Java Tutorial

Top of Form

Bottom of Form

Java is a high-level programming language originally developed by Sun Microsystems and released in 1995. Java runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.

Why to Learn java Programming?

Java is a MUST for students and working professionals to become a great Software Engineer specially when they are working in Software Development Domain. I will list down some of the key advantages of learning Java Programming:

* **Object Oriented** − In Java, everything is an Object. Java can be easily extended since it is based on the Object model.
* **Platform Independent** − Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by the Virtual Machine (JVM) on whichever platform it is being run on.
* **Simple** − Java is designed to be easy to learn. If you understand the basic concept of OOP Java, it would be easy to master.
* **Secure** − With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.
* **Architecture-neutral** − Java compiler generates an architecture-neutral object file format, which makes the compiled code executable on many processors, with the presence of Java runtime system.
* **Portable** − Being architecture-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C with a clean portability boundary, which is a POSIX subset.
* **Robust** − **Java** is **robust** because: It uses strong memory management. ... There is automatic garbage collection in **java** which runs on the **Java** Virtual Machine to get rid of objects which are not being used by a **Java** application anymore. There are exception handling and the type checking mechanism in **Java**.

Hello World using Java Programming.

Just to give you a little excitement about Java programming, I'm going to give you a small conventional C Programming Hello World program, You can try it using Demo link.

public class MyFirstJavaProgram {

/\* This is my first java program.

\* This will print 'Hello World' as the output

\*/

public static void main(String []args) {

System.out.println("Hello World"); // prints Hello World

}

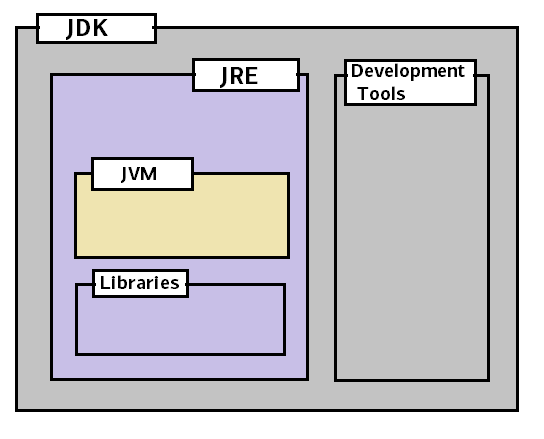
}

**Applications of Java Programming**

Java is guaranteed to be **Write Once, Run Anywhere.**

* **Multithreaded** − With Java's multithreaded feature it is possible to write programs that can perform many tasks simultaneously. This design feature allows the developers to construct interactive applications that can run smoothly.
* **Interpreted** − Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light-weight process.
* **High Performance** − With the use of Just-In-Time compilers, Java enables high performance.
* **Dynamic** − Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

**How Java works** ????



## Basic Syntax

About Java programs, it is very important to keep in mind the following points.

* **Case Sensitivity** − Java is case sensitive, which means identifier **Hello** and **hello** would have different meaning in Java.
* **Class Names** − For all class names the first letter should be in Upper Case. If several words are used to form a name of the class, each inner word's **first letter** **should be in Upper Case**.

**Example:** *class EdubridgeIndia*

**Method Names** − All method names should start with a Lower Case letter. If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.

**Example:**  *void edubridgeExample()*

* **Program File Name** − Name of the program file should exactly match the class name.

When saving the file, you should save it using the class name (Remember Java is case sensitive) and append '.java' to the end of the name (if the file name and the class name do not match, your program will not compile).

But please make a note that in case you do not have a public class present in the file then file name can be different than class name. It is also not mandatory to have a public class in the file.

**Example:** Assume 'MyFirstJavaProgram' is the class name. Then the file should be saved as *'MyFirstJavaProgram.java'*

* **public static void main(String args[])** − Java program processing starts from the main() method which is a mandatory part of every Java program.
* **Print,Println and Printf**

**import** java.io.\*;

**public** **class** PrintDemo

{

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

{

//declaration of different datatypes

**int** num = 122;

**char** ch = 'A';

String str = "Oracle";

**double** d = 190.98;

**float** f = 3.14f;

//prints the values on the console

System.out.println(); //prints nothing but throws the cursor to the next line

System.out.println(num); //prints integer

System.out.println(ch); //prints character

System.out.print(str+"\n");

System.out.print(d +"\n");

System.out.print(f+"\n");

System.out.printf("'%s' %n", "javatpoint");

System.out.printf("'%S' %n", "Jack");

}

}

## Java Identifiers

All Java components require names. Names used for classes, variables, and methods are called **identifiers**.

In Java, there are several points to remember about identifiers. They are as follows −

* All identifiers should begin with a letter (A to Z or a to z), currency character ($) or an underscore (\_).
* After the first character, identifiers can have any combination of characters.
* A key word cannot be used as an identifier.
* Most importantly, identifiers are case sensitive.
* Examples of legal identifiers: age, $salary, \_value, \_\_1\_value.
* Examples of illegal identifiers: 123abc, #salary,&salary.

## Java Modifiers

Like other languages, it is possible to modify classes, methods, etc., by using modifiers. There are two categories of modifiers −

* **Access Specifiers** − default, public , protected, private
* **Access Modifiers** − final, abstract, static

We will be looking into more details about modifiers in the next section.

## Java Variables

Following are the types of variables in Java −

* Local Variables
* Class Variables
* Static Variables

## Java Keywords

The following list shows the reserved words in Java. These reserved words may not be used as constant or variable or any other identifier names.

|  |  |  |  |
| --- | --- | --- | --- |
| abstract | assert | boolean | break |
| byte | case | catch | char |
| class | const | continue | default |
| do | double | else | enum |
| extends | final | finally | float |
| for | goto | if | implements |
| import | instanceof | int | interface |
| long | native | new | package |
| private | protected | public | return |
| short | static | strictfp | super |
| switch | synchronized | this | throw |
| throws | transient | try | void |
| volatile | while |  |  |

## Comments in Java

Java supports single-line and multi-line comments very similar to C and C++. All characters available inside any comment are ignored by Java compiler.

### **Example**

public class MyFirstJavaProgram {

/\* This is my first java program.

\* This will print 'Hello World' as the output

\* This is an example of multi-line comments.

\*/

public static void main(String []args) {

// This is an example of single line comment

/\* This is also an example of single line comment. \*/

System.out.println("Hello World");

}

}

### **Output**

Hello World

## Using Blank Lines

A line containing only white space, possibly with a comment, is known as a blank line, and Java totally ignores it.

# Java - Basic Datatypes

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in the memory.

Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

There are two data types available in Java −

* Primitive Data Types
* Reference/Object Data Types (classes , arrays , Interfaces , Strings .. etc)

## Primitive Data Types

There are eight primitive datatypes supported by Java. Primitive datatypes are predefined by the language and named by a keyword. Let us now look into the eight primitive data types in detail.

### **byte**

* Byte data type is an 8-bit signed **two's complement** integer
* Minimum value is -128 (-2^7)
* Maximum value is 127 (inclusive)(2^7 -1)
* Default value is 0
* Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an integer.
* Example: byte a = 100, byte b = -50

### **short**

* Short data type is a 16-bit signed **two's complement** integer
* Minimum value is -32,768 (-2^15)
* Maximum value is 32,767 (inclusive) (2^15 -1)
* Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an integer
* Default value is 0.
* Example: short s = 10000, short r = -20000

### **int**

* Int data type is a 32-bit signed two's complement integer.
* Minimum value is - 2,147,483,648 (-2^31)
* Maximum value is 2,147,483,647(inclusive) (2^31 -1)
* Integer is generally used as the default data type for integral values unless there is a concern about memory.
* The default value is 0
* Example: int a = 100000, int b = -200000

### **long**

* Long data type is a 64-bit signed two's complement integer
* Minimum value is -9,223,372,036,854,775,808(-2^63)
* Maximum value is 9,223,372,036,854,775,807 (inclusive)(2^63 -1)
* This type is used when a wider range than int is needed
* Default value is 0L
* Example: long a = 100000L, long b = -200000L

### **float**

* Float data type is a single-precision 32-bit IEEE 754 floating point
* Float is mainly used to save memory in large arrays of floating point numbers
* Default value is 0.0f
* Float data type is never used for precise values such as currency
* Example: float f1 = 234.5f

### **double**

* double data type is a double-precision 64-bit IEEE 754 floating point
* This data type is generally used as the default data type for decimal values, generally the default choice
* Double data type should never be used for precise values such as currency
* Default value is 0.0d
* Example: double d1 = 123.4

### **boolean**

* boolean data type represents one bit of information
* There are only two possible values: true and false
* This data type is used for simple flags that track true/false conditions
* Default value is false
* Example: boolean one = true;

### **char**

* char data type is a single 16-bit Unicode character
* Minimum value is '\u0000' (or 0)
* Maximum value is '\uffff' (or 65,535 inclusive)
* Char data type is used to store any character
* Example: char letterA = 'A'

## Reference Datatypes

* Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy, etc.
* Class objects and various type of array variables come under reference datatype.
* Default value of any reference variable is null.
* A reference variable can be used to refer any object of the declared type or any compatible type.
* Example: Animal animal = new Animal("giraffe");

## Java Literals

A literal is a source code representation of a **fixed value**. They are represented directly in the code without any computation.

Literals can be assigned to any primitive type variable. For example −

byte a = 68;

char a = 'A';

byte, int, long, and short can be expressed in decimal(base 10), hexadecimal(base 16) or octal(base 8) number systems as well.

Prefix 0 is used to indicate octal, and prefix 0x indicates hexadecimal when using these number systems for literals. For example −

int decimal = 100;

int octal = 0144;

int hexa = 0x64;

### **Example**

"Hello World"

"two\nlines"

"\"This is in quotes\""

String and char types of literals can contain any Unicode characters. For example −

char a = '\u0001';

String a = "\u0001";

Java language supports few special escape sequences for String and char literals as well. They are −

|  |  |
| --- | --- |
| **Notation** | **Character represented** |
| \n | Newline (0x0a) |
| \b | Backspace (0x08) |
| \s | Space (0x20) |
| \t | tab |
| \" | Double quote |
| \' | Single quote |
| \\ | backslash |

# Java Type Casting

## Java Type Casting

Type casting is when you assign a value of one primitive data type to another type.

In Java, there are two types of casting:

* **Widening Casting** (automatically) - converting a smaller type to a larger type size  
  byte -> short -> char -> int -> long -> float -> double
* **Narrowing Casting** (manually) - converting a larger type to a smaller size type  
  double -> float -> long -> int -> char -> short -> byte

## Widening Casting

Widening casting is done automatically when passing a smaller size type to a larger size type:

public class Main {

public static void main(String[] args) {

int myInt = 9;

double myDouble = myInt; // Automatic casting: int to double

System.out.println(myInt); // Outputs 9

System.out.println(myDouble); // Outputs 9.0

}

}

## Narrowing Casting

Narrowing casting must be done manually by placing the type in parentheses in front of the value:

### **Example**

public class Main {

public static void main(String[] args) {

double myDouble = 9.78d;

int myInt = (int) myDouble; // Manual casting: double to int

System.out.println(myDouble); // Outputs 9.78

System.out.println(myInt); // Outputs 9

}

}

# Java Operators

In this tutorial, you'll learn about different types of operators in Java, their syntax and how to use them with the help of examples.

Operators are symbols that perform operations on variables and values. For example, + is an operator used for addition, while \* is also an operator used for multiplication.

Operators in Java can be classified into 5 types:

1. Arithmetic Operators
2. Assignment Operators
3. Relational Operators
4. Logical Operators
5. Unary Operators
6. Bitwise Operators

## 1. Java Arithmetic Operators

Arithmetic operators are used to perform arithmetic operations on variables and data. For example,

a + b;

Here, the + operator is used to add two variables a and b. Similarly, there are various other arithmetic operators in Java.

|  |  |
| --- | --- |
| Operator | Operation |
| + | Addition |
| - | Subtraction |
| \* | Multiplication |
| / | Division |
| % | Modulo Operation (Remainder after division) |

### Example 1: Arithmetic Operators

class Main {

public static void main(String[] args) {

// declare variables

int a = 12, b = 5;

// addition operator

System.out.println("a + b = " + (a + b));

// subtraction operator

System.out.println("a - b = " + (a - b));

// multiplication operator

System.out.println("a \* b = " + (a \* b));

// division operator

System.out.println("a / b = " + (a / b));

// modulo operator

System.out.println("a % b = " + (a % b));

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**

a + b = 17

a - b = 7

a \* b = 60

a / b = 2

a % b = 2

In the above example, we have used +, -, and \* operators to compute addition, subtraction, and multiplication operations.

**/ Division Operator**

Note the operation, a / b in our program. The / operator is the division operator.

If we use the division operator with two integers, then the resulting quotient will also be an integer. And, if one of the operands is a floating-point number, we will get the result will also be in floating-point.

In Java,

(9 / 2) is 4

(9.0 / 2) is 4.5

(9 / 2.0) is 4.5

(9.0 / 2.0) is 4.5

**% Modulo Operator**

The modulo operator % computes the remainder. When a = 7 is divided by b = 4, the remainder is **3**.

**Note**: The % operator is mainly used with integers.

## 2. Java Assignment Operators

Assignment operators are used in Java to assign values to variables. For example,

int age;

age = 5;

Here, = is the assignment operator. It assigns the value on its right to the variable on its left. That is, **5** is assigned to the variable age.

Let's see some more assignment operators available in Java.

|  |  |  |
| --- | --- | --- |
| Operator | Example | Equivalent to |
| = | a = b; | a = b; |
| += | a += b; | a = a + b; |
| -= | a -= b; | a = a - b; |
| \*= | a \*= b; | a = a \* b; |
| /= | a /= b; | a = a / b; |
| %= | a %= b; | a = a % b; |

### Example 2: Assignment Operators

class Main {

public static void main(String[] args) {

// create variables

int a = 4;

int var;

// assign value using =

var = a;

System.out.println("var using =: " + var);

// assign value using =+

var += a;

System.out.println("var using +=: " + var);

// assign value using =\*

var \*= a;

System.out.println("var using \*=: " + var);

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**

var using =: 4

var using +=: 8

var using \*=: 32

## 3. Java Relational Operators

Relational operators are used to check the relationship between two operands. For example,

// check if a is less than b

a < b;

Here, < operator is the relational operator. It checks if a is less than b or not.

It returns either true or false.

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| == | Is Equal To | 3 == 5 returns **false** |
| != | Not Equal To | 3 != 5 returns **true** |
| > | Greater Than | 3 > 5 returns **false** |
| < | Less Than | 3 < 5 returns **true** |
| >= | Greater Than or Equal To | 3 >= 5 returns **false** |
| <= | Less Than or Equal To | 3 <= 5 returns **true** |

### Example 3: Relational Operators

class Main {

public static void main(String[] args) {

// create variables

int a = 7, b = 11;

// value of a and b

System.out.println("a is " + a + " and b is " + b);

// == operator

System.out.println(a == b); // false

// != operator

System.out.println(a != b); // true

// > operator

System.out.println(a > b); // false

// < operator

System.out.println(a < b); // true

// >= operator

System.out.println(a >= b); // false

// <= operator

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

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Note**: Relational operators are used in decision making and loops.

## 4. Java Logical Operators

Logical operators are used to check whether an expression is true or false. They are used in decision making.

|  |  |  |
| --- | --- | --- |
| Operator | Example | Meaning |
| && (Logical AND) | expression1 **&&** expression2 | true only if both expression1 and expression2 are true |
| || (Logical OR) | expression1 **||** expression2 | true if either expression1 or expression2 is true |
| ! (Logical NOT) | **!**expression | true if expression is false and vice versa |

### Example 4: Logical Operators

class Main {

public static void main(String[] args) {

// && operator

System.out.println((5 > 3) && (8 > 5)); // true

System.out.println((5 > 3) && (8 < 5)); // false

// || operator

System.out.println((5 < 3) || (8 > 5)); // true

System.out.println((5 > 3) || (8 < 5)); // true

System.out.println((5 < 3) || (8 < 5)); // false

// ! operator

System.out.println(!(5 == 3)); // true

System.out.println(!(5 > 3)); // false

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Working of Program**

* (5 > 3) && (8 > 5) returns true because both (5 > 3) and (8 > 5) are true.
* (5 > 3) && (8 < 5) returns false because the expression (8 < 5) is false.
* (5 < 3) || (8 > 5) returns true because the expression (8 > 5) is true.
* (5 > 3) || (8 < 5) returns true because the expression (5 > 3) is true.
* (5 < 3) || (8 < 5) returns false because both (5 < 3) and (8 < 5) are false.
* !(5 == 3) returns true because 5 == 3 is false.
* !(5 > 3) returns false because 5 > 3 is true.

## 5. Java Unary Operators

Unary operators are used with only one operand. For example, ++ is a unary operator that increases the value of a variable by **1**. That is, ++5 will return **6**.

Different types of unary operators are:

|  |  |
| --- | --- |
| Operator | Meaning |
| + | **Unary plus**: not necessary to use since numbers are positive without using it |
| - | **Unary minus**: inverts the sign of an expression |
| ++ | **Increment operator**: increments value by 1 |
| -- | **Decrement operator**: decrements value by 1 |
| ! | **Logical complement operator**: inverts the value of a boolean |

## Increment and Decrement Operators

Java also provides increment and decrement operators: ++ and -- respectively. ++ increases the value of the operand by **1**, while -- decrease it by **1**. For example,

int num = 5;

// increase num by 1

++num;

Here, the value of num gets increased to **6** from its initial value of **5**.

### Example 5: Increment and Decrement Operators

class Main {

public static void main(String[] args) {

// declare variables

int a = 12, b = 12;

int result1, result2;

// original value

System.out.println("Value of a: " + a);

// increment operator

result1 = ++a;

System.out.println("After increment: " + result1);

System.out.println("Value of b: " + b);

// decrement operator

result2 = --b;

System.out.println("After decrement: " + result2);

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**

Value of a: 12

After increment: 13

Value of b: 12

After decrement: 11

In the above program, we have used the ++ and -- operator as **prefixes (++a, --b)**. We can also use these operators as **postfix (a++, b++)**.

There is a slight difference when these operators are used as prefix versus when they are used as a postfix.

To learn more about these operators, visit [increment and decrement operators](https://www.programiz.com/article/increment-decrement-operator-difference-prefix-postfix).

## 6. Java Bitwise Operators

Bitwise operators in Java are used to perform operations on individual bits. For example,

Bitwise complement Operation of 35

35 = 00100011 (In Binary)

~ 00100011

\_\_\_\_\_\_\_\_

11011100 = 220 (In decimal)

Here, ~ is a bitwise operator. It inverts the value of each bit (**0** to **1** and **1** to **0**).

The various bitwise operators present in Java are:

|  |  |
| --- | --- |
| Operator | Description |
| ~ | Bitwise Complement |
| << | Left Shift |
| >> | Right Shift |
| >>> | Unsigned Right Shift |
| & | Bitwise AND |
| ^ | Bitwise exclusive OR |

These operators are not generally used in Java. To learn more, visit [Java Bitwise and Bit Shift Operators](https://www.programiz.com/java-programming/bitwise-operators).

## Other operators

Besides these operators, there are other additional operators in Java.

### Java instanceof Operator

The instanceof operator checks whether an object is an instanceof a particular class. For example,

class Main {

public static void main(String[] args) {

String str = "Programiz";

boolean result;

// checks if str is an instance of

// the String class

result = str instanceof String;

System.out.println("Is str an object of String? " + result);

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**

Is str an object of String? true

Here, str is an instance of the String class. Hence, the instanceof operator returns true. To learn more, visit [Java instanceof](https://www.programiz.com/java-programming/instanceof).

### Java Ternary Operator

The ternary operator (conditional operator) is shorthand for the if-then-else statement. For example,

variable = Expression ? expression1 : expression2

Here's how it works.

* If the Expression is true, expression1 is assigned to the variable.
* If the Expression is false, expression2 is assigned to the variable.

Let's see an example of a ternary operator.

class Java {

public static void main(String[] args) {

int februaryDays = 29;

String result;

// ternary operator

result = (februaryDays == 28) ? "Not a leap year" : "Leap year";

System.out.println(result);

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**

Leap year

In the above example, we have used the ternary operator to check if the year is a leap year or not. To learn more, visit [the Java ternary operator](https://www.programiz.com/java-programming/ternary-operator).

# Static blocks in Java

Unlike C++, Java supports a special block, called static block (also called static clause) which can be used for static initializations of a class. This code inside static block is executed only once: the first time the class is loaded into memory. For example, check output of following Java program.

|  |
| --- |
| **class** Test {  **static** **int** *i*;  **int** j;      **static** {  *i* = 10;  System.***out***.println("123 Jayanth");  }    **static** {  System.***out***.println("jayannth");  }    Test(){    System.***out***.println("jayanthhhhhhhh");  }    }    **public** **class** Addition {  **public** **static** **void** main(String args[]) {    Test t =**new** Test();    Test t1=**new** Test();    }  } |

Output:  
123 Jayanth

jayannth

jayanthhhhhhhh

jayanthhhhhhhh

Also, static blocks are executed before constructors. For example, check output of following Java program.

|  |
| --- |
| class Test {      static int i;      int j;      static {          i = 10;          System.out.println("static block called ");      }      Test(){          System.out.println("Constructor called");      }  }    class Main {      public static void main(String args[]) {           // Although we have two objects, static block is executed only once.         Test t1 = new Test();         Test t2 = new Test();      }  } |

Output:  
static block calledConstructor calledConstructor called

# The Initializer Block in Java

Initializer block contains the code that is always executed whenever an instance is created. It is used to declare/initialize the common part of various constructors of a class. For example,

|  |
| --- |
| import java.io.\*;  public class GFG  {      // Initializer block starts..      {          // This code is executed before every constructor.          System.out.println("Common part of constructors invoked !!");      }      // Initializer block ends        public GFG()      {          System.out.println("Default Constructor invoked");      }      public GFG(int x)      {          System.out.println("Parametrized constructor invoked");      }      public static void main(String arr[])      {          GFG obj1, obj2;          obj1 = new GFG();          obj2 = new GFG(0);      }  } |

**Output:**

Common part of constructors invoked!!

Default Constructor invoked

Common part of constructors invoked!!

Parametrized constructor invoked

# Java Inner Classes

## Java Inner Classes

In Java, it is also possible to nest classes (a class within a class). The purpose of nested classes is to group classes that belong together, which makes your code more readable and maintainable.

To access the inner class, create an object of the outer class, and then create an object of the inner class:

### **Example**

class OuterClass {

int x = 10;

class InnerClass {

int y = 5;

}

}

public class MyMainClass {

public static void main(String[] args) {

OuterClass myOuter = new OuterClass();

OuterClass.InnerClass myInner = myOuter.new InnerClass();

System.out.println(myInner.y + myOuter.x);

}

}

// Outputs 15 (5 + 10)

## Private Inner Class

Unlike a "regular" class, an inner class can be private or protected. If you don't want outside objects to access the inner class, declare the class as private:

### **Example**

class OuterClass {

int x = 10;

**private** class InnerClass {

int y = 5;

}

}

public class MyMainClass {

public static void main(String[] args) {

OuterClass myOuter = new OuterClass();

OuterClass.InnerClass myInner = myOuter.new InnerClass();

System.out.println(myInner.y + myOuter.x);

}

}

If you try to access a private inner class from an outside class (MyMainClass), an error occurs:

MyMainClass.java:12: error: OuterClass.InnerClass has private access in OuterClass  
    OuterClass.InnerClass myInner = myOuter.new InnerClass();  
              ^

## Static Inner Class

An inner class can also be static, which means that you can access it without creating an object of the outer class:

### **Example**

class OuterClass {

int x = 10;

static class InnerClass {

int y = 5;

}

}

public class MyMainClass {

public static void main(String[] args) {

OuterClass.InnerClass myInner = new OuterClass.InnerClass();

System.out.println(myInner.y);

}

}

// Outputs 5

**Note:** just like static attributes and methods, a static inner class does not have access to members of the outer class.

## Access Outer Class From Inner Class

One advantage of inner classes, is that they can access attributes and methods of the outer class:

### **Example**

class OuterClass {

int x = 10;

class InnerClass {

public int myInnerMethod() {

return x;

}

}

}

public class MyMainClass {

public static void main(String[] args) {

OuterClass myOuter = new OuterClass();

OuterClass.InnerClass myInner = myOuter.new InnerClass();

System.out.println(myInner.myInnerMethod());

}

}

// Outputs 10

**Static nested classes**

In the case of normal or regular inner classes, without an outer class object existing, there cannot be an inner class object. i.e., an object of the inner class is always strongly associated with an outer class object. But in the case of static nested class, Without an outer class object existing, there may be a static nested class object. i.e., an object of a static nested class is not strongly associated with the outer class object.  
As with class methods and variables, a static nested class is associated with its outer class. And like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class: it can use them only through an object reference.  
They are accessed using the enclosing class name.

OuterClass.StaticNestedClass

For example, to create an object for the static nested class, use this syntax:

OuterClass.StaticNestedClass nestedObject =

new OuterClass.StaticNestedClass();

|  |
| --- |
| // Java program to demonstrate accessing  // a static nested class    // outer class  class OuterClass  {      // static member      static int outer\_x = 10;        // instance(non-static) member      int outer\_y = 20;        // private member      private static int outer\_private = 30;        // static nested class      static class StaticNestedClass      {          void display()          {              // can access static member of outer class              System.out.println("outer\_x = " + outer\_x);                // can access display private static member of outer class              System.out.println("outer\_private = " + outer\_private);                // The following statement will give compilation error              // as static nested class cannot directly access non-static membera              // System.out.println("outer\_y = " + outer\_y);            }      }  }    // Driver class  public class StaticNestedClassDemo  {      public static void main(String[] args)      {          // accessing a static nested class          OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();            nestedObject.display();        }  } |

**Output:**

outer\_x = 10

outer\_private = 30

# Decision Making

# if statement in java

An **if** statement consists of a Boolean expression followed by one or more statements.

## Syntax

Following is the syntax of an if statement −

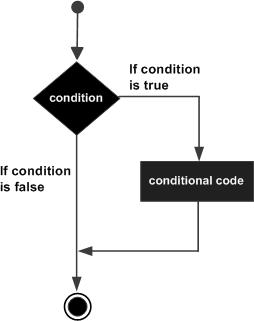
if(Boolean\_expression) {

// Statements will execute if the Boolean expression is true

}

If the Boolean expression evaluates to true then the block of code inside the if statement will be executed. If not, the first set of code after the end of the if statement (after the closing curly brace) will be executed.

## Flow Diagram



## Example

public class Test {

public static void main(String args[]) {

int x = 10;

if( x < 20 ) {

System.out.print("This is if statement");

}

}

}

This will produce the following result −

## Output

This is if statement.

# if-else statement in java

An **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false.

## Syntax

Following is the syntax of an if...else statement −

if(Boolean\_expression) {

// Executes when the Boolean expression is true

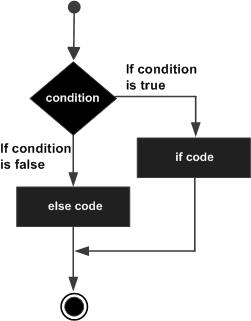
}else {

// Executes when the Boolean expression is false

}

If the boolean expression evaluates to true, then the if block of code will be executed, otherwise else block of code will be executed.

## Flow Diagram



## Example

public class Test {

public static void main(String args[]) {

int x = 30;

if( x < 20 ) {

System.out.print("This is if statement");

}else {

System.out.print("This is else statement");

}

}

}

This will produce the following result −

## Output

This is else statement

## The if...else if...else Statement

An if statement can be followed by an optional *else if...else* statement, which is very useful to test various conditions using single if...else if statement.

When using if, else if, else statements there are a few points to keep in mind.

* An if can have zero or one else's and it must come after any else if's.
* An if can have zero to many else if's and they must come before the else.
* Once an else if succeeds, none of the remaining else if's or else's will be tested.

## Syntax

Following is the syntax of an if...else statement −

if(Boolean\_expression 1) {

// Executes when the Boolean expression 1 is true

}else if(Boolean\_expression 2) {

// Executes when the Boolean expression 2 is true

}else if(Boolean\_expression 3) {

// Executes when the Boolean expression 3 is true

}else {

// Executes when the none of the above condition is true.

}

## Example

public class Test {

public static void main(String args[]) {

int x = 30;

if( x == 10 ) {

System.out.print("Value of X is 10");

}else if( x == 20 ) {

System.out.print("Value of X is 20");

}else if( x == 30 ) {

System.out.print("Value of X is 30");

}else {

System.out.print("This is else statement");

}

}

}

This will produce the following result −

## Output

Value of X is 30

# nested if statement in java

It is always legal to nest if-else statements which means you can use one if or else if statement inside another if or else if statement.

## Syntax

The syntax for a nested if...else is as follows −

if(Boolean\_expression 1) {

// Executes when the Boolean expression 1 is true

if(Boolean\_expression 2) {

// Executes when the Boolean expression 2 is true

}

}

You can nest **else if...else** in the similar way as we have nested *if* statement.

## Example

public class Test {

public static void main(String args[]) {

int x = 30;

int y = 10;

if( x == 30 ) {

if( y == 10 ) {

System.out.print("X = 30 and Y = 10");

}

}

}

}

This will produce the following result −

## Output

X = 30 and Y = 10

# switch statement in java

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

## Syntax

The syntax of enhanced for loop is −

switch(expression) {

case value :

// Statements

break; // optional

case value :

// Statements

break; // optional

// You can have any number of case statements.

default : // Optional

// Statements

}

The following rules apply to a **switch** statement −

* The variable used in a switch statement can only be integers, convertable integers (byte, short, char), strings and enums.
* You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
* The value for a case must be the same data type as the variable in the switch and it must be a constant or a literal.
* When the variable being switched on is equal to a case, the statements following that case will execute until a *break* statement is reached.
* When a *break* statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
* Not every case needs to contain a break. If no break appears, the flow of control will *fall through*to subsequent cases until a break is reached.
* A *switch* statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

## Flow Diagram



## Example

public class Test {

public static void main(String args[]) {

// char grade = args[0].charAt(0);

char grade = 'C';

switch(grade) {

case 'A' :

System.out.println("Excellent!");

break;

case 'B' :

case 'C' :

System.out.println("Well done");

break;

case 'D' :

System.out.println("You passed");

case 'F' :

System.out.println("Better try again");

break;

default :

System.out.println("Invalid grade");

}

System.out.println("Your grade is " + grade);

}

}

Compile and run the above program using various command line arguments. This will produce the following result −

## Output

Well done

Your grade is C

# Loop Control

# while Loop in java

A **while** loop statement in Java programming language repeatedly executes a target statement as long as a given condition is true.

## Syntax

The syntax of a while loop is −

while(Boolean\_expression) {

// Statements

}

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non zero value.

When executing, if the *boolean\_expression* result is true, then the actions inside the loop will be executed. This will continue as long as the expression result is true.

When the condition becomes false, program control passes to the line immediately following the loop.

## Flow Diagram



Here, key point of the *while* loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

## Example

public class Test {

public static void main(String args[]) {

int x = 10;

while( x < 20 ) {

System.out.print("value of x : " + x );

x++;

System.out.print("\n");

}

}

}

This will produce the following result −

## Output

value of x : 10

value of x : 11

value of x : 12

value of x : 13

value of x : 14

value of x : 15

value of x : 16

value of x : 17

value of x : 18

value of x : 19

# for loop in java

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to be executed a specific number of times.

A **for** loop is useful when you know how many times a task is to be repeated.

## Syntax

The syntax of a for loop is −

for(initialization; Boolean\_expression; update) {

// Statements

}

Here is the flow of control in a **for** loop −

* The **initialization** step is executed first, and only once. This step allows you to declare and initialize any loop control variables and this step ends with a semi colon (;).
* Next, the **Boolean expression** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop will not be executed and control jumps to the next statement past the for loop.
* After the **body** of the for loop gets executed, the control jumps back up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank with a semicolon at the end.
* The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats (body of loop, then update step, then Boolean expression). After the Boolean expression is false, the for loop terminates.

## Flow Diagram



## Example

Following is an example code of the for loop in Java.

public class Test {

public static void main(String args[]) {

for(int x = 10; x < 20; x = x + 1) {

System.out.print("value of x : " + x );

System.out.print("\n");

}

}

}

This will produce the following result −

## Output

value of x : 10

value of x : 11

value of x : 12

value of x : 13

value of x : 14

value of x : 15

value of x : 16

value of x : 17

value of x : 18

value of x : 19

# do while loop in java

A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

## Syntax

Following is the syntax of a do...while loop −

do {

// Statements

}while(Boolean\_expression);

Notice that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested.

If the Boolean expression is true, the control jumps back up to do statement, and the statements in the loop execute again. This process repeats until the Boolean expression is false.

## Flow Diagram



## Example

public class Test {

public static void main(String args[]) {

int x = 10;

do {

System.out.print("value of x : " + x );

x++;

System.out.print("\n");

}while( x < 20 );

}

}

This will produce the following result −

## Output

value of x : 10

value of x : 11

value of x : 12

value of x : 13

value of x : 14

value of x : 15

value of x : 16

value of x : 17

value of x : 18

value of x : 19

**For Each :**

**Example :**

**int** a[]={50,60,70,80,90,100};

**for** (**int** a1:a){

System.***out***.println(a1);

}

# Java - Numbers Class

Normally, when we work with Numbers, we use primitive data types such as byte, int, long, double, etc.

### **Example**

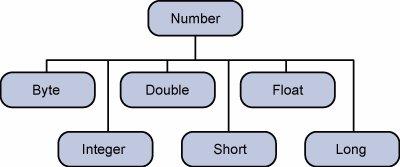
int i = 5000;

float gpa = 13.65;

double mask = 0xaf;

However, in development, we come across situations where we need to use objects instead of primitive data types. In order to achieve this, Java provides **wrapper classes**.

All the wrapper classes (Integer, Long, Byte, Double, Float, Short) are subclasses of the abstract class Number.



The object of the wrapper class contains or wraps its respective primitive data type. Converting primitive data types into object is called **boxing**, and this is taken care by the **compiler**.

And the Wrapper object will be converted back to a primitive data type, and this process is called **unboxing**. The **Number** class is part of the java.lang package.

Following is an example of boxing and unboxing −

### **Example**

public class Test {

public static void main(String args[]) {

Integer x = 5; // boxes int to an Integer object

x = x + 10; // unboxes the Integer to a int

System.out.println(x);

}

}

This will produce the following result −

### **Output**

15

# xxxValue() Method

## Description

The method converts the value of the Number Object that invokes the method to the primitive data type that is returned from the method.

## Syntax

Here is a separate method for each primitive data type −

byte byteValue()

short shortValue()

int intValue()

long longValue()

float floatValue()

double doubleValue()

## Parameters

Here is the detail of parameters −

* All these are default methods and accepts no parameter.

## Return Value

* This method returns the primitive data type that is given in the signature.

## Example

public class Test {

public static void main(String args[]) {

Integer x = 5;

// Returns byte primitive data type

System.out.println( x.byteValue() );

// Returns double primitive data type

System.out.println(x.doubleValue());

// Returns long primitive data type

System.out.println( x.longValue() );

}

}

This will produce the following result −

## Output

5

5.0

5

# Java - compareTo() Method

## Description

The method compares the Number object that invoked the method to the argument. It is possible to compare Byte, Long, Integer, etc.

However, two different types cannot be compared, both the argument and the Number object invoking the method should be of the same type.

## Syntax

public int compareTo( NumberSubClass referenceName )

## Parameters

Here is the detail of parameters −

* **referenceName** − This could be a Byte, Double, Integer, Float, Long, or Short.

## Return Value

* If the Integer is equal to the argument then 0 is returned.
* If the Integer is less than the argument then -1 is returned.
* If the Integer is greater than the argument then 1 is returned.

## Example

public class Test {

public static void main(String args[]) {

Integer x = 5;

System.out.println(x.compareTo(3));

System.out.println(x.compareTo(5));

System.out.println(x.compareTo(8));

}

}

This will produce the following result −

## Output

1

0

-1

# Java - equals() Method

## Description

The method determines whether the Number object that invokes the method is equal to the object that is passed as an argument.

## Syntax

public boolean equals(Object o)

## Parameters

Here is the detail of parameters −

* Any object.

## Return Value

* The method returns True if the argument is not null and is an object of the same type and with the same numeric value. There are some extra requirements for Double and Float objects that are described in the Java API documentation.

## Example

public class Test {

public static void main(String args[]) {

Integer x = 5;

Integer y = 10;

Integer z =5;

Short a = 5;

System.out.println(x.equals(y));

System.out.println(x.equals(z));

System.out.println(x.equals(a));

}

}

This will produce the following result −

## Output

false

true

false

# Java - parseInt() Method

## Description

This method is used to get the primitive data type of a certain String. parseXxx() is a static method and can have one argument or two.

## Syntax

Following are all the variants of this method −

static int parseInt(String s)

static int parseInt(String s, int radix)

## Parameters

Here is the detail of parameters −

* **s** − This is a string representation of decimal.
* **radix** − This would be used to convert String s into integer.

## Return Value

* **parseInt(String s)** − This returns an integer (decimal only).
* **parseInt(int i)** − This returns an integer, given a string representation of decimal, binary, octal, or hexadecimal (radix equals 10, 2, 8, or 16 respectively) numbers as input.

## Example

public class Test {

public static void main(String args[]) {

int x =Integer.parseInt("9");

double c = Double.parseDouble("5");

int b = Integer.parseInt("444",16);

System.out.println(x);

System.out.println(c);

System.out.println(b);

}

}

This will produce the following result −

## Output

9

5.0

1092

# Java - min() Method

## Description

The method gives the smaller of the two arguments. The argument can be int, float, long, double.

## Syntax

This method has the following variants −

double min(double arg1, double arg2)

float min(float arg1, float arg2)

int min(int arg1, int arg2)

long min(long arg1, long arg2)

## Parameters

Here is the detail of parameters −

* This method accepts any primitive data type as a parameter.

## Return Value

* This method returns the smaller of the two arguments.

## Example

public class Test {

public static void main(String args[]) {

System.out.println(Math.min(12.123, 12.456));

System.out.println(Math.min(23.12, 23.0));

}

}

This will produce the following result −

## Output

12.123

23.0

|  |
| --- |
| [pow()](https://www.tutorialspoint.com/java/number_pow.htm)  Returns the value of the first argument raised to the power of the second argument. |
|  | [sqrt()](https://www.tutorialspoint.com/java/number_sqrt.htm)  Returns the square root of the argument. |

# Java - toString() Method

## Description

The method is used to get a String object representing the value of the Number Object.

If the method takes a primitive data type as an argument, then the String object representing the primitive data type value is returned.

If the method takes two arguments, then a String representation of the first argument in the radix specified by the second argument will be returned.

## Syntax

Following are all the variants of this method −

String toString()

static String toString(int i)

## Parameters

Here is the detail of parameters −

* **i** − An int for which string representation would be returned.

## Return Value

* **toString()** − This returns a String object representing the value of **this** Integer.
* **toString(int i)** − This returns a String object representing the specified integer.

## Example

public class Test {

public static void main(String args[]) {

Integer x = 5;

System.out.println(x.toString());

System.out.println(Integer.toString(12));

}

}

This will produce the following result −

## Output

5

12

# Java Enums

## Enums

An enum is a special "class" that represents a group of **constants** (unchangeable variables, like final variables).

To create an enum, use the enum keyword (instead of class or interface), and separate the constants with a **comma**. Note that they should be in **uppercase letters**:

### **Example**

enum Level {

LOW,

MEDIUM,

HIGH

}

You can access enum constants with the **dot** syntax:

Level myVar = Level.MEDIUM;

**Enum** is short for "enumerations", which means "specifically listed".

## Enum inside a Class

You can also have an enum inside a class:

### **Example**

public class MyClass {

enum Level {

LOW,

MEDIUM,

HIGH

}

public static void main(String[] args) {

Level myVar = Level.MEDIUM;

System.out.println(myVar);

}

}

The output will be:

MEDIUM

## Enum in a Switch Statement

Enums are often used in switch statements to check for corresponding values:

### **Example**

enum Level {

LOW,

MEDIUM,

HIGH

}

public class MyClass {

public static void main(String[] args) {

Level myVar = Level.MEDIUM;

switch(myVar) {

case LOW:

System.out.println("Low level");

break;

case MEDIUM:

System.out.println("Medium level");

break;

case HIGH:

System.out.println("High level");

break;

}

}

}

The output will be:

Medium level

## Loop Through an Enum

The enum type has a values() method, which returns an array of all enum constants. This method is useful when you want to loop through the constants of an enum:

### **Example**

for (Level ll: Level.values()) {

System.out.println(ll);

}

The output will be:

LOW  
MEDIUM  
HIGH

#### **Difference between Enums and Classes**

An enum can, just like a class, have attributes and methods. The only difference is that enum constants are public, static and final (unchangeable - cannot be overridden).

An enum cannot be used to create objects, and it cannot extend other classes (but it can implement interfaces).

#### **Why And When To Use Enums?**

Use enums when you have values that you know aren't going to change, like month days, days, colors, deck of cards, etc.

# Java User Input (Scanner)

## Java User Input

The Scanner class is used to get user input, and it is found in the java.util package.

To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class documentation. In our example, we will use the nextLine() method, which is used to read Strings:

### **Example**

import java.util.Scanner; // Import the Scanner class

class MyClass {

public static void main(String[] args) {

Scanner myObj = new Scanner(System.in); // Create a Scanner object

System.out.println("Enter username");

String userName = myObj.nextLine(); // Read user input

System.out.println("Username is: " + userName); // Output user input

}

## Input Types

In the example above, we used the nextLine() method, which is used to read Strings. To read other types, look at the table below:

|  |  |
| --- | --- |
| **Method** | **Description** |
| nextBoolean() | Reads a boolean value from the user |
| nextByte() | Reads a byte value from the user |
| nextDouble() | Reads a double value from the user |
| nextFloat() | Reads a float value from the user |
| nextInt() | Reads a int value from the user |
| nextLine() | Reads a String value from the user |
| nextLong() | Reads a long value from the user |
| nextShort() | Reads a short value from the user |

In the example below, we use different methods to read data of various types:

### **Example**

import java.util.Scanner;

class MyClass {

public static void main(String[] args) {

Scanner myObj = new Scanner(System.in);

System.out.println("Enter name, age and salary:");

// String input

String name = myObj.nextLine();

// Numerical input

int age = myObj.nextInt();

double salary = myObj.nextDouble();

// Output input by user

System.out.println("Name: " + name);

System.out.println("Age: " + age);

System.out.println("Salary: " + salary);

}

}