



BASIC JAVA CRASH COURSE

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

Why Use Java?

- It is one of the most popular programming language in the world . It has a huge community support (tens of millions of developers)
- It is open-source, free, secure, fast , portable and powerful
- Java is an object oriented language which gives a clear structure to programs and allows code to be reused, lowering development costs.

```
public class HelloWorld{  
    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.
- A class should always start with an uppercase first letter.

- Every line of code that runs in Java must be inside a class.

Single-line Comments

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

Java Multi-line Comments

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

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;`

```
int myNum = 5;
float myFloatNum = 5.99f;
char myLetter = 'D';
boolean myBool = true;
String myText = "Hello"
final int myNum = 15;
```

Final Variables




However, you can add the `final` keyword if you don't want to overwrite existing values declare the variable as "final" or "constant".

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:

Primitive Data Types

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

Scientific Numbers

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

```
float f1 = 35e3f;  
double d1 = 12E4d;
```

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, Arrays, Classes, Interface, 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`

Widening Casting

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

```
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

```
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

Operators are used to perform operations on variables and values.

Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.

Arithmetic Operators

Operator	Name	Description	Example
----------	------	-------------	---------

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

JAVA assignment operator

A list of all assignment operators:

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

Java Comparison Operators

Comparison operators are used to compare two values:

Java Comparison Operators

Operator	Name	Example
----------	------	---------

Operator	Name	Example
<code>==</code>	Equal to	<code>x == y</code>
<code>></code>	Greater than	<code>x > y</code>
<code><</code>	Less than	<code>x < y</code>
<code>>=</code>	Greater than or equal to	<code>x >= y</code>
<code><=</code>	Less than or equal to	<code>x <= y</code>
<code>!=</code>	Not equal	<code>x != y</code>

Java Logical Operators

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

Java Logical Operators




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

Java Strings




- Strings are used for storing text. A String variable contains a collection of characters surrounded by double quotes:
- the length of a string can be found with the `length()` method
- `toUpperCase()` and `toLowerCase()`
- The `indexOf()` method returns the index (the position) of the first occurrence of a specified text in a string (including whitespace).
- The `+` operator or the `concat()` method can be used between strings to combine them. This is called **concatenation**

-

Special Characters

 Escape character	 Result	 Description
<code>\'</code>	'	Single quote
<code>\"</code>	"	Double quote
<code>\\</code>	\	Backslash
<code>\n</code>		New Line
<code>\r</code>		Carriage Return
<code>\t</code>		Tab
<code>\b</code>		Backspace
<code>\f</code>		Form Feed

All String Methods

 Method	 Description	 Return Type
<code>charAt()</code>	<u>Returns the character at the specified index (position).</u>	char
<code>codePointAt()</code>	<u>Returns the Unicode of the character at the specified index</u>	int
<code>codePointBefore()</code>	<u>Returns the Unicode of the character before the specified index</u>	int
<code>codePointCount()</code>	<u>Returns the Unicode in the specified text range of this String</u>	int
<code>compareTo()</code>	<u>Compares two strings lexicographically.</u>	int
<code>compareToIgnoreCase()</code>	<u>Compares two strings lexicographically, ignoring case differences</u>	int
<code>concat()</code>	<u>Appends a string to the end of another string</u>	String
<code>contains()</code>	<u>Checks whether a string contains a sequence of characters</u>	boolean
<code>contentEquals()</code>	<u>Checks whether a string contains the exact same sequence of characters of the specified CharSequence or StringBuffer</u>	boolean
<code>copyValueOf()</code>	<u>Returns a String that represents the characters of the character array.</u>	String

 Method	 Description	 Return Type
<u>endsWith()</u>	<u>Checks whether a string ends with the specified character(s).</u>	boolean
<u>equals()</u>	<u>Compares two strings. Returns true if the strings are equal, and false if not</u>	boolean
<u>equalsIgnoreCase()</u>	<u>Compares two strings, ignoring case considerations</u>	boolean
<u>format()</u>	<u>Returns a formatted string using the specified locale, format string, and arguments</u>	String
<u>getBytes()</u>	<u>Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.</u>	byte[]
<u>getChars()</u>	<u>Copies characters from a string to an array of chars</u>	void
<u>hashCode()</u>	<u>Returns the hash code of a string</u>	int
<u>indexOf()</u>	<u>Returns the position of the first found occurrence of specified characters in a string</u>	int
<u>intern()</u>	<u>Returns the canonical representation for the string object</u>	String
<u>isEmpty()</u>	<u>Checks whether a string is empty or not</u>	boolean
<u>lastIndexOf()</u>	<u>Returns the position of the last found occurrence of specified characters in a string</u>	int
<u>length()</u>	<u>Returns the length of a specified string</u>	int
<u>matches()</u>	<u>Searches a string for a match against a regular expression, and returns the matches</u>	boolean
<u>offsetByCodePoints()</u>	<u>Returns the index within this String that is offset from the given index by codePointOffset code points</u>	int
<u>regionMatches()</u>	<u>Tests if two string regions are equal</u>	boolean
<u>replace()</u>	<u>Searches a string for a specified value, and returns a new string where the specified values are replaced</u>	String
<u>replaceFirst()</u>	<u>Replaces the first occurrence of a substring that matches the given regular expression with the given replacement</u>	String

☰ Method	Aa Description	☰ Return Type
replaceAll()	<u>Replaces each substring of this string that matches the given regular expression with the given replacement</u>	String
split()	<u>Splits a string into an array of substrings</u>	String[]
<u>startsWith().</u>	<u>Checks whether a string starts with specified characters</u>	boolean
subSequence()	<u>Returns a new character sequence that is a subsequence of this sequence</u>	CharSequence
substring()	<u>Returns a new string which is the substring of a specified string</u>	String
toCharArray()	<u>Converts this string to a new character array.</u>	char[]
<u>toLowerCase().</u>	<u>Converts a string to lower case letters</u>	String
toString()	<u>Returns the value of a String object</u>	String
<u>toUpperCase().</u>	<u>Converts a string to upper case letters</u>	String
<u>trim().</u>	<u>Removes whitespace from both ends of a string</u>	String
valueOf()	<u>Returns the string representation of the specified value</u>	String

Java Math

```
Math.max(5, 10);
Math.min(5, 10);
Math.sqrt(64);
Math.abs(-4.7);
Math.random();
```

Java Conditions and If Statements

Java supports the usual logical conditions from mathematics:

- Less than: $a < b$
- Less than or equal to: $a \leq b$
- Greater than: $a > b$

- Greater than or equal to: `a >= b`
- Equal to `a == b`
- Not Equal to: `a != b`

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

```
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
}

//shorthand
variable = (condition) ? expressionTrue : expressionFalse;
```

```
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.

While Loop

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

```
// While loop
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.

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

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 ( variable ; condition; increment) {
    // code block to be executed
}
```

For-Each Loop

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

```
for (type variableName : arrayName) {
    // code block to be executed
}
```

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

- The **break** statement can also be used to jump out of a loop.
- The **continue** statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop

Java Arrays

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

```
String[] cars;

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"}; // array of strings

int[] myNum = {10, 20, 30, 40}; // array of integers

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

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 should one 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 Main {  
    static void myMethod() {  
        // code to be executed  
    }  
}
```

- `myMethod()` is the name of the method
- `static` means that the method belongs to the Main class and not an object of the Main 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;

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

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

```

public class Main {
    static void myMethod(String fname, int age) {
        System.out.println(fname + " is " + age);
    }

    public static void main(String[] args) {
        myMethod("Liam", 5);
        myMethod("Jenny", 8);
        myMethod("Anja", 31);
    }
}

```

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 Main {
    static int myMethod(int x) {
        return 5 + x;
    }

    public static void main(String[] args) {
        System.out.println(myMethod(3));
    }
}

```

Method Overloading

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

```

static int plusMethodInt(int x, int y) {
    return x + y;
}

static double plusMethodDouble(double x, double y) {
    return x + y;
}

public static void main(String[] args) {
    int myNum1 = plusMethodInt(8, 5);
    double myNum2 = plusMethodDouble(4.3, 6.26);
}

```

```
System.out.println("int: " + myNum1);
System.out.println("double: " + myNum2);
}
```

Java Scope

In Java, variables are only accessible inside the region they are created. This is called **scope**.

- Method Scope
- Block Scope


Java Recursion

Recursion is the technique of making a function call itself. This technique provides a way to break complicated problems down into simple problems which are easier to solve.

Java Dates

Java does not have a built-in Date class, but we can import the `java.time` package to work with the date and time API. The package includes many date and time classes. For example:

Copy of java.time

 Class	<u>Aa</u> Description
<code>LocalDate</code>	<u>Represents a date (year, month, day (yyyy-MM-dd)).</u>
<code>LocalTime</code>	<u>Represents a time (hour, minute, second and nanoseconds (HH-mm-ss-ns)).</u>
<code>LocalDateTime</code>	<u>Represents both a date and a time (yyyy-MM-dd-HH-mm-ss-ns).</u>
<code>DateTimeFormatter</code>	<u>Formatter for displaying and parsing date-time objects</u>

```
import java.time.LocalDate; // import the LocalDate class

public class Main {
    public static void main(String[] args) {
```



```

    LocalDate myObj = LocalDate.now(); // Create a date object
    System.out.println(myObj); // Display the current date
  }
}

```

```

import java.time.LocalTime; // import the LocalTime class

public class Main {
    public static void main(String[] args) {
        LocalTime myObj = LocalTime.now();
        System.out.println(myObj);
    }
}

```

```

import java.time.LocalDateTime; // import the LocalDateTime class

public class Main {
    public static void main(String[] args) {
        LocalDateTime myObj = LocalDateTime.now();
        System.out.println(myObj);
    }
}

```

Formatting Date and Time

You can use the `DateTimeFormatter` class with the `ofPattern()` method in the same package to format or parse date-time objects.

```

import java.time.LocalDateTime; // Import the LocalDateTime class
import java.time.format.DateTimeFormatter; // Import the DateTimeFormatter class

public class Main {
    public static void main(String[] args) {
        LocalDateTime myDateObj = LocalDateTime.now();
        System.out.println("Before formatting: " + myDateObj);
        DateTimeFormatter myFormatObj = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");

        String formattedDate = myDateObj.format(myFormatObj);
        System.out.println("After formatting: " + formattedDate);
    }
}

```