Java

JavaHome:

* Java is a programming language.
* Java is used to develop mobile apps, web apps, desktop apps, games and much more.

Example:

public class MyClass {

public static void main(String[] args) {

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

}

}

Introduction:

What is Java?

Java is a popular programming language, created in 1995.

It is owned by Oracle, and more than **3 billion** devices run Java.

It is used for:

* Mobile applications (specially Android apps)
* Desktop applications
* Web applications
* Web servers and application servers
* Games
* Database connection
* And much, much more!

## Why Use Java?

* Java works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.)
* It is one of the most popular programming language in the world
* It is easy to learn and simple to use
* It is open-source and free
* It is secure, fast and powerful
* It has a huge community support (tens of millions of developers)
* Java is an object oriented language which gives a clear structure to programs and allows code to be reused, lowering development costs
* As Java is close to [C++](https://www.w3schools.com/cpp/default.asp) and [C#](https://www.w3schools.com/cs/default.asp), it makes it easy for programmers to switch to Java or vice versa

Setup for Windows

To install Java on Windows:

1. Go to "System Properties" (Can be found on Control Panel > System and Security > System > Advanced System Settings)
2. Click on the "Environment variables" button under the "Advanced" tab
3. Then, select the "Path" variable in System variables and click on the "Edit" button
4. Click on the "New" button and add the path where Java is installed, followed by **\bin**. By default, Java is installed in C:\Program Files\Java\jdk-11.0.1 (If nothing else was specified when you installed it). In that case, You will have to add a new path with: **C:\Program Files\Java\jdk-11.0.1\bin**  
   Then, click "OK", and save the settings
5. At last, open Command Prompt (cmd.exe) and type **java -version** to see if Java is running on your machine

## Java Quickstart

In Java, every application begins with a class name, and that class must match the filename.

Let's create our first Java file, called MyClass.java, which can be done in any text editor (like Notepad).

The file should contain a "Hello World" message, which is written with the following code:

MyClass.java

public class MyClass {

public static void main(String[] args) {

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

}

}

Save the code in Notepad as "MyClass.java". Open Command Prompt (cmd.exe), navigate to the directory where you saved your file, and type "javac MyClass.java":

C:\Users\*Your Name*>javac MyClass.java

This will compile your code. If there are no errors in the code, the command prompt will take you to the next line. Now, type "java MyClass" to run the file:

C:\Users\*Your Name*>java MyClass

Java syntax:

### **Example explained**

Every line of code that runs in Java must be inside a class. In our example, we named the class **MyClass**. A class should always start with an uppercase first letter.

**Note:** Java is case-sensitive: "MyClass" and "myclass" has different meaning.

**Note:** The curly braces {} marks the beginning and the end of a block of code.

**Note:** Each code statement must end with a semicolon.

Comments:

Single line comments //

Multiline comments /\* \*/

Java Variables

Variables are containers for storing data values.

In Java, there are different **types** of variables, for example:

* String - stores text, such as "Hello". String values are surrounded by double quotes
* int - stores integers (whole numbers), without decimals, such as 123 or -123
* float - stores floating point numbers, with decimals, such as 19.99 or -19.99
* char - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
* boolean - stores values with two states: true or false

## Final Variables

However, you can add the final keyword if you don't want others (or yourself) to overwrite existing values (this will declare the variable as "final" or "constant", which means unchangeable and read-only):

### **Example**

final int myNum = 15;

myNum = 20; // will generate an error: cannot assign a value to a final variable

Identifiers:

The general rules for constructing names for variables (unique identifiers) are:

* Names can contain letters, digits, underscores, and dollar signs
* Names must begin with a letter
* Names should start with a lowercase letter and it cannot contain whitespace
* Names can also begin with $ and \_ (but we will not use it in this tutorial)
* Names are case sensitive ("myVar" and "myvar" are different variables)
* Reserved words (like Java keywords, such as int or boolean) cannot be used as names

Datatypes:

Data types are divided into two groups:

* Primitive data types - includes byte, short, int, long, float, double, boolean and char
* Non-primitive data types - such as [String](https://www.w3schools.com/java/java_strings.asp), [Arrays](https://www.w3schools.com/java/java_arrays.asp) and [Classes](https://www.w3schools.com/java/java_classes.asp) (you will learn more about these in a later chapter)

## Primitive Data Types

A primitive data type specifies the size and type of variable values, and it has no additional methods.

There are eight primitive data types in Java:

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Size** | **Description** |
| byte | 1 byte | Stores whole numbers from -128 to 127 |
| short | 2 bytes | Stores whole numbers from -32,768 to 32,767 |
| int | 4 bytes | Stores whole numbers from -2,147,483,648 to 2,147,483,647 |
| long | 8 bytes | Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| float | 4 bytes | Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits |
| double | 8 bytes | Stores fractional numbers. Sufficient for storing 15 decimal digits |
| boolean | 1 bit | Stores true or false values |
| char | 2 bytes | Stores a single character/letter or ASCII values |

## Numbers

Primitive number types are divided into two groups:

**Integer types** stores whole numbers, positive or negative (such as 123 or -456), without decimals. Valid types are byte, short, int and long. Which type you should use, depends on the numeric value.

**Floating point types** represents numbers with a fractional part, containing one or more decimals. There are two types: float and double.

Note:

* Before using the long mention ”L” or “f” after the number
* Float “F” or “f”

### **Double**

The double data type can store fractional numbers from 1.7e−308 to 1.7e+308. Note that you should end the value with a "d":

### **Scientific Numbers**

A floating point number can also be a scientific number with an "e" to indicate the power of 10:

### **Example**

float f1 = 35e3f;

double d1 = 12E4d;

System.out.println(f1);

System.out.println(d1);

## Non-Primitive Data Types

Non-primitive data types are called **reference types** because they refer to objects.

The main difference between **primitive** and **non-primitive** data types are:

* Primitive types are predefined (already defined) in Java. Non-primitive types are created by the programmer and is not defined by Java (except for String).
* Non-primitive types can be used to call methods to perform certain operations, while primitive types cannot.
* A primitive type has always a value, while non-primitive types can be null.
* A primitive type starts with a lowercase letter, while non-primitive types starts with an uppercase letter.
* The size of a primitive type depends on the data type, while non-primitive types have all the same size.

Examples of non-primitive types are [Strings](https://www.w3schools.com/java/java_strings.asp), [Arrays](https://www.w3schools.com/java/java_arrays.asp), [Classes,](https://www.w3schools.com/java/java_classes.asp)[Interface](https://www.w3schools.com/java/java_interface.asp), etc. You will learn more about these in a later chapter.

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

## Narrowing Casting

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

### **Example**

public class MyClass {

public static void main(String[] args) {

double myDouble = 9.78;

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

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

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

}

}

## Java Operators

Operators are used to perform operations on variables and values.

Although the + operator is often used to add together two values, like in the example above, it can also be used to add together a variable and a value, or a variable and another variable:

Java divides the operators into the following groups:

* Arithmetic operators
* Assignment operators
* Comparison operators
* Logical operators
* Bitwise operators

## Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operator** | **Name** | **Description** | **Example** |  |
| + | Addition | Adds together two values | x + y |  |
| - | Subtraction | Subtracts one value from another | x - y |  |
| \* | Multiplication | Multiplies two values | x \* y |  |
| / | Division | Divides one value by another | x / y |  |
| % | Modulus | Returns the division remainder | x % y |  |
| ++ | Increment | Increases the value of a variable by 1 | ++x |  |
| -- | Decrement | Decreases the value of a variable by 1 | --x |  |

## Java Assignment Operators

Assignment operators are used to assign values to variables.

In the example below, we use the **assignment** operator (=) to assign the value **10** to a variable called **x**:

A list of all assignment operators:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Example** | **Same As** |  |
| = | x = 5 | x = 5 |  |
| += | x += 3 | x = x + 3 |  |
| -= | x -= 3 | x = x - 3 |  |
| \*= | x \*= 3 | x = x \* 3 |  |
| /= | x /= 3 | x = x / 3 |  |
| %= | x %= 3 | x = x % 3 |  |
| &= | x &= 3 | x = x & 3 |  |
| |= | x |= 3 | x = x | 3 |  |
| ^= | x ^= 3 | x = x ^ 3 |  |
| >>= | x >>= 3 | x = x >> 3 |  |
| <<= | x <<= 3 | x = x << 3 |  |

## Java Comparison Operators

Comparison operators are used to compare two values:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Name** | **Example** |  |
| == | Equal to | x == y |  |
| != | Not equal | x != y |  |
| > | Greater than | x > y |  |
| < | Less than | x < y |  |
| >= | Greater than or equal to | x >= y |  |
| <= | Less than or equal to | x <= y |  |

## Java Logical Operators

Logical operators are used to determine the logic between variables or values:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operator** | **Name** | **Description** | **Example** |  |
| && | Logical and | Returns true if both statements are true | x < 5 &&  x < 10 |  |
| || | Logical or | Returns true if one of the statements is true | x < 5 || x < 4 |  |
| ! | Logical not | Reverse the result, returns false if the result is true | !(x < 5 && x < 10) |  |

## Java Bitwise Operators

Bitwise operators are used to perform binary logic with the bits of an integer or long integer.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Operator** | **Description** | **Example** | **Same as** | **Result** | **Decimal** |
| & | AND - Sets each bit to 1 if both bits are 1 | 5 & 1 | 0101 & 0001 | 0001 | 1 |
| | | OR - Sets each bit to 1 if any of the two bits is 1 | 5 | 1 | 0101 | 0001 | 0101 | 5 |
| ~ | NOT - Inverts all the bits | ~ 5 | ~0101 | 1010 | 10 |
| ^ | XOR - Sets each bit to 1 if only one of the two bits is 1 | 5 ^ 1 | 0101 ^ 0001 | 0100 | 4 |
| << | Zero-fill left shift - Shift left by pushing zeroes in from the right and letting the leftmost bits fall off | 9 << 1 | 1001 << 1 | 0010 | 2 |
| >> | Signed right shift - Shift right by pushing copies of the leftmost bit in from the left and letting the rightmost bits fall off | 9 >> 1 | 1001 >> 1 | 1100 | 12 |
| >>> | Zero-fill right shift - Shift right by pushing zeroes in from the left and letting the rightmost bits fall off | 9 >>> 1 | 1001 >>> 1 | 0100 | 4 |

**Note:** The Bitwise examples above use 4-bit unsigned examples, but Java uses 32-bit signed integers and 64-bit signed long integers. Because of this, in Java, ~5 will not return 10. It will return -6. ~00000000000000000000000000000101 will return 11111111111111111111111111111010

In Java, 9 >> 1 will not return 12. It will return 4. 00000000000000000000000000001001 >> 1 will return 00000000000000000000000000000100