System (Predefined class)
6. out (Variable name)
7. println (method)

let's understand the rules for Java identifier:

### Rules for Identifiers in Java

There are some rules and conventions for declaring the identifiers in Java. If the identifiers are not properly declared, we may get a compile-time error. Following are some rules and conventions for declaring identifiers:

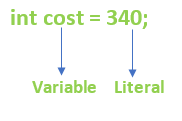
* A valid identifier must have characters [A-Z] or [a-z] or numbers [0-9], and underscore(\_) or a dollar sign ($). for example, @javatype is not a valid identifier because it contains a special character which is @.
* There should not be any space in an identifier. For example, **java type** is an invalid identifier.
* An identifier should not contain a number at the starting. For example, 123javat is an invalid identifier.
* An identifier should be of length 4-15 letters only. However, there is no limit on its length. But, it is good to follow the standard conventions.
* We can't use the Java reserved keywords as an identifier such as int, float, double, char, etc. For example, **int double** is an invalid identifier in Java.
* An identifier should not be any query language keywords such as SELECT, FROM, COUNT, DELETE, etc.

# Literals in Java

In [Java](https://www.javatpoint.com/java-tutorial), **literal** is a notation that represents a fixed value in the source code. In lexical analysis, literals of a given type are generally known as [**tokens**](https://www.javatpoint.com/java-tokens).

## Literals

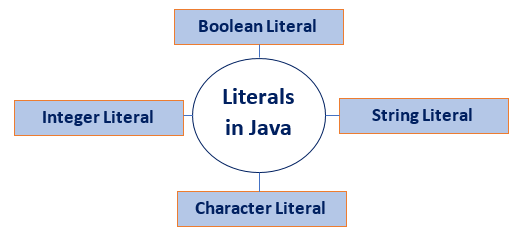
In Java, **literals** are the constant values that appear directly in the program. It can be assigned directly to a variable. Java has various types of literals. The following figure represents a literal.



## Types of Literals in Java

There are the majorly **four** types of literals in Java:

1. Integer Literal
2. Character Literal
3. Boolean Literal
4. String Literal



### Integer Literals

Integer literals are sequences of digits. There are three types of integer literals:

* **Decimal Integer:** These are the set of numbers that consist of digits from 0 to 9. It may have a positive (**+)** or negative (**-**) Note that between numbers commas and non-digit characters are not permitted. For example, **5678, +657, -89,** etc.

1. **int** decVal = 26;

* **Octal Integer:** It is a combination of number have digits from 0 to 7 with a leading 0. For example, **045, 026,**

AD

1. **int** octVal = 067;

* **Hexa-Decimal:** The sequence of digits preceded by **0x** or **0X** is considered as hexadecimal integers. It may also include a character from **a** to **f** or **A** to **F** that represents numbers from **10** to **15**, respectively. For example, **0xd, 0xf,**

1. **int** hexVal = 0x1a;

* **Binary Integer:** Base 2, whose digits consists of the numbers 0 and 1 (you can create binary literals in Java SE 7 and later). Prefix 0b represents the Binary system. For example, 0b11010.

1. **int** binVal = 0b11010;

### Real Literals

The numbers that contain fractional parts are known as real literals. We can also represent real literals in exponent form. For example, **879.90, 99E-3,** etc.

### Backslash Literals

Java supports some special backslash character literals known as backslash literals. They are used in formatted output. For example:

**\n:** It is used for a new line

**\t:** It is used for horizontal tab

**\b:** It is used for blank space

**\v:** It is used for vertical tab

**\a:** It is used for a small beep

**\r:** It is used for carriage return

**\':** It is used for a single quote

**\":** It is used for double quotes

### Character Literals

A character literal is expressed as a character or an escape sequence, enclosed in a **single** quote (**''**) mark. It is always a type of char. For example, **'a', '%', '\u000d',** etc.

### String Literals

String literal is a sequence of characters that is enclosed between **double** quotes ("") marks. It may be alphabet, numbers, special characters, blank space, etc. For example, "**Jack", "12345", "\n",** etc.

### Floating Point Literals

The vales that contain decimal are floating literals. In Java, float and double primitive types fall into floating-point literals. Keep in mind while dealing with floating-point literals.

* Floating-point literals for float type end with F or f. For example, **6f, 8.354F**, etc. It is a **32**-bit float literal.
* Floating-point literals for double type end with D or d. It is optional to write D or d. For example, **6d, 8.354D,** etc. It is a **64**-bit double literal.
* It can also be represented in the form of the **exponent**.

**Floating:**

* **float** length = 155.4f;

**Decimal:**

* **double** interest = 99658.445;

**Decimal in Exponent form:**

* **double** val= 1.234e2;

### Boolean Literals

Boolean literals are the value that is either true or false. It may also have values 0 and 1. For example, **true, 0,** etc.

1. **boolean** isEven = **true**;

### Null Literals

**Null** literal is often used in programs as a marker to indicate that reference type object is unavailable. The value **null** may be assigned to any variable, except variables of primitive types.

AD

1. String stuName = **null**;
2. Student age = **null**;

### Class Literals

**Class literal** formed by taking a type name and appending **.class** extension. For example, **Scanner.class**. It refers to the object (of type Class) that represents the type itself.

1. **class** classType = Scanner.**class**;

## Invalid Literals

There is some invalid declaration of literals.

1. **float** g = 6\_.674f;
2. **float** g = 6.\_674F;
3. **long** phoneNumber = 99\_00\_99\_00\_99\_L;
4. **int** x = 77\_;
5. **int** y = 0\_x76;
6. **int** z = 0X\_12;
7. **int** z = 0X12\_;

### Restrictions to Use Underscore (\_)

* It can be used at the beginning, at the end, and in-between of a number.
* It can be adjacent to a decimal point in a floating-point literal.
* Also, can be used prior to an F or L suffix.
* In positions where a string of digits is expected.

AD

## Why use literals?

To avoid defining the constant somewhere and making up a label for it. Instead, to write the value of a constant operand as a part of the instruction.

## How to use literals?

A literal in Java can be identified with the prefix **=** followed by a specific value.

**LiteralsExample.java**

**public** **class** LiteralsExample

{

**public** **static** **void** main(String args[])

{

**int** count = 987;

**float** floatVal = 4534.99f;

**double** cost = 19765.567;

**int** hexaVal = 0x7e4;

**int** binary = 0b11010;

**char** alpha = 'p';

String str = "Java";

**boolean** boolVal = **true**;

**int** octalVal = 067;

String stuName = **null**;

**char** ch1 = '\u0021';

**char** ch2 = 1456;

System.out.println(count);

System.out.println(floatVal);

System.out.println(cost);

System.out.println(hexaVal);

System.out.println(binary);

System.out.println(alpha);

System.out.println(str);

System.out.println(boolVal);

System.out.println(octalVal);

System.out.println(stuName);

System.out.println(ch1);

System.out.println("\t" +"backslash literal");

System.out.println(ch2);

}

}

**Output:**

987

4534.99

19765.567

2020

26

p

Java

true

55

null

!

backslash literal

?

# 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

AD

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

**public** **class** OperatorExample{

**public** **static** **void** main(String args[]){

**int** x=10;

System.out.println(x++);//10 (11)

System.out.println(++x);//12

System.out.println(x--);//12 (11)

System.out.println(--x);//10

}}

**Output:**

10

12

12

10

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

**public** **class** OperatorExample{

**public** **static** **void** main(String args[]){

**int** a=10;

**int** b=10;

System.out.println(a++ + ++a);//10+12=22

System.out.println(b++ + b++);//10+11=21

}}

**Output:**

22

21

### Java Unary Operator Example: ~ and !

**public** **class** OperatorExample{

**public** **static** **void** main(String args[]){

**int** a=10;

**int** b=-10;

**boolean** c=**true**;

**boolean** d=**false**;

System.out.println(~a);//11 (minus of total positive value which starts from 0)

System.out.println(~b);//9 (positive of total minus, positive starts from 0)

System.out.println(!c);//false (opposite of boolean value)

System.out.println(!d);//true

}}

**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

**public** **class** OperatorExample{

**public** **static** **void** main(String args[]){

**int** a=10;

**int** b=5;

System.out.println(a+b);//15

System.out.println(a-b);//5

System.out.println(a\*b);//50

System.out.println(a/b);//2

System.out.println(a%b);//0

}}

**Output:**

15

5

50

2

0

### Java Arithmetic Operator Example: Expression

**public** **class** OperatorExample{

**public** **static** **void** main(String args[]){

System.out.println(10\*10/5+3-1\*4/2);

}}

**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

**public** **class** OperatorExample{

**public** **static** **void** main(String args[]){

System.out.println(10<<2);//10\*2^2=10\*4=40

System.out.println(10<<3);//10\*2^3=10\*8=80

System.out.println(20<<2);//20\*2^2=20\*4=80

System.out.println(15<<4);//15\*2^4=15\*16=240

}}

**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

**public** OperatorExample{

**public** **static** **void** main(String args[]){

System.out.println(10>>2);//10/2^2=10/4=2

System.out.println(20>>2);//20/2^2=20/4=5

System.out.println(20>>3);//20/2^3=20/8=2

}}

**Output:**

2

5

2

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

**public** **class** OperatorExample{

**public** **static** **void** main(String args[]){

    //For positive number, >> and >>> works same

    System.out.println(20>>2);

    System.out.println(20>>>2);

    //For negative number, >>> changes parity bit (MSB) to 0

    System.out.println(-20>>2);

    System.out.println(-20>>>2);

}}

**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.

**public** **class** OperatorExample{

**public** **static** **void** main(String args[]){

**int** a=10;

**int** b=5;

**int** c=20;

System.out.println(a<b&&a<c);//false && true = false

System.out.println(a<b&a<c);//false & true = false

}}

**Output:**

false

false

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

**public** **class** OperatorExample{

**public** **static** **void** main(String args[]){

**int** a=10;

**int** b=5;

**int** c=20;

System.out.println(a<b&&a++<c);//false && true = false

System.out.println(a);//10 because second condition is not checked

System.out.println(a<b&a++<c);//false && true = false

System.out.println(a);//11 because second condition is checked

}}

**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.

**public** **class** OperatorExample{

**public** **static** **void** main(String args[]){

**int** a=10;

**int** b=5;

**int** c=20;

System.out.println(a>b||a<c);//true || true = true

System.out.println(a>b|a<c);//true | true = true

//|| vs |

System.out.println(a>b||a++<c);//true || true = true

System.out.println(a);//10 because second condition is not checked

System.out.println(a>b|a++<c);//true | true = true

System.out.println(a);//11 because second condition is checked

}}

**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

**public** **class** OperatorExample{

**public** **static** **void** main(String args[]){

**int** a=2;

**int** b=5;

**int** min=(a<b)?a:b;

System.out.println(min);

}}

**Output:**

2

Another Example:

**public** **class** OperatorExample{

**public** **static** **void** main(String args[]){

**int** a=10;

**int** b=5;

**int** min=(a<b)?a:b;

System.out.println(min);

}}

**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

**public** **class** OperatorExample{

**public** **static** **void** main(String args[]){

**int** a=10;

**int** b=20;

a+=4;//a=a+4 (a=10+4)

b-=4;//b=b-4 (b=20-4)

System.out.println(a);

System.out.println(b);

}}

**Output:**

14

16

### Java Assignment Operator Example

**public** **class** OperatorExample{

**public** **static** **void** main(String[] args){

**int** a=10;

a+=3;//10+3

System.out.println(a);

a-=4;//13-4

System.out.println(a);

a\*=2;//9\*2

System.out.println(a);

a/=2;//18/2

System.out.println(a);

}}

**Output:**

13

9

18

9

### Java Assignment Operator Example: Adding short

**public** **class** OperatorExample{

**public** **static** **void** main(String args[]){

**short** a=10;

**short** b=10;

//a+=b;//a=a+b internally so fine

a=a+b;//Compile time error because 10+10=20 now int

System.out.println(a);

}}

**Output:**

Compile time error

After type cast:

**public** **class** OperatorExample{

**public** **static** **void** main(String args[]){

**short** a=10;

**short** b=10;

a=(**short**)(a+b);//20 which is int now converted to short

System.out.println(a);

}}

**Output:**

20

# Java Keywords

Java keywords are also known as reserved words. Keywords are particular words that act as a key to a code. These are predefined words by Java so they cannot be used as a variable or object name or class name.

## List of Java Keywords

A list of Java keywords or reserved words are given below:

1. [**abstract**](https://www.javatpoint.com/abstract-keyword-in-java)**:** Java abstract keyword is used to declare an abstract class. An abstract class can provide the implementation of the interface. It can have abstract and non-abstract methods.
2. [**boolean:**](https://www.javatpoint.com/boolean-keyword-in-java) Java boolean keyword is used to declare a variable as a boolean type. It can hold True and False values only.
3. [**break**](https://www.javatpoint.com/java-break)**:** Java break keyword is used to break the loop or switch statement. It breaks the current flow of the program at specified conditions.
4. [**byte**](https://www.javatpoint.com/byte-keyword-in-java)**:** Java byte keyword is used to declare a variable that can hold 8-bit data values.
5. [**case**](https://www.javatpoint.com/case-keyword-in-java)**:** Java case keyword is used with the switch statements to mark blocks of text.
6. [**catch**](https://www.javatpoint.com/try-catch-block)**:** Java catch keyword is used to catch the exceptions generated by try statements. It must be used after the try block only.
7. [**char**](https://www.javatpoint.com/char-keyword-in-java)**:** Java char keyword is used to declare a variable that can hold unsigned 16-bit Unicode characters
8. [**class**](https://www.javatpoint.com/class-keyword-in-java)**:** Java class keyword is used to declare a class.
9. [**continue**](https://www.javatpoint.com/java-continue)**:** Java continue keyword is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition.
10. [**default**](https://www.javatpoint.com/default-keyword-in-java)**:** Java default keyword is used to specify the default block of code in a switch statement.
11. [**do**](https://www.javatpoint.com/java-do-while-loop)**:** Java do keyword is used in the control statement to declare a loop. It can iterate a part of the program several times.
12. [**double**](https://www.javatpoint.com/double-keyword-in-java)**:** Java double keyword is used to declare a variable that can hold 64-bit floating-point number.
13. [**else**](https://www.javatpoint.com/java-if-else)**:** Java else keyword is used to indicate the alternative branches in an if statement.
14. [**enum**](https://www.javatpoint.com/enum-in-java)**:** Java enum keyword is used to define a fixed set of constants. Enum constructors are always private or default.
15. [**extends**](https://www.javatpoint.com/inheritance-in-java)**:** Java extends keyword is used to indicate that a class is derived from another class or interface.
16. [**final**](https://www.javatpoint.com/final-keyword)**:** Java final keyword is used to indicate that a variable holds a constant value. It is used with a variable. It is used to restrict the user from updating the value of the variable.
17. [**finally**](https://www.javatpoint.com/finally-block-in-exception-handling)**:** Java finally keyword indicates a block of code in a try-catch structure. This block is always executed whether an exception is handled or not.
18. [**float**](https://www.javatpoint.com/float-keyword-in-java)**:** Java float keyword is used to declare a variable that can hold a 32-bit floating-point number.
19. [**for**](https://www.javatpoint.com/java-for-loop)**:** Java for keyword is used to start a for loop. It is used to execute a set of instructions/functions repeatedly when some condition becomes true. If the number of iteration is fixed, it is recommended to use for loop.
20. [**if**](https://www.javatpoint.com/java-if-else)**:** Java if keyword tests the condition. It executes the if block if the condition is true.
21. [**implements**](https://www.javatpoint.com/interface-in-java)**:** Java implements keyword is used to implement an interface.
22. [**import**](https://www.javatpoint.com/package)**:** Java import keyword makes classes and interfaces available and accessible to the current source code.
23. [**instanceof**](https://www.javatpoint.com/downcasting-with-instanceof-operator)**:** Java instanceof keyword is used to test whether the object is an instance of the specified class or implements an interface.
24. [**int**](https://www.javatpoint.com/int-keyword-in-java)**:** Java int keyword is used to declare a variable that can hold a 32-bit signed integer.
25. [**interface**](https://www.javatpoint.com/interface-in-java)**:** Java interface keyword is used to declare an interface. It can have only abstract methods.
26. [**long**](https://www.javatpoint.com/long-keyword-in-java)**:** Java long keyword is used to declare a variable that can hold a 64-bit integer.
27. **native:** Java native keyword is used to specify that a method is implemented in native code using JNI (Java Native Interface).
28. [**new**](https://www.javatpoint.com/new-keyword-in-java)**:** Java new keyword is used to create new objects.
29. [**null**](https://www.javatpoint.com/null-keyword-in-java)**:** Java null keyword is used to indicate that a reference does not refer to anything. It removes the garbage value.
30. [**package**](https://www.javatpoint.com/package)**:** Java package keyword is used to declare a Java package that includes the classes.
31. [**private**](https://www.javatpoint.com/private-keyword-in-java)**:** Java private keyword is an access modifier. It is used to indicate that a method or variable may be accessed only in the class in which it is declared.
32. [**protected**](https://www.javatpoint.com/protected-keyword-in-java)**:** Java protected keyword is an access modifier. It can be accessible within the package and outside the package but through inheritance only. It can't be applied with the class.
33. [**public**](https://www.javatpoint.com/public-keyword-in-java)**:** Java public keyword is an access modifier. It is used to indicate that an item is accessible anywhere. It has the widest scope among all other modifiers.
34. [**return**](https://www.javatpoint.com/return-keyword-in-java)**:** Java return keyword is used to return from a method when its execution is complete.
35. [**short**](https://www.javatpoint.com/short-keyword-in-java)**:** Java short keyword is used to declare a variable that can hold a 16-bit integer.
36. [**static**](https://www.javatpoint.com/static-keyword-in-java)**:** Java static keyword is used to indicate that a variable or method is a class method. The static keyword in Java is mainly used for memory management.
37. [**strictfp**](https://www.javatpoint.com/strictfp-keyword)**:** Java strictfp is used to restrict the floating-point calculations to ensure portability.
38. [**super**](https://www.javatpoint.com/super-keyword)**:** Java super keyword is a reference variable that is used to refer to parent class objects. It can be used to invoke the immediate parent class method.
39. [**switch**](https://www.javatpoint.com/java-switch)**:** The Java switch keyword contains a switch statement that executes code based on test value. The switch statement tests the equality of a variable against multiple values.
40. [**synchronized**](https://www.javatpoint.com/synchronization-in-java)**:** Java synchronized keyword is used to specify the critical sections or methods in multithreaded code.
41. [**this**](https://www.javatpoint.com/this-keyword)**:** Java this keyword can be used to refer the current object in a method or constructor.
42. [**throw**](https://www.javatpoint.com/throw-keyword)**:** The Java throw keyword is used to explicitly throw an exception. The throw keyword is mainly used to throw custom exceptions. It is followed by an instance.
43. [**throws**](https://www.javatpoint.com/throws-keyword-and-difference-between-throw-and-throws)**:** The Java throws keyword is used to declare an exception. Checked exceptions can be propagated with throws.
44. [**transient**](https://www.javatpoint.com/transient-keyword)**:** Java transient keyword is used in serialization. If you define any data member as transient, it will not be serialized.
45. [**try**](https://www.javatpoint.com/try-catch-block)**:** Java try keyword is used to start a block of code that will be tested for exceptions. The try block must be followed by either catch or finally block.
46. **void:** Java void keyword is used to specify that a method does not have a return value.
47. [**volatile**](https://www.javatpoint.com/volatile-keyword-in-java)**:** Java volatile keyword is used to indicate that a variable may change asynchronously.
48. [**while**](https://www.javatpoint.com/java-while-loop)**:** Java while keyword is used to start a while loop. This loop iterates a part of the program several times. If the number of iteration is not fixed, it is recommended to use the while loop.

# Data Types in Java

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include [Classes](https://www.javatpoint.com/object-and-class-in-java), [Interfaces](https://www.javatpoint.com/interface-in-java), and [Arrays](https://www.javatpoint.com/array-in-java).

## Java Primitive Data Types

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in [Java language](https://www.javatpoint.com/java-tutorial).

Java is a statically-typed programming language. It means, all [variables](https://www.javatpoint.com/java-variables) must be declared before its use. That is why we need to declare variable's type and name.

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



|  |  |  |
| --- | --- | --- |
| **Data Type** | **Default Value** | **Default size** |
| Boolean | false | 1 bit |
| Char | '\u0000' | 2 byte |
| Byte | 0 | 1 byte |
| Short | 0 | 2 byte |
| Int | 0 | 4 byte |
| Long | 0L | 8 byte |
| Float | 0.0f | 4 byte |
| Double | 0.0d | 8 byte |

## Boolean Data Type

The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions.

The Boolean data type specifies one bit of information, but its "size" can't be defined precisely.

**Example:**

1. Boolean one = **false**

## Byte Data Type

The byte data type is an example of primitive data type. It isan 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0.

The byte data type is used to save memory in large arrays where the memory savings is most required. It saves space because a byte is 4 times smaller than an integer. It can also be used in place of "int" data type.

**Example:**

1. **byte** a = 10, **byte** b = -20

## Short Data Type

The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768 and maximum value is 32,767. Its default value is 0.

The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer.

**Example:**

1. **short** s = 10000, **short** r = -5000

## Int Data Type

The int data type is a 32-bit signed two's complement integer. Its value-range lies between - 2,147,483,648 (-2^31) to 2,147,483,647 (2^31 -1) (inclusive). Its minimum value is - 2,147,483,648and maximum value is 2,147,483,647. Its default value is 0.

The int data type is generally used as a default data type for integral values unless if there is no problem about memory.

**Example:**

1. **int** a = 100000, **int** b = -200000

## Long Data Type

The long data type is a 64-bit two's complement integer. Its value-range lies between -9,223,372,036,854,775,808(-2^63) to 9,223,372,036,854,775,807(2^63 -1)(inclusive). Its minimum value is - 9,223,372,036,854,775,808and maximum value is 9,223,372,036,854,775,807. Its default value is 0. The long data type is used when you need a range of values more than those provided by int.

**Example:**

1. **long** a = 100000L, **long** b = -200000L

## Float Data Type

The float data type is a single-precision 32-bit IEEE 754 floating point.Its value range is unlimited. It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers. The float data type should never be used for precise values, such as currency. Its default value is 0.0F.

**Example:**

1. **float** f1 = 234.5f

## Double Data Type

The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency. Its default value is 0.0d.

**Example:**

1. **double** d1 = 12.3

## Char Data Type

The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive).The char data type is used to store characters.

**Example:**

1. **char** letterA = 'A'

# Java If-else Statement

The [Java](https://www.javatpoint.com/java-tutorial) if statement is used to test the condition. It checks [boolean](https://www.javatpoint.com/boolean-keyword-in-java) condition: true or false. There are various types of if statement in Java.

* if statement
* if-else statement
* if-else-if ladder
* nested if statement

## Java if Statement

The Java if statement tests the condition. It executes the if block if condition is true.

**Syntax:**

**if**(condition){

//code to be executed

}



**Example:**

//Java Program to demonstate the use of if statement.

**public** **class** IfExample {

**public** **static** **void** main(String[] args) {

    //defining an 'age' variable

**int** age=20;

    //checking the age

**if**(age>18){

        System.out.print("Age is greater than 18");

    }

}

}

Output:

Age is greater than 18

## Java if-else Statement

The Java if-else statement also tests the condition. It executes the if block if condition is true otherwise else block is executed.

**Syntax:**

**if**(condition){

//code if condition is true

}**else**{

//code if condition is false

}



**Example:**

//A Java Program to demonstrate the use of if-else statement.

//It is a program of odd and even number.

**public** **class** IfElseExample {

**public** **static** **void** main(String[] args) {

    //defining a variable

**int** number=13;

    //Check if the number is divisible by 2 or not

**if**(number%2==0){

        System.out.println("even number");

    }**else**{

        System.out.println("odd number");

    }

}

}

Output:

odd number

**Leap Year Example:**

A year is leap, if it is divisible by 4 and 400. But, not by 100.

**public** **class** LeapYearExample {

**public** **static** **void** main(String[] args) {

**int** year=2020;

**if**(((year % 4 ==0) && (year % 100 !=0)) || (year % 400==0)){

        System.out.println("LEAP YEAR");

    }

**else**{

        System.out.println("COMMON YEAR");

    }

}

}

Output:

LEAP YEAR

## Using Ternary Operator

We can also use ternary operator (? :) to perform the task of if...else statement. It is a shorthand way to check the condition. If the condition is true, the result of ? is returned. But, if the condition is false, the result of : is returned.

**Example:**

**public** **class** IfElseTernaryExample {

**public** **static** **void** main(String[] args) {

**int** number=13;

    //Using ternary operator

    String output=(number%2==0)?"even number":"odd number";

    System.out.println(output);

}

}

Output:

odd number

# Switch Statement

The Java switch statement executes one statement from multiple conditions. It is like [if-else-if](https://www.javatpoint.com/java-if-else) ladder statement. The switch statement works with byte, short, int, long, enum types, String and some wrapper types like Byte, Short, Int, and Long. Since Java 7, you can use [strings](https://www.javatpoint.com/java-string) in the switch statement.

In other words, the switch statement tests the equality of a variable against multiple values.

### Points to Remember

* There can be one or N number of case values for a switch expression.
* The case value must be of switch expression type only. The case value must be literal or constant. It doesn't allow [variables](https://www.javatpoint.com/java-variables).
* The case values must be unique. In case of duplicate value, it renders compile-time error.
* The Java switch expression must be of byte, short, int, long (with its Wrapper type), *[enums](https://www.javatpoint.com/java-switch)* and string.
* Each case statement can have a break statement which is optional. When control reaches to the [break statement](https://www.javatpoint.com/java-break), it jumps the control after the switch expression. If a break statement is not found, it executes the next case.
* The case value can have a default label which is optional.

**Syntax:**

**switch**(expression){

**case** value1:

 //code to be executed;

**break**;  //optional

**case** value2:

 //code to be executed;

**break**;  //optional

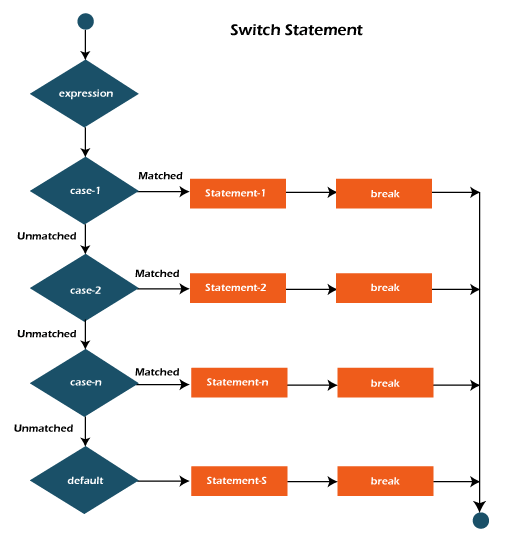
......

**default**:

code to be executed **if** all cases are not matched;

}

**Flowchart of Switch Statement**



**Example:**

**SwitchExample.java**

**public** **class** SwitchExample {

**public** **static** **void** main(String[] args) {

    //Declaring a variable for switch expression

**int** number=20;

    //Switch expression

**switch**(number){

    //Case statements

**case** 10: System.out.println("10");

**break**;

**case** 20: System.out.println("20");

**break**;

**case** 30: System.out.println("30");

**break**;

    //Default case statement

**default**:System.out.println("Not in 10, 20 or 30");

    }

}

}

**Output:**

20

**Finding Month Example:**

**SwitchMonthExample.javaHTML**

//Java Program to demonstrate the example of Switch statement

//where we are printing month name for the given number

**public** **class** SwitchMonthExample {

**public** **static** **void** main(String[] args) {

    //Specifying month number

**int** month=7;

    String monthString="";

    //Switch statement

**switch**(month){

    //case statements within the switch block

**case** 1: monthString="1 - January";

**break**;

**case** 2: monthString="2 - February";

**break**;

**case** 3: monthString="3 - March";

**break**;

**case** 4: monthString="4 - April";

**break**;

**case** 5: monthString="5 - May";

**break**;

**case** 6: monthString="6 - June";

**break**;

**case** 7: monthString="7 - July";

**break**;

**case** 8: monthString="8 - August";

**break**;

**case** 9: monthString="9 - September";

**break**;

**case** 10: monthString="10 - October";

**break**;

**case** 11: monthString="11 - November";

**break**;

**case** 12: monthString="12 - December";

**break**;

**default**:System.out.println("Invalid Month!");

    }

    //Printing month of the given number

    System.out.println(monthString);

}

}

**Output:**

7 - July

**Program to check Vowel or Consonant:**

If the character is A, E, I, O, or U, it is vowel otherwise consonant. It is not case-sensitive.

**SwitchVowelExample.java**

**public** **class** SwitchVowelExample {

**public** **static** **void** main(String[] args) {

**char** ch='O';

**switch**(ch)

    {

**case** 'a':

            System.out.println("Vowel");

**break**;

**case** 'e':

            System.out.println("Vowel");

**break**;

**case** 'i':

            System.out.println("Vowel");

**break**;

**case** 'o':

            System.out.println("Vowel");

**break**;

**case** 'u':

            System.out.println("Vowel");

**break**;

**case** 'A':

            System.out.println("Vowel");

**break**;

**case** 'E':

            System.out.println("Vowel");

**break**;

**case** 'I':

            System.out.println("Vowel");

**break**;

**case** 'O':

            System.out.println("Vowel");

**break**;

**case** 'U':

            System.out.println("Vowel");

**break**;

**default**:

            System.out.println("Consonant");

    }

}

}

**Output:**

Vowel

## Java Switch Statement is fall-through

The Java switch statement is fall-through. It means it executes all statements after the first match if a break statement is not present.

**Example:**

**SwitchExample2.java**

//Java Switch Example where we are omitting the

//break statement

**public** **class** SwitchExample2 {

**public** **static** **void** main(String[] args) {

**int** number=20;

    //switch expression with int value

**switch**(number){

    //switch cases without break statements

**case** 10: System.out.println("10");

**case** 20: System.out.println("20");

**case** 30: System.out.println("30");

**default**:System.out.println("Not in 10, 20 or 30");

    }

}

}

**Output:**

20

30

Not in 10, 20 or 30

## Java Switch Statement with String

Java allows us to use strings in switch expression since Java SE 7. The case statement should be string literal.

**Example:**

**SwitchStringExample.java**

//Java Program to demonstrate the use of Java Switch

//statement with String

**public** **class** SwitchStringExample {

**public** **static** **void** main(String[] args) {

    //Declaring String variable

    String levelString="Expert";

**int** level=0;

    //Using String in Switch expression

**switch**(levelString){

    //Using String Literal in Switch case

**case** "Beginner": level=1;

**break**;

**case** "Intermediate": level=2;

**break**;

**case** "Expert": level=3;

**break**;

**default**: level=0;

**break**;

    }

    System.out.println("Your Level is: "+level);

}

}

**Output:**

Your Level is: 3

## Java Nested Switch Statement

We can use switch statement inside other switch statement in Java. It is known as nested switch statement.

**Example:**

**NestedSwitchExample.java**

//Java Program to demonstrate the use of Java Nested Switch

**public** **class** NestedSwitchExample {

**public** **static** **void** main(String args[])

      {

      //C - CSE, E - ECE, M - Mechanical

**char** branch = 'C';

**int** collegeYear = 4;

**switch**( collegeYear )

        {

**case** 1:

                System.out.println("English, Maths, Science");

**break**;

**case** 2:

**switch**( branch )

                {

**case** 'C':

                        System.out.println("Operating System, Java, Data Structure");

**break**;

**case** 'E':

                        System.out.println("Micro processors, Logic switching theory");

**break**;

**case** 'M':

                        System.out.println("Drawing, Manufacturing Machines");

**break**;

                }

**break**;

**case** 3:

**switch**( branch )

                {

**case** 'C':

                        System.out.println("Computer Organization, MultiMedia");

**break**;

**case** 'E':

                        System.out.println("Fundamentals of Logic Design, Microelectronics");

**break**;

**case** 'M':

                        System.out.println("Internal Combustion Engines, Mechanical Vibration");

**break**;

                }

**break**;

**case** 4:

**switch**( branch )

                {

**case** 'C':

                        System.out.println("Data Communication and Networks, MultiMedia");

**break**;

**case** 'E':

                        System.out.println("Embedded System, Image Processing");

**break**;

**case** 'M':

                        System.out.println("Production Technology, Thermal Engineering");

**break**;

                }

**break**;

        }

    }

}

**Output:**

Data Communication and Networks, MultiMedia

## Java Enum in Switch Statement

Java allows us to use enum in switch statement. Java enum is a class that represent the group of constants. (immutable such as final variables). We use the keyword enum and put the constants in curly braces separated by comma.

**Example:**

**JavaSwitchEnumExample.java**

//Java Program to demonstrate the use of Enum

//in switch statement

**public** **class** JavaSwitchEnumExample {

**public** **enum** Day {  Sun, Mon, Tue, Wed, Thu, Fri, Sat  }

**public** **static** **void** main(String args[])

       {

         Day[] DayNow = Day.values();

**for** (Day Now : DayNow)

           {

**switch** (Now)

                {

**case** Sun:

                        System.out.println("Sunday");

**break**;

**case** Mon:

                        System.out.println("Monday");

**break**;

**case** Tue:

                        System.out.println("Tuesday");

**break**;

**case** Wed:

                        System.out.println("Wednesday");

**break**;

**case** Thu:

                        System.out.println("Thursday");

**break**;

**case** Fri:

                        System.out.println("Friday");

**break**;

**case** Sat:

                        System.out.println("Saturday");

**break**;

                }

            }

        }

}

**Output:**

Sunday

Monday

Twesday

Wednesday

Thursday

Friday

Saturday

## Java Wrapper in Switch Statement

Java allows us to use four [wrapper classes](https://www.javatpoint.com/wrapper-class-in-java): Byte, Short, Integer and Long in switch statement.

**Example:**

**WrapperInSwitchCaseExample.java**

//Java Program to demonstrate the use of Wrapper class

//in switch statement

**public** **class** WrapperInSwitchCaseExample {

**public** **static** **void** main(String args[])

       {

            Integer age = 18;

**switch** (age)

            {

**case** (16):

                    System.out.println("You are under 18.");

**break**;

**case** (18):

                    System.out.println("You are eligible for vote.");

**break**;

**case** (65):

                    System.out.println("You are senior citizen.");

**break**;

**default**:

                    System.out.println("Please give the valid age.");

**break**;

            }

        }

}

**Output:**

You are eligible for vote.

# 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:**

**while** (condition){

//code to be executed

Increment / decrement statement

}

**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

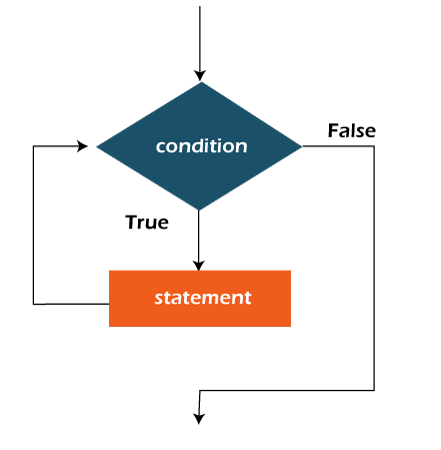
**2. Update expression:** Every time the loop body is executed, this expression increments or decrements loop variable.

**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**

**public** **class** WhileExample {

**public** **static** **void** main(String[] args) {

**int** i=1;

**while**(i<=10){

        System.out.println(i);

    i++;

    }

}

}

**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:**

**while**(**true**){

//code to be executed

}

**Example:**

**WhileExample2.java**

**public** **class** WhileExample2 {

**public** **static** **void** main(String[] args) {

 // setting the infinite while loop by passing true to the condition

**while**(**true**){

        System.out.println("infinitive while loop");

    }

}

}

**Output:**

infinitive while loop

infinitive while loop

infinitive while loop

infinitive while loop

infinitive while loop

ctrl+c

# Java 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:**

**do**{

//code to be executed / loop body

//update statement

}**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**

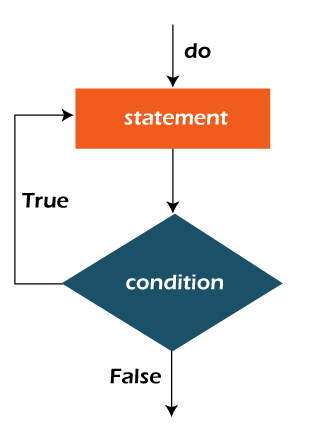
**2. Update expression:** Every time the loop body is executed, the this expression increments or decrements loop variable.

**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**

**public** **class** DoWhileExample {

**public** **static** **void** main(String[] args) {

**int** i=1;

**do**{

        System.out.println(i);

    i++;

    }**while**(i<=10);

}

}

**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:**

**do**{

//code to be executed

}**while**(**true**);

**Example:**

**DoWhileExample2.java**

**public** **class** DoWhileExample2 {

**public** **static** **void** main(String[] args) {

**do**{

        System.out.println("infinitive do while loop");

    }**while**(**true**);

}

}

**Output:**

infinitive do while loop

infinitive do while loop

infinitive do while loop

ctrl+c

# Loops in Java

The Java for loop is used to iterate a part of the program several times. If the number of iteration is **fixed**, it is recommended to use for loop.

There are three types of for loops in Java.



* 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:**

**for**(initialization; condition; increment/decrement){

//statement or code to be executed

}

**Flowchart:**



**Example:**

**ForExample.java**

//Java Program to demonstrate the example of for loop

//which prints table of 1

**public** **class** ForExample {

**public** **static** **void** main(String[] args) {

    //Code of Java for loop

**for**(**int** i=1;i<=10;i++){

        System.out.println(i);

    }

}

}

**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**

**public** **class** NestedForExample {

**public** **static** **void** main(String[] args) {

//loop of i

**for**(**int** i=1;i<=3;i++){

//loop of j

**for**(**int** j=1;j<=3;j++){

        System.out.println(i+" "+j);

}//end of i

}//end of j

}

}

**Output:**

1 1

1 2

1 3

2 1

2 2

2 3

3 1

3 2

3 3

**Pyramid Example 1:**

**PyramidExample.java**

**public** **class** PyramidExample {

**public** **static** **void** main(String[] args) {

**for**(**int** i=1;i<=5;i++){

**for**(**int** j=1;j<=i;j++){

        System.out.print("\* ");

}

System.out.println();//new line

}

}

}

**Output:**

\*

\* \*

\* \* \*

\* \* \* \*

\* \* \* \* \*

**Pyramid Example 2:**

**PyramidExample2.java**

**public** **class** PyramidExample2 {

**public** **static** **void** main(String[] args) {

**int** term=6;

**for**(**int** i=1;i<=term;i++){

**for**(**int** j=term;j>=i;j--){

        System.out.print("\* ");

}

System.out.println();//new line

}

}

}

**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:**

**for**(data\_type variable : array\_name){

//code to be executed

}

**Example:**

**ForEachExample.java**

//Java For-each loop example which prints the

//elements of the array

**public** **class** ForEachExample {

**public** **static** **void** main(String[] args) {

    //Declaring an array

**int** arr[]={12,23,44,56,78};

    //Printing array using for-each loop

**for**(**int** i:arr){

        System.out.println(i);

    }

}

}

**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:**

labelname:

**for**(initialization; condition; increment/decrement){

//code to be executed

}

**Example:**

**LabeledForExample.java**

//A Java program to demonstrate the use of labeled for loop

**public** **class** LabeledForExample {

**public** **static** **void** main(String[] args) {

    //Using Label for outer and for loop

    aa:

**for**(**int** i=1;i<=3;i++){

            bb:

**for**(**int** j=1;j<=3;j++){

**if**(i==2&&j==2){

**break** aa;

                    }

                    System.out.println(i+" "+j);

                }

        }

}

}

**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**

**public** **class** LabeledForExample2 {

**public** **static** **void** main(String[] args) {

    aa:

**for**(**int** i=1;i<=3;i++){

            bb:

**for**(**int** j=1;j<=3;j++){

**if**(i==2&&j==2){

**break** bb;

                    }

                    System.out.println(i+" "+j);

                }

        }

}

}

**Output:**

1 1

1 2

1 3

2 1

3 1

3 2

3 3

## Java Infinitive for Loop

If you use two semicolons ;; in the for loop, it will be infinitive for loop.

**Syntax:**

**for**(;;){

//code to be executed

}

**Example:**

**ForExample.java**

//Java program to demonstrate the use of infinite for loop

//which prints an statement

**public** **class** ForExample {

**public** **static** **void** main(String[] args) {

    //Using no condition in for loop

**for**(;;){

        System.out.println("infinitive loop");

    }

}

}

**Output:**

infinitive loop

infinitive loop

infinitive loop

infinitive loop

infinitive loop

ctrl+c

Now, you need to press ctrl+c to exit from the program.

## Java for Loop vs while Loop vs do-while Loop

|  |  |  |  |
| --- | --- | --- | --- |
| **Comparison** | **for loop** | **while loop** | **do-while loop** |
| Introduction | The Java for loop is a control flow statement that iterates a part of the [programs](https://www.javatpoint.com/java-programs) multiple times. | The Java while loop is a control flow statement that executes a part of the programs repeatedly on the basis of given boolean condition. | The Java do while loop is a control flow statement that executes a part of the programs at least once and the further execution depends upon the given boolean condition. |
| When to use | If the number of iteration is fixed, it is recommended to use for loop. | If the number of iteration is not fixed, it is recommended to use while loop. | If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use the do-while loop. |
| Syntax | for(init;condition;incr/decr){ // code to be executed } | while(condition){ //code to be executed } | do{ //code to be executed }while(condition); |
| Example | //for loop for(int i=1;i<=10;i++){ System.out.println(i); } | //while loop int i=1; while(i<=10){ System.out.println(i); i++; } | //do-while loop int i=1; do{ System.out.println(i); i++; }while(i<=10); |
| Syntax for infinitive loop | for(;;){ //code to be executed } | while(true){ //code to be executed } | do{ //code to be executed }while(true); |

# Type Casting in Java

In Java, **type casting** is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer. In this section, we will discuss **type casting** and **its types** with proper examples.



## Type casting

Convert a value from one data type to another data type is known as **type casting**.

## Types of Type Casting

There are two types of type casting:

* Widening Type Casting
* Narrowing Type Casting

### Widening Type Casting

Converting a lower data type into a higher one is called **widening** type casting. It is also known as **implicit conversion** or **casting down**. It is done automatically. It is safe because there is no chance to lose data. It takes place when:

* Both data types must be compatible with each other.
* The target type must be larger than the source type.

1. **byte** -> **short** -> **char** -> **int** -> **long** -> **float** -> **double**

For example, the conversion between numeric data type to char or Boolean is not done automatically. Also, the char and Boolean data types are not compatible with each other. Let's see an example.

**WideningTypeCastingExample.java**

**public** **class** WideningTypeCastingExample

{

**public** **static** **void** main(String[] args)

{

**int** x = 7;

//automatically converts the integer type into long type

**long** y = x;

//automatically converts the long type into float type

**float** z = y;

System.out.println("Before conversion, int value "+x);

System.out.println("After conversion, long value "+y);

System.out.println("After conversion, float value "+z);

}

}

**Output**

Before conversion, the value is: 7

After conversion, the long value is: 7

After conversion, the float value is: 7.0

In the above example, we have taken a variable x and converted it into a long type. After that, the long type is converted into the float type.

### Narrowing Type Casting

Converting a higher data type into a lower one is called **narrowing** type casting. It is also known as **explicit conversion** or **casting up**. It is done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

1. **double** -> **float** -> **long** -> **int** -> **char** -> **short** -> **byte**

Let's see an example of narrowing type casting.

In the following example, we have performed the narrowing type casting two times. First, we have converted the double type into long data type after that long data type is converted into int type.

**NarrowingTypeCastingExample.java**

**public** **class** NarrowingTypeCastingExample

{

**public** **static** **void** main(String args[])

{

**double** d = 166.66;

//converting double data type into long data type

**long** l = (**long**)d;

//converting long data type into int data type

**int** i = (**int**)l;

System.out.println("Before conversion: "+d);

//fractional part lost

System.out.println("After conversion into long type: "+l);

//fractional part lost

System.out.println("After conversion into int type: "+i);

}

}

**Output**

Before conversion: 166.66

After conversion into long type: 166

After conversion into int type: 166