**Java tutorial**

**What is Java:**

Java is a **programming language** and a **platform**. Java is a high level, robust, object-oriented and secure programming language.

Java was developed by *Sun Microsystems* (which is now the subsidiary of Oracle) in the year 1995. *James Gosling* is known as the father of Java. Before Java, its name was *Oak*. Since Oak was already a registered company, so James Gosling and his team changed the name from Oak to Java.

**Why to use Java, it’s features:**

* 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

**About this tutorial:**

This tutorial is designed to make it easy for everyone to understand Java. This tutorial will include all significant topics about JAva and will help you to understand the concepts via examples.

Please make note that although tutorial does offer an insight to the language, it is not and extensive tutorial. An overview of the Java programminig languagee has been provided

**Table of contents**

Level 1:

1. Installing Java
2. Syntax
3. Variables
4. Data types
5. Operators
6. Arrays

Level 2:

1. Conditional Statements and Loops
2. Methods
3. Classes and Objects
4. Inheritance
5. Interfaces
6. Abstract classes

Level 3:

1. Packages
2. Exception Handling
3. Threads

**Level 1:**

1. **Installing Java**

Some PCs might have Java already installed.

To check if you have Java installed on a Windows PC, search in the start bar for Java or type the following in Command Prompt (cmd.exe):

C:\Users\*Your Name*>java -version

If Java is installed, you will see something like this (depending on version):

java version "11.0.1" 2018-10-16 LTS  
Java(TM) SE Runtime Environment 18.9 (build 11.0.1+13-LTS)  
Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.1+13-LTS, mixed mode)

If you do not have Java installed on your computer, you can download it for free at [oracle.com](https://www.oracle.com/technetwork/java/javase/overview/index.html).

1. **Syntax**

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

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

The name of the java file **must match** the class name. When saving the file, save it using the class name and add ".java" to the end of the filename

**Example:**

public class Main { public static void main(String[] args) { System.out.println("Hello World"); }}

The main() method is required and you will see it in every Java program:

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

Inside the main() method, we can use the println() method to print a line of text to the screen:

public static void main(String[] args) { **System.out.println("Hello World");**}

Comments In JAva:

Single-line comments start with two forward slashes (//).

Any text between // and the end of the line is ignored by Java (will not be executed).

Multi-line comments start with /\* and ends with \*/.

Any text between /\* and \*/ will be ignored by Java.

1. **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

To create a variable, you must specify the type and assign it a value:

### Syntax

*type variableName = value;*

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

The println() method is often used to display variables.

All Java **variables** must be **identified** with **unique names**.

These unique names are called **identifiers**.

1. **Data types**

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)

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** |
| byte | 1 byte |
| short | 2 bytes |
| int | 4 bytes |
| long | 8 bytes |
| float | 4 bytes |
| double | 8 bytes |
| boolean | 1 bit |
| char | 2 bytes |

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.

A boolean data type is declared with the boolean keyword and can only take the values true or false

The char data type is used to store a **single** character. The character must be surrounded by single quotes, like 'A' or 'c':

### Example

char myGrade = 'B';System.out.println(myGrade);

The String data type is used to store a sequence of characters (text). String values must be surrounded by double quotes:

### Example

String greeting = "Hello World";System.out.println(greeting);

1. **Operators**

Operators are used to perform operations on variables and values.

In the example below, we use the + **operator** to add together two values:

### Example

int x = 100 + 50;

Java divides the operators into the following groups:

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

**#Please insert the Table for operators as per your convenience**

1. **Arrays**

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type with **square brackets**:

String[] cars;

We have now declared a variable that holds an array of strings. To insert values to it, we can use an array literal - place the values in a comma-separated list, inside curly braces:

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

You access an array element by referring to the index number.

This statement accesses the value of the first element in cars:

### Example

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};System.out.println(cars[0]);// Outputs Volvo

A multidimensional array is an array of arrays.

To create a two-dimensional array, add each array within its own set of **curly braces**:

### Example

int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };

**Level 2:**

1. **Conditional Statements and Loops**

Java supports the usual logical conditions from mathematics:

* Less than: a < b
* Less than or equal to: a <= b
* Greater than: a > b
* Greater than or equal to: a >= b
* Equal to a == b
* Not Equal to: a != b

You can use these conditions to perform different actions for different decisions.

Java has the following conditional statements:

* Use if to specify a block of code to be executed, if a specified condition is true
* Use else to specify a block of code to be executed, if the same condition is false
* Use else if to specify a new condition to test, if the first condition is false
* Use switch to specify many alternative blocks of code to be executed

Use the if statement to specify a block of Java code to be executed if a condition is true.

### Syntax

if (*condition*) { *// block of code to be executed if the condition is true*}

Use the else statement to specify a block of code to be executed if the condition is false.

### Syntax

if (*condition*) { *// block of code to be executed if the condition is true*} else { *// block of code to be executed if the condition is false*}

Use the else if statement to specify a new condition if the first condition is false.

### Syntax

if (*condition1*) { *// block of code to be executed if condition1 is true*} else if (*condition2*) { *// block of code to be executed if the condition1 is false and condition2 is true*} else { *// block of code to be executed if the condition1 is false and condition2 is false*}

Use the switch statement to select one of many code blocks to be executed.

### Syntax

switch(*expression*) { case x: *// code block* break; case y: *// code block* break; default: *// code block*}

## Loops

Loops can execute a block of code as long as a specified condition is reached.

Loops are handy because they save time, reduce errors, and they make code more readable.

## Java While Loop

The while loop loops through a block of code as long as a specified condition is true:

### Syntax

while (*condition*) { *// code block to be executed*}

## The Do/While Loop

The do/while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

### Syntax

do { *// code block to be executed*}while (*condition*);

## Java For Loop

When you know exactly how many times you want to loop through a block of code, use the for loop instead of a while loop:

### Syntax

for (*statement 1*; *statement 2*; *statement 3*) { *// code block to be executed*}

he break statement can also be used to jump out of a **loop**.

This example stops the loop when i is equal to 4:

### Example

for (int i = 0; i < 10; i++) { if (i == 4) { break; } System.out.println(i);}

The continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

This example skips the value of 4:

### Example

for (int i = 0; i < 10; i++) { if (i == 4) { continue; } System.out.println(i);}

1. **Methods**

A **method** is a block of code which only runs when it is called.

You can pass data, known as parameters, into a method.

Methods are used to perform certain actions, and they are also known as **functions**.

Why use methods? To reuse code: define the code once, and use it many times.

A method must be declared within a class. It is defined with the name of the method, followed by parentheses **()**. Java provides some pre-defined methods, such as System.out.println(), but you can also create your own methods to perform certain actions:

### Example

Create a method inside Main:

public class Main { static void myMethod() { // code to be executed }}

## Call a Method

To call a method in Java, write the method's name followed by two parentheses **()** and a semicolon**;**

Information can be passed to methods as parameter. Parameters act as variables inside the method.

Parameters are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

With **method overloading**, multiple methods can have the same name with different parameters:

### Example

int myMethod(int x)float myMethod(float x)double myMethod(double x, double y)

1. **Classes and Objects**

A Class is like an object constructor, or a "blueprint" for creating objects.

## Create a Class

To create a class, use the keyword class:

### Main.java

Create a class named "Main" with a variable x:

public class Main { int x = 5;}

In Java, an object is created from a class. We have already created the class named Main, so now we can use this to create objects.

To create an object of Main, specify the class name, followed by the object name, and use the keyword new:

### Example

Create an object called "myObj" and print the value of x:

public class Main { int x = 5; public static void main(String[] args) { Main **myObj** = new Main(); System.out.println(myObj.x); }}

In the previous example, we used the term "variable" for x in the example (as shown below). It is actually an **attribute** of the class

You can access attributes by creating an object of the class, and by using the dot syntax (.)

Java classes can also hav methods, which can be static or non static

1. **Inheritance**

### Why use inheritance in java

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

## Types of inheritance in java

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

In java programming, multiple and hybrid inheritance is supported through interface only.



In Java, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

* **subclass** (child) - the class that inherits from another class
* **superclass** (parent) - the class being inherited from

To inherit from a class, use the extends keyword.

If you don't want other classes to inherit from a class, use the final keyword:

If you try to access a final class, Java will generate an error:

final class Vehicle { ...}class Car extends Vehicle { ...}

1. **Interfaces**

Another way to achieve [abstraction](https://www.w3schools.com/java/java_abstract.asp) in Java, is with interfaces.

An interface is a completely "**abstract class**" that is used to group related methods with empty bodies:

### Example

// interfaceinterface Animal { public void animalSound(); // interface method (does not have a body) public void run(); // interface method (does not have a body)}

To access the interface methods, the interface must be "implemented" (kinda like inherited) by another class with the implements keyword (instead of extends). The body of the interface method is provided by the "implement" class:

### Example

// Interfaceinterface Animal { public void animalSound(); // interface method (does not have a body) public void sleep(); // interface method (does not have a body)}// Pig "implements" the Animal interfaceclass Pig implements Animal { public void animalSound() { // The body of animalSound() is provided here System.out.println("The pig says: wee wee"); } public void sleep() { // The body of sleep() is provided here System.out.println("Zzz"); }}class Main { public static void main(String[] args) { Pig myPig = new Pig(); // Create a Pig object myPig.animalSound(); myPig.sleep(); }}

1. **Abstract classes**

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.  
Abstraction can be achieved with either **abstract classes** or [**interfaces**](https://www.w3schools.com/java/java_interface.asp) (which you will learn more about in the next chapter).

The abstract keyword is a non-access modifier, used for classes and methods:

* **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
* **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

An abstract class can have both abstract and regular methods:

abstract class Animal { public abstract void animalSound(); public void sleep() { System.out.println("Zzz"); }}

### Example

// Abstract classabstract class Animal { // Abstract method (does not have a body) public abstract void animalSound(); // Regular method public void sleep() { System.out.println("Zzz"); }}// Subclass (inherit from Animal)class Pig extends Animal { public void animalSound() { // The body of animalSound() is provided here System.out.println("The pig says: wee wee"); }}class Main { public static void main(String[] args) { Pig myPig = new Pig(); // Create a Pig object myPig.animalSound(); myPig.sleep(); }}

**Level 3:**

1. **Packages**

A package in Java is used to group related classes. Think of it as **a folder in a file directory**. We use packages to avoid name conflicts, and to write a better maintainable code. Packages are divided into two categories:

* Built-in Packages (packages from the Java API)
* User-defined Packages (create your own packages)

## Built-in Packages

The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment.

The library contains components for managing input, database programming, and much much more. The complete list can be found at Oracles website: <https://docs.oracle.com/javase/8/docs/api/>.

The library is divided into **packages** and **classes**. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.

To use a class or a package from the library, you need to use the import keyword:

### Syntax

import *package*.*name*.*Class*; // Import a single classimport *package*.*name*.\*; // Import the whole package

To create your own package, you need to understand that Java uses a file system directory to store them. Just like folders on your computer:

### Example

└── root └── mypack └── MyPackageClass.java

To create a package, use the package keyword:

### MyPackageClass.java

package mypack;class MyPackageClass { public static void main(String[] args) { System.out.println("This is my package!"); }}

Save the file as **MyPackageClass.java**, and compile it:

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

Then compile the package:

C:\Users\*Your Name*>javac -d . MyPackageClass.java

When we compiled the package in the example above, a new folder was created, called "mypack".

To run the **MyPackageClass.java** file, write the following:

C:\Users\*Your Name*>java mypack.MyPackageClass

The output will be:

This is my package!

**Access Modifiers:**

The public keyword is an **access modifier**, meaning that it is used to set the access level for classes, attributes, methods and constructors.

We divide modifiers into two groups:

* **Access Modifiers** - controls the access level
* **Non-Access Modifiers** - do not control access level, but provides other functionality

|  |  |
| --- | --- |
| **Modifier** | **Description** |
| public | The code is accessible for all classes |
| private | The code is only accessible within the declared class |
| *default* | The code is only accessible in the same package. This is used when you don't specify a modifier. You will learn more about packages in the [Packages chapter](https://www.w3schools.com/java/java_packages.asp) |
| protected | The code is accessible in the same package and **subclasses**. You will learn more about subclasses and superclasses in the [Inheritance chapter](https://www.w3schools.com/java/java_inheritance.asp) |

1. **Exception Handling**

When executing Java code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, Java will normally stop and generate an error message. The technical term for this is: Java will throw an **exception** (throw an error).

## Java try and catch

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

The try and catch keywords come in pairs:

### Syntax

try { // *Block of code to try*}catch(Exception *e*) { // *Block of code to handle errors*}

## Finally

The finally statement lets you execute code, after try...catch, regardless of the result:

### Example

public class Main { public static void main(String[] args) { try { int[] myNumbers = {1, 2, 3}; System.out.println(myNumbers[10]); } catch (Exception e) { System.out.println("Something went wrong."); } finally { System.out.println("The 'try catch' is finished."); } }}

1. **Threads**

Threads allows a program to operate more efficiently by doing multiple things at the same time.

Threads can be used to perform complicated tasks in the background without interrupting the main program.

## Creating a Thread

There are two ways to create a thread.

It can be created by extending the Thread class and overriding its run() method:

### Extend Syntax

public class Main extends Thread { public void run() { System.out.println("This code is running in a thread"); }}

Another way to create a thread is to implement the Runnable interface:

### Implement Syntax

public class Main implements Runnable { public void run() { System.out.println("This code is running in a thread"); }}

## Running Threads

If the class extends the Thread class, the thread can be run by creating an instance of the class and call its start() method:

### Extend Example

public class Main extends Thread { public static void main(String[] args) { Main thread = new Main(); thread.start(); System.out.println("This code is outside of the thread"); } public void run() { System.out.println("This code is running in a thread"); }}

If the class implements the Runnable interface, the thread can be run by passing an instance of the class to a Thread object's constructor and then calling the thread's start() method:

### Implement Example

public class Main implements Runnable { public static void main(String[] args) { Main obj = new Main(); Thread thread = new Thread(obj); thread.start(); System.out.println("This code is outside of the thread"); } public void run() { System.out.println("This code is running in a thread"); }}