Java Tutorials

Why Use Java?

* Java works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.)
* It is one of the most popular programming languages in the world
* It has a large demand in the current job market
* It is easy to learn and simple to use
* It is open-source and free
* It is secure, fast and powerful
* It has 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.

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

## 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

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

C:\Users\*Your Name*>javac Main.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 Main" to run the file:

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

The output should read:

Hello World

## System.out.println()

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

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

System is a built-in Java class that contains useful members, such as out, which is short for "output". The println() method, short for "print line", is used to print a value to the screen (or a file).

Don't worry too much about System, out and println(). Just know that you need them together to print stuff to the screen.

## 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

Note that if you assign a new value to an existing variable, it will overwrite the previous value:

int myNum = 15;

myNum = 20; // myNum is now 20

System.out.println(myNum);

## Final Variables

If you don't want others (or yourself) to overwrite existing values, use the final keyword (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

## Other Types

A demonstration of how to declare variables of other types:

### **Example**

int myNum = 5;

float myFloatNum = 5.99f;

char myLetter = 'D';

boolean myBool = true;

String myText = "Hello";

You can also use the + character to add a variable to another variable:

String firstName = "John ";

String lastName = "Doe";

String fullName = firstName + lastName;

## System.out.println(fullName);

## Declare Many Variables

To declare more than one variable of the **same type**, you can use a comma-separated list:

### **Example**[**Get your own Java Server**](https://www.w3schools.com/java/java_server.asp)

Instead of writing:

int x = 5;

int y = 6;

int z = 50;

System.out.println(x + y + z);

You can simply write:

int x = 5, y = 6, z = 50;

System.out.println(x + y + z);

## One Value to Multiple Variables

You can also assign the **same value** to multiple variables in one line:

### **Example**

int x, y, z;

x = y = z = 50;

System.out.println(x + y + z);

The general rules for naming variables are:

* Names can contain letters, digits, underscores, and dollar signs
* Names must begin with a letter
* Names should start with a lowercase letter, and 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

## 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:

|  |  |  |
| --- | --- | --- |
| **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.

Even though there are many numeric types in Java, the most used for numbers are int (for whole numbers) and double (for floating point numbers). However, we will describe them all as you continue to read.

## 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:

### **Example**[**Get your own Java Server**](https://www.w3schools.com/java/java_server.asp)

byte myNum = 100;

System.out.println(myNum);

[Try it Yourself »](https://www.w3schools.com/java/tryjava.asp?filename=demo_type_byte)

### **Short**

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

### **Example**

short myNum = 5000;

System.out.println(myNum);

[Try it Yourself »](https://www.w3schools.com/java/tryjava.asp?filename=demo_type_short)

### **Int**

The int data type can store whole numbers from -2147483648 to 2147483647. In general, and in our tutorial, the int data type is the preferred data type when we create variables with a numeric value.

### **Example**

int myNum = 100000;

System.out.println(myNum);

[Try it Yourself »](https://www.w3schools.com/java/tryjava.asp?filename=demo_type_int)

### **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":

### **Example**

long myNum = 15000000000L;

System.out.println(myNum);

[Try it Yourself »](https://www.w3schools.com/java/tryjava.asp?filename=demo_type_long)

## Floating Point Types

You should use a floating point type whenever you need a number with a decimal, such as 9.99 or 3.14515.

The float and double data types can store fractional numbers. Note that you should end the value with an "f" for floats and "d" for doubles:

### **Float Example**

float myNum = 5.75f;

System.out.println(myNum);

[Try it Yourself »](https://www.w3schools.com/java/tryjava.asp?filename=demo_type_float)

### **Double Example**

double myNum = 19.99d;

System.out.println(myNum);

[Try it Yourself »](https://www.w3schools.com/java/tryjava.asp?filename=demo_type_double)

Use float or double?

The **precision** of a floating point value indicates how many digits the value can have after the decimal point. The precision of float is only six or seven decimal digits, while double variables have a precision of about 15 digits. Therefore it is safer to use double for most calculations.

### **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);

Char

Alternatively, if you are familiar with ASCII values, you can use those to display certain characters:

### **Example**

char myVar1 = 65, myVar2 = 66, myVar3 = 67;

System.out.println(myVar1);

System.out.println(myVar2);

System.out.println(myVar3);

## Strings

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);

The String type is so much used and integrated in Java, that some call it "the special **ninth** type".

A String in Java is actually a **non-primitive** data type, because it refers to an object. The String object has methods that are used to perform certain operations on strings. **Don't worry if you don't understand the term "object" just yet**. We will learn more about strings and objects in a later chapter.

## 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.

# **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:

### **Example**

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**

Operators are used to perform operations on variables and values.

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** | **Try it** |
| + | Addition | Adds together two values | x + y | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_add) |
| - | Subtraction | Subtracts one value from another | x - y | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_sub) |
| \* | Multiplication | Multiplies two values | x \* y | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_mult) |
| / | Division | Divides one value by another | x / y | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_div) |
| % | Modulus | Returns the division remainder | x % y | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_mod) |
| ++ | Increment | Increases the value of a variable by 1 | ++x | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_inc) |
| -- | Decrement | Decreases the value of a variable by 1 | --x | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_dec) |

## Java Assignment Operators

Assignment operators are used to assign values to variables.

A list of all assignment operators:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Example** | **Same As** | **Try it** |
| = | x = 5 | x = 5 | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_ass1) |
| += | x += 3 | x = x + 3 | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_ass2) |
| -= | x -= 3 | x = x - 3 | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_ass3) |
| \*= | x \*= 3 | x = x \* 3 | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_ass4) |
| /= | x /= 3 | x = x / 3 | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_ass5) |
| %= | x %= 3 | x = x % 3 | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_ass6) |
| &= | x &= 3 | x = x & 3 | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_ass7) |
| |= | x |= 3 | x = x | 3 | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_ass8) |
| ^= | x ^= 3 | x = x ^ 3 | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_ass9) |
| >>= | x >>= 3 | x = x >> 3 | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_ass10) |
| <<= | x <<= 3 | x = x << 3 |  |

## Java Comparison Operators

Comparison operators are used to compare two values (or variables). This is important in programming, because it helps us to find answers and make decisions.

The return value of a comparison is either true or false. These values are known as Boolean values, and you will learn more about them in the [Booleans](https://www.w3schools.com/java/java_booleans.asp) and [If..Else](https://www.w3schools.com/java/java_conditions.asp) chapter.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Name** | **Example** | **Try it** |
| == | Equal to | x == y | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_compare1) |
| != | Not equal | x != y | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_compare2) |
| > | Greater than | x > y | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_compare3) |
| < | Less than | x < y | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_compare4) |
| >= | Greater than or equal to | x >= y | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_compare5) |
| <= | Less than or equal to | x <= y |  |

## Java Logical Operators

You can also test for true or false values with logical operators.

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operator** | **Name** | **Description** | **Example** | **Try it** |
| && | Logical and | Returns true if both statements are true | x < 5 &&  x < 10 | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_logical1) |
| || | Logical or | Returns true if one of the statements is true | x < 5 || x < 4 | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_oper_logical2) |
| ! | Logical not | Reverse the result, returns false if the result is true | !(x < 5 && x < 10) |  |

## Java Strings

Strings are used for storing text.

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

## 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():

### **Example**

String txt = "Hello World";

System.out.println(txt.toUpperCase()); // Outputs "HELLO WORLD"

System.out.println(txt.toLowerCase()); // Outputs "hello world"

## Finding a Character in a String

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

### **Example**

String txt = "Please locate where 'locate' occurs!";

System.out.println(txt.indexOf("locate")); // Outputs 7

Java counts positions from zero.  
0 is the first position in a string, 1 is the second, 2 is the third ...

## String Concatenation

The + operator can be used between strings to combine them. This is called **concatenation**:

String firstName = "John";

String lastName = "Doe";

System.out.println(firstName + " " + lastName);

Note that we have added an empty text (" ") to create a space between firstName and lastName on print.

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

String firstName = "John ";

String lastName = "Doe";

System.out.println(firstName.concat(lastName));

## Adding Numbers and Strings

WARNING!

Java uses the + operator for both addition and concatenation.

Numbers are added. Strings are concatenated.

If you add two numbers, the result will be a number:

### **Example**

int x = 10;

int y = 20;

int z = x + y; // z will be 30 (an integer/number)

If you add two strings, the result will be a string concatenation:

### **Example**

String x = "10";

String y = "20";

String z = x + y; // z will be 1020 (a String)

If you add a number and a string, the result will be a string concatenation:

### **Example**

String x = "10";

int y = 20;

String z = x + y; // z will be 1020 (a String)

# **Java Special Characters**

## Strings - Special Characters

Because strings must be written within quotes, Java will misunderstand this string, and generate an error:

String txt = "We are the so-called "Vikings" from the north.";

The solution to avoid this problem, is to use the **backslash escape character**.

The backslash (\) escape character turns special characters into string characters:

|  |  |  |
| --- | --- | --- |
| **Escape character** | **Result** | **Description** |
| \' | ' | Single quote |
| \" | " | Double quote |
| \\ | \ | Backslash |

The sequence \"  inserts a double quote in a string:

### **Example**

String txt = "We are the so-called \"Vikings\" from the north.";

The sequence \'  inserts a single quote in a string:

### **Example**

String txt = "It\'s alright.";

The sequence \\  inserts a single backslash in a string:

### **Example**

String txt = "The character \\ is called backslash.";

Other common escape sequences that are valid in Java are:

|  |  |  |
| --- | --- | --- |
| **Code** | **Result** | **Try it** |
| \n | New Line | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_strings_newline) |
| \r | Carriage Return | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_strings_r) |
| \t | Tab | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_strings_tab) |
| \b | Backspace | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_strings_b) |
| \f | Form Feed |  |

# **Java Math**

The Java Math class has many methods that allows you to perform mathematical tasks on numbers.

## 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 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);

## Random Numbers

Math.random() returns a random number between 0.0 (inclusive), and 1.0 (exclusive):

Math.random();

To get more control over the random number, for example, if you only want a random number between 0 and 100, you can use the following formula:

int randomNum = (int)(Math.random() \* 101); // 0 to 100

## 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 store true or false values.

boolean isJavaFun = true;

boolean isFishTasty = false;

System.out.println(isJavaFun); // Outputs true

System.out.println(isFishTasty); // Outputs false

int x = 10;

int y = 9;

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

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

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

Let's think of a "real life example" where we need to find out if a person is old enough to vote.

In the example below, we use the >= comparison operator to find out if the age (25) is **greater than** OR **equal to** the voting age limit, which is set to 18:

int myAge = 25;

int votingAge = 18;

System.out.println(myAge >= votingAge);

# **Java If ... Else**

## Java Conditions and If Statements

You already know that 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 if Statement

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

if (condition) {

// block of code to be executed if the condition is true

}

Note that if is in lowercase letters. Uppercase letters (If or IF) will generate an error.

if (20 > 18) {

System.out.println("20 is greater than 18");

}

int x = 20;

int y = 18;

if (x > y) {

System.out.println("x is greater than y");

}

## The else Statement

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

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

}

int time = 20;

if (time < 18) {

System.out.println("Good day.");

} else {

System.out.println("Good evening.");

}

// Outputs "Good evening."

In the example above, time (20) is greater than 18, so the condition is false. Because of this, we move on to the else condition and print to the screen "Good evening". If the time was less than 18, the program would print "Good day".

int time = 22;

if (time < 10) {

System.out.println("Good morning.");

} else if (time < 18) {

System.out.println("Good day.");

} else {

System.out.println("Good evening.");

}

// Outputs "Good evening."In the example above, time (22) is greater than 10, so the **first condition** is false. The next condition, in the else

#### **Example explained**

if statement, is also false, so we move on to the else condition since **condition1** and **condition2** is both false - and print to the screen "Good evening".

However, if the time was 14, our program would print "Good day."

**ternary operator:** It can be used to replace multiple lines of code with a single line, and is most often used to replace simple if else statements:

**Syntax** variable *= (*condition*) ?* expressionTrue *:*  expressionFalse*;*

Instead of writing: int time = 20;

if (time < 18) {

System.out.println("Good day.");

} else {

System.out.println("Good evening.");

}

You can simply write: int time = 20;

String result = (time < 18) ? "Good day." : "Good evening.";

System.out.println(result);

## Java Switch Statements

Instead of writing **many** if..else statements, you can use the switch statement.

The switch statement selects 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

The example below uses the weekday number to calculate the weekday name:

int day = 4;

switch (day) {

case 1:

System.out.println("Monday");

break;

case 2:

System.out.println("Tuesday");

break;

case 3:

System.out.println("Wednesday");

break;

case 4:

System.out.println("Thursday");

break;

case 5:

System.out.println("Friday");

break;

case 6:

System.out.println("Saturday");

break;

case 7:

System.out.println("Sunday");

break;

}

// Outputs "Thursday" (day 4)

## The break Keyword

When Java reaches a break keyword, it breaks out of the switch block.

This will stop the execution of more code and case testing inside the block.

When a match is found, and the job is done, it's time for a break. There is no need for more testing.

A break can save a lot of execution time because it "ignores" the execution of all the rest of the code in the switch block.

## The default Keyword

The default keyword specifies some code to run if there is no case match:

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"

Note that if the default statement is used as the last statement in a switch block, it does not need a break.

## 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:

while (condition) {

*// code block to be executed*

}

int i = 0;

while (i < 5) {

System.out.println(i);

i++;

}**Note:** Do not forget to increase the variable used in the condition, otherwise the loop will never end!

## 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);

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 (*statement 1*; *statement 2*; *statement 3*) {

*// code block to be executed*

}

**Statement 1** is executed (one time) before the execution of the code block.

**Statement 2** defines the condition for executing the code block.

**Statement 3** is executed (every time) after the code block has been executed.

The example below will print the numbers 0 to 4:

for (int i = 0; i < 5; i++) {

System.out.println(i);

}

#### **Example explained**

Statement 1 sets a variable before the loop starts (int i = 0).

Statement 2 defines the condition for the loop to run (i must be less than 5). If the condition is true, the loop will start over again, if it is false, the loop will end.

Statement 3 increases a value (i++) each time the code block in the loop has been executed.

## Nested Loops

It is also possible to place a loop inside another loop. This is called a **nested loop**.

The "inner loop" will be executed one time for each iteration of the "outer loop":

for (int i = 1; i <= 2; i++) {

System.out.println("Outer: " + i); // Executes 2 times

// Inner loop

for (int j = 1; j <= 3; j++) {

System.out.println(" Inner: " + j); // Executes 6 times (2 \* 3)

}

}

## For-Each Loop

There is also a "**for-each**" loop, which is used exclusively to loop through elements in an [**array**](https://www.w3schools.com/java/java_arrays.asp):

### **Syntax**

for (type variableName : arrayName) {

*// code block to be executed*

}

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

for (String i : cars) {

System.out.println(i);

}

## Java Break

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

This example stops 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);

}

## Break and Continue in While Loop

You can also use break and continue in while loops:

int i = 0;

while (i < 10) {

System.out.println(i);

i++;

if (i == 4) {

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

## Access the Elements of an Array

You can 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

**Note:** Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

## 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

## Multidimensional Arrays

A multidimensional array is an array of arrays.

Multidimensional arrays are useful when you want to store data as a tabular form, like a table with rows and columns.

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} };

## Access 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} };

System.out.println(myNumbers[1][2]); // Outputs 7

## Change Element Values

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

myNumbers[1][2] = 9;

System.out.println(myNumbers[1][2]); // Outputs 9 instead of 7

## Loop Through a Multi-Dimensional Array

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 Main {

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:

Create a method inside Main:

public class Main {

static void myMethod() {

// code to be executed

}

}

#### **Example Explained**

* 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**;**

Inside main, call the myMethod() method:

public class Main {

static void myMethod() {

System.out.println("I just got executed!");

}

public static void main(String[] args) {

myMethod();

}

}

## Parameters and Arguments

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.

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 Main {

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

When a **parameter** is passed to the method, it is called an **argument**. So, from the example above: fname is a **parameter**, while Liam, Jenny and Anja are **arguments**.

## Multiple Parameters

You can have as many parameters as you like: 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);

}

}

// Liam is 5

// Jenny is 8

// Anja is 31

Note that when you are working with multiple parameters, the method call must have the same number of arguments as there are parameters, and the arguments must be passed in the same order.

## 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));

}

}

// Outputs 8 (5 + 3)

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

public class Main {

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, as it is easier to read and maintain):

public class Main {

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 Main {

// 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, or equal to, 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!"

# **Java Method Overloading**

## Method Overloading

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

int myMethod(int x)

float myMethod(float x)

double myMethod(double x, double y)

Consider the following example, which has two methods that add numbers of different type:

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);

}

Instead of defining two methods that should do the same thing, it is better to overload one.

In the example below, we overload the plusMethod method to work for both int and double:

static int plusMethod(int x, int y) {

return x + y;

}

static double plusMethod(double x, double y) {

return x + y;

}

public static void main(String[] args) {

int myNum1 = plusMethod(8, 5);

double myNum2 = plusMethod(4.3, 6.26);

System.out.println("int: " + myNum1);

System.out.println("double: " + myNum2);

}

**Note:** Multiple methods can have the same name as long as the number and/or type of parameters are different.

## Java Scope

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

## Method Scope

Variables declared directly inside a method are available anywhere in the method following the line of code in which they were declared:

public class Main {

public static void main(String[] args) {

// Code here CANNOT use x

int x = 100;

// Code here can use x

System.out.println(x);

}

}

## Block Scope

A block of code refers to all of the code between curly braces {}.

Variables declared inside blocks of code are only accessible by the code between the curly braces, which follows the line in which the variable was declared:

public class Main {

public static void main(String[] args) {

// Code here CANNOT use x

{ // This is a block

// Code here CANNOT use x

int x = 100;

// Code here CAN use x

System.out.println(x);

} // The block ends here

// Code here CANNOT use x

}

}

A block of code may exist on its own or it can belong to an if, while or for statement. In the case of for statements, variables declared in the statement itself are also available inside the block's 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.

Recursion may be a bit difficult to understand. The best way to figure out how it works is to experiment with it.

## Recursion Example

Adding two numbers together is easy to do, but adding a range of numbers is more complicated. In the following example, recursion is used to add a range of numbers together by breaking it down into the simple task of adding two numbers:

public class Main {

public static void main(String[] args) {

int result = sum(10);

System.out.println(result);

}

public static int sum(int k) {

if (k > 0) {

return k + sum(k - 1);

} else {

return 0;

}

}

}

### **Example Explained**

When the sum() function is called, it adds parameter k to the sum of all numbers smaller than k and returns the result. When k becomes 0, the function just returns 0. When running, the program follows these steps:

10 + sum(9)  
10 + ( 9 + sum(8) )  
10 + ( 9 + ( 8 + sum(7) ) )  
...  
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + sum(0)  
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 0

Since the function does not call itself when k is 0, the program stops there and returns the result.

## Halting Condition

Just as loops can run into the problem of infinite looping, recursive functions can run into the problem of infinite recursion. Infinite recursion is when the function never stops calling itself. Every recursive function should have a halting condition, which is the condition where the function stops calling itself. In the previous example, the halting condition is when the parameter k becomes 0.

It is helpful to see a variety of different examples to better understand the concept. In this example, the function adds a range of numbers between a start and an end. The halting condition for this recursive function is when **end** is not greater than **start**:

### **Example**

Use recursion to add all of the numbers between 5 to 10.

public class Main {

public static void main(String[] args) {

int result = sum(5, 10);

System.out.println(result);

}

public static int sum(int start, int end) {

if (end > start) {

return end + sum(start, end - 1);

} else {

return end;

}

}

}

The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

## Java - What is OOP?

OOP stands for **Object-Oriented Programming**.

Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.

Object-oriented programming has several advantages over procedural programming:

* OOP is faster and easier to execute
* OOP provides a clear structure for the programs
* OOP helps to keep the Java code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
* OOP makes it possible to create full reusable applications with less code and shorter development time

**Tip:** The "Don't Repeat Yourself" (DRY) principle is about reducing the repetition of code. You should extract out the codes that are common for the application, and place them at a single place and reuse them instead of repeating it.

## Java - What are Classes and Objects?

Classes and objects are the two main aspects of object-oriented programming.

Look at the following illustration to see the difference between class and objects:Ex 1.

## class

Fruit

## objects

Apple

Banana

Mango

Ex 2.

## class

Car

## objects

Volvo

Audi

Toyota

So, a class is a template for objects, and an object is an instance of a class.

## When the individual objects are created, they inherit all the variables and methods from the class.

## Java Classes/Objects

Java is an object-oriented programming language.

Everything in Java is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.

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

## Multiple Objects

You can create multiple objects of one class:

public class Main {

int x = 5;

public static void main(String[] args) {

Main **myObj1** = new Main(); // Object 1

Main **myObj2** = new Main(); // Object 2

System.out.println(myObj1.x);

System.out.println(myObj2.x);

}

}

## Using Multiple Classes

You can also create an object of a class and access it in another class. This is often used for better organization of classes (one class has all the attributes and methods, while the other class holds the main() method (code to be executed)).

Remember that the name of the java file should match the class name. In this example, we have created two files in the same directory/folder:

* Main.java
* Second.java

#### **Main.java**

public class Main {

int x = 5;

}

#### **Second.java**

class Second {

public static void main(String[] args) {

Main **myObj** = new Main();

System.out.println(myObj.x);

}

}

## Java Class Attributes

In the previous chapter, we used the term "variable" for x in the example (as shown below). It is actually an **attribute** of the class. Or you could say that class attributes are variables within a class:

Create a class called "Main" with two attributes: x and y:

public class Main {

int x = 5;

int y = 3;

}

## Accessing Attributes

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

## Modify Attributes

Set the value of x to 40:

public class Main {

int x;

public static void main(String[] args) {

Main myObj = new Main();

myObj.x = 40;

System.out.println(myObj.x);

}

}

If you don't want the ability to override existing values, declare the attribute as final:

public class Main {

**final** int x = 10;

public static void main(String[] args) {

Main myObj = new Main();

myObj.x = 25; // will generate an error: cannot assign a value to a **final** variable

System.out.println(myObj.x);

}

}

The final keyword is useful when you want a variable to always store the same value, like PI (3.14159...).

The final keyword is called a "modifier". You will learn more about these in the [Java Modifiers Chapter](https://www.w3schools.com/java/java_modifiers.asp).

## Multiple Objects

If you create multiple objects of one class, you can change the attribute values in one object, without affecting the attribute values in the other:

Change the value of x to 25 in myObj2, and leave x in myObj1 unchanged:

public class Main {

int x = 5;

public static void main(String[] args) {

Main myObj1 = new Main(); // Object 1

Main myObj2 = new Main(); // Object 2

myObj2.x = 25;

System.out.println(myObj1.x); // Outputs 5

System.out.println(myObj2.x); // Outputs 25

}

}

## Java Class Methods

You learned from the [Java Methods](https://www.w3schools.com/java/java_methods.asp) chapter that methods are declared within a class, and that they are used to perform certain actions:

myMethod() prints a text (the action), when it is **called**. To call a method, write the method's name followed by two parentheses **()** and a semicolon**;**

public class Main {

static void myMethod() {

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

}

public static void main(String[] args) {

myMethod();

}

}

// Outputs "Hello World!"

## Static vs. Public

You will often see Java programs that have either static or public attributes and methods.

In the example above, we created a static method, which means that it can be accessed without creating an object of the class, unlike public, which can only be accessed by objects:

An example to demonstrate the differences between static and public **methods**:

## Access Methods With an Object

### **Example**

Create a Car object named myCar. Call the fullThrottle() and speed() methods on the myCar object, and run the program:

// Create a Main class

public class Main {

// Create a fullThrottle() method

public void fullThrottle() {

System.out.println("The car is going as fast as it can!");

}

// Create a speed() method and add a parameter

public void speed(int maxSpeed) {

System.out.println("Max speed is: " + maxSpeed);

}

// Inside main, call the methods on the myCar object

public static void main(String[] args) {

Main myCar = new Main(); // Create a myCar object

myCar.fullThrottle(); // Call the fullThrottle() method

myCar.speed(200); // Call the speed() method

}

}

// The car is going as fast as it can!

// Max speed is: 200

## Using Multiple Classes

Like we specified in the [Classes chapter](https://www.w3schools.com/java/java_classes.asp), it is a good practice to create an object of a class and access it in another class.

Remember that the name of the java file should match the class name. In this example, we have created two files in the same directory:

* Main.java
* Second.java

#### **Main.java**

public class Main {

public void fullThrottle() {

System.out.println("The car is going as fast as it can!");

}

public void speed(int maxSpeed) {

System.out.println("Max speed is: " + maxSpeed);

}

}

#### **Second.java**

class Second {

public static void main(String[] args) {

Main myCar = new Main(); // Create a myCar object

myCar.fullThrottle(); // Call the fullThrottle() method

myCar.speed(200); // Call the speed() method

}

}

## Java Constructors

A constructor in Java is a **special method** that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes:

Create a constructor:

// Create a Main class

public class Main {

int x; // Create a class attribute

// Create a **class constructor** for the Main class

public Main() {

x = 5; // Set the initial value for the class attribute x

}

public static void main(String[] args) {

Main myObj = new Main(); // Create an object of class Main (This will **call the constructor**)

System.out.println(myObj.x); // Print the value of x

}

}

Note that the constructor name must **match the class name**, and it cannot have a **return type** (like void).

Also note that the constructor is called when the object is created.

All classes have constructors by default: if you do not create a class constructor yourself, Java creates one for you. However, then you are not able to set initial values for object attributes.

## Constructor Parameters

Constructors can also take parameters, which is used to initialize attributes.

The following example adds an int y parameter to the constructor. Inside the constructor we set x to y (x=y). When we call the constructor, we pass a parameter to the constructor (5), which will set the value of x to 5:

public class Main {

int x;

public Main(int y) {

x = y;

}

public static void main(String[] args) {

Main myObj = new Main(5);

System.out.println(myObj.x);

}

}

// Outputs 5

You can have as many parameters as you want:

public class Main {

int modelYear;

String modelName;

public Main(int year, String name) {

modelYear = year;

modelName = name;

}

public static void main(String[] args) {

Main myCar = new Main(1969, "Mustang");

System.out.println(myCar.modelYear + " " + myCar.modelName);

}

}

// Outputs 1969 Mustang

# **Java 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

## Access Modifiers

For **classes**, you can use either public or default:

|  |  |  |
| --- | --- | --- |
| **Modifier** | **Description** | **Try it** |
| public | The class is accessible by any other class | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_mod_public) |
| default | The class is only accessible by classes 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) | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_mod_default) |

For **attributes, methods and constructors**, you can use the one of the following:

|  |  |  |
| --- | --- | --- |
| **Modifier** | **Description** | **Try it** |
| public | The code is accessible for all classes | [Try it »](https://www.w3schools.com/java/tryjava_multi.asp?filename=demo_mod_public2&multi=demo_mod_public2_multi) |
| private | The code is only accessible within the declared class | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_access_mod) |
| 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) | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_mod_default2) |
| 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) | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_mod_protected) |

## Non-Access Modifiers

For **classes**, you can use either final or abstract:

|  |  |  |
| --- | --- | --- |
| **Modifier** | **Description** | **Try it** |
| final | The class cannot be inherited by other classes (You will learn more about inheritance in the [Inheritance chapter](https://www.w3schools.com/java/java_inheritance.asp)) | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_inherit_final) |
| abstract | The class cannot be used to create objects (To access an abstract class, it must be inherited  from another class. You will learn more about inheritance and abstraction in the [Inheritance](https://www.w3schools.com/java/java_inheritance.asp) and [Abstraction](https://www.w3schools.com/java/java_abstract.asp) chapters) | [Try it »](https://www.w3schools.com/java/tryjava_multi.asp?filename=demo_mod_abstract&multi=demo_mod_abstract_multi) |

For **attributes and methods**, you can use the one of the following:

|  |  |
| --- | --- |
| **Modifier** | **Description** |
| final | Attributes and methods cannot be overridden/modified |
| static | Attributes and methods belongs to the class, rather than an object |
| abstract | Can only be used in an abstract class, and can only be used on methods.  The method does not have a body, for example **abstract void run();**.  The body is provided by the subclass (inherited from). You will learn more about  inheritance and abstraction in the [Inheritance](https://www.w3schools.com/java/java_inheritance.asp) and [Abstraction](https://www.w3schools.com/java/java_abstract.asp) chapters. |
| transient | Attributes and methods are skipped when serializing the object containing them |
| synchronized | Methods can only be accessed by one thread at a time |
| volatile | The value of an attribute is not cached thread-locally, and is always read from  the "main memory" |

## Final

If you don't want the ability to override existing attribute values, declare attributes as final:

public class Main {

**final** int x = 10;

**final** double PI = 3.14;

public static void main(String[] args) {

Main myObj = new Main();

myObj.x = 50; // will generate an error: cannot assign a value to a **final** variable

myObj.PI = 25; // will generate an error: cannot assign a value to a **final** variable

System.out.println(myObj.x);

}

}

## Static

A static method means that it can be accessed without creating an object of the class, unlike public:

An example to demonstrate the differences

between static and public methods:

public class Main {

// Static method

static void myStaticMethod() {

System.out.println("Static methods can be called without creating objects");

}

// Public method

public void myPublicMethod() {

System.out.println("Public methods must be called by creating objects");

}

// Main method

public static void main(String[ ] args) {

myStaticMethod(); // Call the static method

// myPublicMethod(); This would output an error

Main myObj = new Main(); // Create an object of Main

myObj.myPublicMethod(); // Call the public method

}

}

## Abstract

An abstract method belongs to an abstract class, and it does not have a body. The body is provided by the subclass:

// Code from filename: Main.java

// abstract class  
abstract class Main {

public String fname = "John";

public int age = 24;

public **abstract** void study(); // abstract method

}

// Subclass (inherit from Main)

class Student extends Main {

public int graduationYear = 2018;

public void study() { // the body of the abstract method is provided here

System.out.println("Studying all day long");

}

}

// End code from filename: Main.java

// Code from filename: Second.java

class Second {

public static void main(String[] args) {

// create an object of the Student class (which inherits attributes and methods from Main)

Student myObj = new Student();

System.out.println("Name: " + myObj.fname);

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

System.out.println("Graduation Year: " + myObj.graduationYear);

myObj.study(); // call abstract method  
 }

}

## Encapsulation

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, you must:

* declare class variables/attributes as private
* provide public **get** and **set** methods to access and update the value of a private variable.

## Get and Set

You learned from the previous chapter that private variables can only be accessed within the same class (an outside class has no access to it). However, it is possible to access them if we provide public **get** and **set** methods.

The get method returns the variable value, and the set method sets the value.

Syntax for both is that they start with either get or set, followed by the name of the variable, with the first letter in upper case:

public class Person {

private String name; // private = restricted access

// Getter

public String getName() {

return name;

}

// Setter

public void setName(String newName) {

this.name = newName;

}

}

#### // Setter **Example explained**

The get method returns the value of the variable name.

The set method takes a parameter (newName) and assigns it to the name variable. The this keyword is used to refer to the current object.

However, as the name variable is declared as private, we **cannot** access it from outside this class:

public class Main {

public static void main(String[] args) {

Person myObj = new Person();

myObj.name = "John"; // error

System.out.println(myObj.name); // error

}

}

If the variable was declared as public, we would expect the following output:

John

Instead, we use the getName() and setName() methods to access and update the variable:

public class Main {

public static void main(String[] args) {

Person myObj = new Person();

myObj.setName("John"); // Set the value of the name variable to "John"

System.out.println(myObj.getName());

}

}

## Why Encapsulation?

* Better control of class attributes and methods
* Class attributes can be made **read-only** (if you only use the get method), or **write-only** (if you only use the set method)
* Flexible: the programmer can change one part of the code without affecting other parts
* Increased security of data

## Java Packages & API

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)

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 class

import package.name.\*; // Import the whole package

# **Java Inheritance**

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.

In the example below, the Car class (subclass) inherits the attributes and methods from the Vehicle class (superclass):

class Vehicle {

protected String brand = "Ford"; // Vehicle attribute

public void honk() { // Vehicle method

System.out.println("Tuut, tuut!");

}

}

class Car extends Vehicle {

private String modelName = "Mustang"; // Car attribute

public static void main(String[] args) {

// Create a myCar object

Car myCar = new Car();

// Call the honk() method (from the Vehicle class) on the myCar object

myCar.honk();

// Display the value of the brand attribute (from the Vehicle class) and the value of the modelName from the Car class

System.out.println(myCar.brand + " " + myCar.modelName);

}

}

Did you notice the protected modifier in Vehicle?

We set the **brand** attribute in **Vehicle** to a protected [access modifier](https://www.w3schools.com/java/java_modifiers.asp). If it was set to private, the Car class would not be able to access it.

#### **Why And When To Use "Inheritance"?**

- It is useful for code reusability: reuse attributes and methods of an existing class when you create a new class.

**Tip:** Also take a look at the next chapter, [Polymorphism](https://www.w3schools.com/java/java_polymorphism.asp), which uses inherited methods to perform different tasks.

## The final 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 {

...

}

## Java Polymorphism

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

Like we specified in the previous chapter; [**Inheritance**](https://www.w3schools.com/java/java_inheritance.asp) lets us inherit attributes and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.

For example, think of a superclass called Animal that has a method called animalSound(). Subclasses of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

class Animal {

public void animalSound() {

System.out.println("The animal makes a sound");

}

}

class Pig extends Animal {

public void animalSound() {

System.out.println("The pig says: wee wee");

}

}

class Dog extends Animal {

public void animalSound() {

System.out.println("The dog says: bow wow");

}

}

Now we can create Pig and Dog objects and call the animalSound() method on both of them:

class Animal {

public void animalSound() {

System.out.println("The animal makes a sound");

}

}

class Pig extends Animal {

public void animalSound() {

System.out.println("The pig says: wee wee");

}

}

class Dog extends Animal {

public void animalSound() {

System.out.println("The dog says: bow wow");

}

}

class Main {

public static void main(String[] args) {

Animal myAnimal = new Animal(); // Create a Animal object

Animal myPig = new Pig(); // Create a Pig object

Animal myDog = new Dog(); // Create a Dog object

myAnimal.animalSound();

myPig.animalSound();

myDog.animalSound();

}

}

# **Java Abstraction**

## Abstract Classes and Methods

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");

}

}

From the example above, it is not possible to create an object of the Animal class:

Animal myObj = new Animal(); // will generate an error

To access the abstract class, it must be inherited from another class. Let's convert the Animal class we used in the [Polymorphism](https://www.w3schools.com/java/java_polymorphism.asp) chapter to an abstract class:

// Abstract class

abstract 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();

}

}

#### **Why And When To Use Abstract Classes and Methods?**

To achieve security - hide certain details and only show the important details of an object.

# **Java Interface**

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**[**Get your own Java Server**](https://www.w3schools.com/java/java_server.asp)

// interface

interface 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:

// Interface

interface 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 interface

class 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();

}

}

#### **Notes on Interfaces:**

* Like **abstract classes**, interfaces **cannot** be used to create objects (in the example above, it is not possible to create an "Animal" object in the MyMainClass)
* Interface methods do not have a body - the body is provided by the "implement" class
* On implementation of an interface, you must override all of its methods
* Interface methods are by default abstract and public
* Interface attributes are by default public, static and final
* An interface cannot contain a constructor (as it cannot be used to create objects)

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

1) To achieve security - hide certain details and only show the important details of an object (interface).

2) Java does not support "multiple inheritance" (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can **implement** multiple interfaces. **Note:** To implement multiple interfaces, separate them with a comma (see example below).

## Multiple Interfaces

To implement multiple interfaces, separate them with a comma:

### **Example**

interface FirstInterface {

public void myMethod(); // interface method

}

interface SecondInterface {

public void myOtherMethod(); // interface method

}

class DemoClass implements FirstInterface, SecondInterface {

public void myMethod() {

System.out.println("Some text..");

}

public void myOtherMethod() {

System.out.println("Some other text...");

}

}

class Main {

public static void main(String[] args) {

DemoClass myObj = new DemoClass();

myObj.myMethod();

myObj.myOtherMethod();

}

}

## 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:

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 Main {

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 Main {

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 myVar : Level.values()) {

System.out.println(myVar);

}

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:

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

class Main {

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 Main {

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);

}

}

**Note:** If you enter wrong input (e.g. text in a numerical input), you will get an exception/error message (like "InputMismatchException").

# **Java Date and Time**

## 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:

|  |  |
| --- | --- |
| **Class** | **Description** |
| LocalDate | Represents a date (year, month, day (yyyy-MM-dd)) |
| LocalTime | Represents a time (hour, minute, second and nanoseconds (HH-mm-ss-ns)) |
| LocalDateTime | Represents both a date and a time (yyyy-MM-dd-HH-mm-ss-ns) |
| DateTimeFormatter | Formatter for displaying and parsing date-time objects |

## Display Current Date

To display the current date, import the java.time.LocalDate class, and use its now() method:

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

}

}

The output will be:

2024-04-09

## Display Current Time

To display the current time (hour, minute, second, and nanoseconds), import the java.time.LocalTime class, and use its now() method:

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);

}

}

The output will be:

08:40:29.133381

## Display Current Date and Time

To display the current date and time, import the java.time.LocalDateTime class, and use its now() method:

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);

}

}

The output will be:

2024-04-09T08:40:29.133061

## Formatting Date and Time

The "T" in the example above is used to separate the date from the time. You can use the DateTimeFormatter class with the ofPattern() method in the same package to format or parse date-time objects. The following example will remove both the "T" and nanoseconds from the date-time:

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);

}

}

The output will be:

Before Formatting: 2024-04-09T08:40:29.134017  
After Formatting: 09-04-2024 08:40:29

The ofPattern() method accepts all sorts of values, if you want to display the date and time in a different format. For example:

|  |  |  |
| --- | --- | --- |
| **Value** | **Example** | **Tryit** |
| *yyyy-MM-dd* | "1988-09-29" | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_date_format) |
| *dd/MM/yyyy* | "29/09/1988" | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_date_formatter1) |
| *dd-MMM-yyyy* | "29-Sep-1988" | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_date_formatter2) |
| *E, MMM dd yyyy* | "Thu, Sep 29 1988" |  |