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)

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

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

## Java Comments

Comments can be used to explain Java code, and to make it more readable. It can also be used to prevent execution when testing alternative code.

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).

This example uses a single-line comment before a line of code:

// This is a comment  
System.out.println("Hello World");

This example uses a single-line comment at the end of a line of code:

System.out.println("Hello World"); // This is a comment

## Java Multi-line Comments

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

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

This example uses a multi-line comment (a comment block) to explain the code:

/\* The code below will print the words Hello World  
to the screen, and it is amazing \*/  
System.out.println("Hello World");

## 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

## Declaring (Creating) Variables

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

*type* *variable* = *value*;

Create a variable called **name** of type String and assign it the value "**John**":

String name = "John";  
System.out.println(name);

Create a variable called **myNum** of type int and assign it the value **15**:

int myNum = 15;  
System.out.println(myNum);

You can also declare a variable without assigning the value, and assign the value later:

int myNum;  
myNum = 15;  
System.out.println(myNum);

A demonstration of how to declare variables of other types:

int myNum = 5;  
float myFloatNum = 5.99f;  
char myLetter = 'D';  
boolean myBool = true;  
String myText = "Hello";

## Display Variables

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

To combine both text and a variable, use the + character:

String firstName = "John ";  
String lastName = "Doe";  
String fullName = firstName + lastName;  
System.out.println(fullName);

## Java Identifiers

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

These unique names are called **identifiers**.

Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

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

* Names can contain letters, digits, underscores, and dollar signs
* Names should begin with a letter
* 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)
* Names should start with a lowercase letter and it cannot contain whitespace
* Reserved words (like Java keywords, such as int or String) cannot be used as names

## Java 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)

## 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:

A screenshot of a cell phone

Description automatically generated

## 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.

## Integer Types

### **Byte**

The byte data type can store whole numbers from -128 to 127. This can be used instead of int or other integer types to save memory when you are certain that the value will be within -128 and 127:

byte myNum = 100;  
System.out.println(myNum);

### **Short**

The short data type can store whole numbers from -32768 to 32767:

short myNum = 5000;  
System.out.println(myNum);

### **Long**

The long data type can store whole numbers from -9223372036854775808 to 9223372036854775807. This is used when int is not large enough to store the value. Note that you should end the value with an "L":

long myNum = 15000000000L;  
System.out.println(myNum);

## Booleans

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

boolean isJavaFun = true;  
boolean isFishTasty = false;  
System.out.println(isJavaFun);     // Outputs true  
System.out.println(isFishTasty);   // Outputs false

Alternatively, you can use ASCII values to display certain characters:

char a = 65, b = 66, c = 67;  
System.out.println(a);  
System.out.println(b);  
System.out.println(c);

## 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.

## 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 MyClass {  
  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 parantheses in front of the value:

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  
  }  
}

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

## Java Strings

Strings are used for storing text.

A String variable contains a collection of characters surrounded by double quotes:

String greeting = "Hello";

## String Length

A String in Java is actually an object, which contain methods that can perform certain operations on strings. For example, the length of a string can be found with the length() method:

String txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";  
System.out.println("The length of the txt string is: " + txt.length());

## More String Methods

There are many string methods available, for example toUpperCase() and toLowerCase():

String txt = "Hello World";  
System.out.println(txt.toUpperCase());   // Outputs "HELLO WORLD"  
System.out.println(txt.toLowerCase());   // Outputs "hello world"

## Finding a String in a String

The indexOf() method returns the **index** (the position) of the first occurrence of a specified text in a string (including whitespace):

String txt = "Please locate where 'locate' occurs!";  
System.out.println(txt.indexOf("locate")); // Outputs 7

## String Concatenation

The + operator can be used between strings to add them together to make a new string. This is called **concatenation**:

String firstName = "John";  
String lastName = "Doe";  
System.out.println(firstName + " " + lastName);

You can also use the concat() method to concatenate two strings:

String firstName = "John ";  
String lastName = "Doe";  
System.out.println(firstName.concat(lastName));

## Math.max(x,y)

The Math.max(x,y) method can be used to find the highest value of x and y:

Math.max(5, 10);

## Math.min(x,y)

The Math.min(x,y) method can be used to find the lowest value of of x and y:

Math.min(5, 10);

## Math.sqrt(x)

The Math.sqrt(x) method returns the square root of x:

Math.sqrt(64);

Math.abs(*x*)

The Math.abs(*x*) method returns the absolute (positive) value of *x*:

Math.abs(-4.7)

## Math.random()

Math.random() returns a random number between 0 (inclusive), and 1 (exclusive):

Math.random();

## Java Booleans

Very often, in programming, you will need a data type that can only have one of two values, like:

* YES / NO
* ON / OFF
* TRUE / FALSE

For this, Java has a boolean data type, which can take the values true or false.

Boolean Values

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

boolean isJavaFun = true;  
boolean isFishTasty = false;  
System.out.println(isJavaFun);     // Outputs true  
System.out.println(isFishTasty);   // Outputs false

## Boolean Expression

A **Boolean expression** is a Java expression that returns a Boolean value: true or false.

You can use a comparison operator, such as the **greater than** (>) operator to find out if an expression (or a variable) is true:

int x = 10;int y = 9;  
System.out.println(x > y); // returns true, because 10 is higher than 9

System.out.println(10 > 9); // returns true, because 10 is higher than 9

In the examples below, we use the **equal to** (==) operator to evaluate an expression:

int x = 10;  
System.out.println(x == 10); // returns true, because the value of x is equal to 10

System.out.println(10 == 15); // returns false, because 10 is not equal to 15

## Java Conditions and If Statements

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

## The else Statement

int time = 20;  
if (time < 18) {  
  System.out.println("Good day.");  
} else {  
  System.out.println("Good evening.");  
}  
// Outputs "Good evening."

## The else if Statement

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*  
}

### **Example**

int time = 22;  
if (time < 10) {  
  System.out.println("Good morning.");  
} else if (time < 20) {  
  System.out.println("Good day.");  
} else {  
  System.out.println("Good evening.");  
}  
// Outputs "Good evening."

## Java Switch Statements

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

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

This is how it works:

* The switch expression is evaluated once.
* The value of the expression is compared with the values of each case.
* If there is a match, the associated block of code is executed.
* The break and default keywords are optional, and will be described later in this chapter

### **Example**

int day = 4;  
switch (day) {  
  case 6:  
    System.out.println("Today is Saturday");  
    break;  
  case 7:  
    System.out.println("Today is Sunday");  
    break;  
  default:  
    System.out.println("Looking forward to the Weekend");  
}  
// Outputs "Looking forward to the Weekend"

Loops

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

Java While Loop

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

int i = 0;  
while (i < 5) {  
  System.out.println(i);  
  i++;  
}

## 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.

The example below uses a do/while loop. The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

int i = 0;  
do {  
  System.out.println(i);  
  i++;  
}  
while (i < 5);

## 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:

for (int i = 0; i < 5; i++) {  
  System.out.println(i);  
}

The following example outputs all elements in the **cars** array, using a "**for-each**" loop:

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};  
for (String i : cars) {  
  System.out.println(i);  
}

## Java Break

You have already seen the break statement used in an earlier chapter of this tutorial. It was used to "jump out" of a switch statement.

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

This example jumps out of the loop when i is equal to 4:

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

## Java Continue

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:

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

### **Continue Example Break**

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

## Java 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"};

To create an array of integers, you could write:

int[] myNum = {10, 20, 30, 40};

Access the Elements of an Array

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

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

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

Change an Array Element

To change the value of a specific element, refer to the index number:

cars[0] = "Opel";

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};  
cars[0] = "Opel";  
System.out.println(cars[0]);  
// Now outputs Opel instead of Volvo

## Array Length

To find out how many elements an array has, use the length property:

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};  
System.out.println(cars.length);  
// Outputs 4

Loop Through an Array

You can loop through the array elements with the for loop, and use the length property to specify how many times the loop should run.

The following example outputs all elements in the **cars** array:

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};  
for (int i = 0; i < cars.length; i++) {  
  System.out.println(cars[i]);  
}

## Loop Through an Array with For-Each

There is also a "**for-each**" loop, which is used exclusively to loop through elements in arrays:

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};  
for (String i : cars) {  
  System.out.println(i);  
}

The example above can be read like this: **for each** String element (called **i** - as in **i**ndex) in **cars**, print out the value of**i**.

If you compare the for loop and **for-each** loop, you will see that the **for-each** method is easier to write, it does not require a counter (using the length property), and it is more readable.

## Multidimensional Arrays

A multidimensional array is an array containing one or more arrays.

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

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

**myNumbers** is now an array with two arrays as its elements.

To access the elements of the **myNumbers** array, specify two indexes: one for the array, and one for the element inside that array. This example accesses the third element (2) in the second array (1) of myNumbers:

int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };  
int x = myNumbers[1][2];  
System.out.println(x); // Outputs 7

We can also use a for loop inside another for loop to get the elements of a two-dimensional array (we still have to point to the two indexes):

public class MyClass {  
  public static void main(String[] args) {  
    int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };  
    for (int i = 0; i < myNumbers.length; ++i) {  
      for(int j = 0; j < myNumbers[i].length; ++j) {  
        System.out.println(myNumbers[i][j]);  
      }  
    }  
  }  
}

# **Java 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.

Create a Method

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:

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

#### **Example Explained**

* myMethod() is the name of the method
* static means that the method belongs to the MyClass class and not an object of the MyClass class. You will learn more about objects and how to access methods through objects later in this tutorial.
* void means that this method does not have a return value. You will learn more about return values later in this chapter

Call a Method

* To call a method in Java, write the method's name followed by two parentheses **()** and a semicolon**;**
* In the following example, myMethod() is used to print a text (the action), when it is called:

Inside main, call the myMethod() method:

public class MyClass {  
  static void myMethod() {  
    System.out.println("I just got executed!");  
  }  
  
  public static void main(String[] args) {  
    **myMethod();**  
  }  
}  
  
// Outputs "I just got executed!"

### **Example**

public class MyClass {  
  static void myMethod() {  
    System.out.println("I just got executed!");  
  }  
  
  public static void main(String[] args) {  
    **myMethod();**  
    **myMethod();**  
    **myMethod();**  
  }  
}  
// I just got executed!  
// I just got executed!

## Method Parameters

Information can be passed to functions 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.

The following example has a method that takes a String called **fname** as parameter. When the method is called, we pass along a first name, which is used inside the method to print the full name:

public class MyClass {  
  static void myMethod(**String fname**) {  
    System.out.println(fname + " Refsnes");  
  }  
  
  public static void main(String[] args) {  
    myMethod("Liam");  
    myMethod("Jenny");  
    myMethod("Anja");  
  }  
}  
// Liam Refsnes  
// Jenny Refsnes  
// Anja Refsnes

## Return Values

The void keyword, used in the examples above, indicates that the method should not return a value. If you want the method to return a value, you can use a primitive data type (such as int, char, etc.) instead of void, and use the return keyword inside the method:

public class MyClass {  
  static **int** myMethod(int x) {  
    **return** 5 + x;  
  }  
  public static void main(String[] args) {  
    System.out.println(myMethod(3));  
  }  
}  
// Outputs 8 (5 + 3)

This example returns the sum of a method's **two parameters**:

public class MyClass {  
  static int myMethod(int x, int y) {  
    return x + y;  
  }  
  
  public static void main(String[] args) {  
    System.out.println(myMethod(5, 3));  
  }  
}  
// Outputs 8 (5 + 3)

You can also store the result in a variable (recommended):

public class MyClass {  
  static int myMethod(int x, int y) {  
    return x + y;  
  }  
  
  public static void main(String[] args) {  
    int z = myMethod(5, 3);  
    System.out.println(z);  
  }  
}  
// Outputs 8 (5 + 3)

## A Method with If...Else

It is common to use if...else statements inside methods:

public class MyClass {  
  
  // Create a checkAge() method with an integer variable called **age**  
  static void checkAge(int age) {  
  
    // If age is less than 18, print "access denied"  
    if (age < 18) {  
      System.out.println("Access denied - You are not old enough!");   
  
    // If age is greater than 18, print "access granted"  
    } else {  
      System.out.println("Access granted - You are old enough!");   
    }  
  
  }   
  
  public static void main(String[] args) {   
    checkAge(20); // Call the checkAge method and pass along an age of 20  
  }   
}  
  
// Outputs "Access granted - You are old enough!"