Java

JavaHome:

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

Example:

public class MyClass {

public static void main(String[] args) {

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

}

}

Introduction:

What is Java?

Java is a popular programming language, created in 1995.

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

It is used for:

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

## Why Use Java?

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

Setup for Windows

To install Java on Windows:

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

## Java QuickStart

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

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

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

MyClass.java

public class MyClass {

public static void main(String[] args) {

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

}

}

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

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

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

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

Java syntax:

### **Example explained**

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

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

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

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

Comments:

Single line comments //

Multiline comments /\* \*/

Java Variables

Variables are containers for storing data values.

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

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

## Final Variables

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

### **Example**

final int myNum = 15;

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

Identifiers:

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

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

Datatypes:

Data types are divided into two groups:

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

## Primitive Data Types

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

There are eight primitive data types in Java:

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

## Numbers

Primitive number types are divided into two groups:

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

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

Note:

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

### **Double**

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

### **Scientific Numbers**

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

### **Example**

float f1 = 35e3f;

double d1 = 12E4d;

System.out.println(f1);

System.out.println(d1);

## Non-Primitive Data Types

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

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

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

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

Java Type Casting

Type casting is when you assign a value of one primitive data type to another type.

In Java, there are two types of casting:

* **Widening Casting** (automatically) - converting a smaller type to a larger type size  
  byte -> short -> char -> int -> long -> float -> double
* **Narrowing Casting** (manually) - converting a larger type to a smaller size type  
  double -> float -> long -> int -> char -> short -> byte

## Narrowing Casting

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

### **Example**

public class MyClass {

public static void main(String[] args) {

double myDouble = 9.78;

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

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

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

}

}

## Java Operators

Operators are used to perform operations on variables and values.

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

Java divides the operators into the following groups:

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

## Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.

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

## Java Assignment Operators

Assignment operators are used to assign values to variables.

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

A list of all assignment operators:

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

## Java Comparison Operators

Comparison operators are used to compare two values:

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

## Java Logical Operators

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

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

## Java Bitwise Operators

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

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

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

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

## Java Strings

Strings are used for storing text.

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

|  |  |  |
| --- | --- | --- |
| **Escape character** | **Result** | **Description** |
| \' | ' | Single quote |
| \" | " | Double quote |
| \\ | \ | Backslash |
| Code | Result |  |
| \n | New Line |  |
| \r | Carriage Return |  |
| \t | Tab |  |
| \b | Backspace |  |
| \f | Form Feed |  |

## All String Methods

The String class has a set of built-in methods that you can use on strings.

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Return Type** |
| [charAt()](https://www.w3schools.com/java/ref_string_charat.asp) | Returns the character at the specified index (position) | char |
| [codePointAt()](https://www.w3schools.com/java/ref_string_codepointat.asp) | Returns the Unicode(Ascii) of the character at the specified index | int |
| [codePointBefore()](https://www.w3schools.com/java/ref_string_codepointbefore.asp) | Returns the Unicode of the character before the specified index | int |
| [codePointCount()](https://www.w3schools.com/java/ref_string_codepointcount.asp) | Returns the Unicode in the specified text range of this String | int |
| [compareTo()](https://www.w3schools.com/java/ref_string_compareto.asp) | Compares two strings lexicographically | int |
| [compareToIgnoreCase()](https://www.w3schools.com/java/ref_string_comparetoignorecase.asp) | Compares two strings lexicographically, ignoring case differences | int |
| [concat()](https://www.w3schools.com/java/ref_string_concat.asp) | Appends a string to the end of another string | String |
| [contains()](https://www.w3schools.com/java/ref_string_contains.asp) | Checks whether a string contains a sequence of characters | boolean |
| [contentEquals()](https://www.w3schools.com/java/ref_string_contentequals.asp) | Checks whether a string contains the exact same sequence of characters of the specified CharSequence or StringBuffer | boolean |
| [copyValueOf()](https://www.w3schools.com/java/ref_string_copyvalueof.asp) | Returns a String that represents the characters of the character array | String |
| [endsWith()](https://www.w3schools.com/java/ref_string_endswith.asp) | Checks whether a string ends with the specified character(s) | boolean |
| [equals()](https://www.w3schools.com/java/ref_string_equals.asp) | Compares two strings. Returns true if the strings are equal, and false if not | boolean |
| [equalsIgnoreCase()](https://www.w3schools.com/java/ref_string_equalsignorecase.asp) | Compares two strings, ignoring case considerations | boolean |
| format() | Returns a formatted string using the specified locale, format string, and arguments | String |
| getBytes() | Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array | byte[] |
| getChars() | Copies characters from a string to an array of chars | void |
| [hashCode()](https://www.w3schools.com/java/ref_string_hashcode.asp) | Returns the hash code of a string | int |
| [indexOf()](https://www.w3schools.com/java/ref_string_indexof.asp) | Returns the position of the first found occurrence of specified characters in a string | int |
| intern() | Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index | String |
| [isEmpty()](https://www.w3schools.com/java/ref_string_isempty.asp) | Checks whether a string is empty or not | boolean |
| [lastIndexOf()](https://www.w3schools.com/java/ref_string_lastindexof.asp) | Returns the position of the last found occurrence of specified characters in a string | int |
| [length()](https://www.w3schools.com/java/ref_string_length.asp) | Returns the length of a specified string | int |
| matches() | Searches a string for a match against a regular expression, and returns the matches | boolean |
| offsetByCodePoints() | Returns the index within this String that is offset from the given index by codePointOffset code points | int |
| regionMatches() | Tests if two string regions are equal | boolean |
| [replace()](https://www.w3schools.com/java/ref_string_replace.asp) | Searches a string for a specified value, and returns a new string where the specified values are replaced | String |
| replaceFirst() | Replaces the first occurrence of a substring that matches the given regular expression with the given replacement | String |
| replaceAll() | Replaces each substring of this string that matches the given regular expression with the given replacement | String |
| split() | Splits a string into an array of substrings | String[] |
| [startsWith()](https://www.w3schools.com/java/ref_string_startswith.asp) | Checks whether a string starts with specified characters | boolean |
| subSequence() | Returns a new character sequence that is a subsequence of this sequence | CharSequence |
| substring() | Extracts the characters from a string, beginning at a specified start position, and through the specified number of character | String |
| toCharArray() | Converts this string to a new character array | char[] |
| [toLowerCase()](https://www.w3schools.com/java/ref_string_tolowercase.asp) | Converts a string to lower case letters | String |
| toString() | Returns the value of a String object | String |
| [toUpperCase()](https://www.w3schools.com/java/ref_string_touppercase.asp) | Converts a string to upper case letters | String |
| [trim()](https://www.w3schools.com/java/ref_string_trim.asp) | Removes whitespace from both ends of a string | String |
| valueOf() | Returns the primitive value of a String object | String |

codePointCount():

String myStr = "Hello";

int result = myStr.codePointCount(0, 5);

compareTo() and compareToIgnoreCase():

|  |  |
| --- | --- |
| **Returns:** | An int value: 0 if the string is equal to the other string, ignoring case differences. < 0 if the string is lexicographically less than the other string > 0 if the string is lexicographically greater than the other string (more characters) |

[copyValueOf()](https://www.w3schools.com/java/ref_string_copyvalueof.asp):

char[] myStr1 = {'H', 'e', 'l', 'l', 'o'};

String myStr2 = "";

myStr2 = myStr2.copyValueOf(myStr1, 0, 5);

# **String hashCode()**

Return the hash code of a string:

String myStr = "Hello";

System.out.println(myStr.hashCode());

## Definition and Usage

The hashCode() method returns the hash code of a string.

The hash code for a String object is computed like this:

s[0]\*31^(n-1) + s[1]\*31^(n-2) + ... + s[n-1]

where s[i] is the ith character of the string, n is the length of the string, and ^ indicates exponentiation

split:

String[] str=MyString.split("p",3);

System.out.println(str.toString());

for (String a : str)

System.out.println(a);

## Java Math:

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

## All Math Methods

A list of all Math methods can be found in the table below:

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Return Type** |
| [abs(x)](https://www.w3schools.com/java/ref_math_abs.asp) | Returns the absolute value of x | double|float|int|long |
| [acos(x)](https://www.w3schools.com/java/ref_math_acos.asp) | Returns the arccosine of x, in radians | double |
| [asin(x)](https://www.w3schools.com/java/ref_math_asin.asp) | Returns the arcsine of x, in radians | double |
| atan(x) | Returns the arctangent of x as a numeric value between -PI/2 and PI/2 radians | double |
| atan2(y,x) | Returns the angle theta from the conversion of rectangular coordinates (x, y) to polar coordinates (r, theta). | double |
| cbrt(x) | Returns the cube root of x | double |
| ceil(x) | Returns the value of x rounded up to its nearest integer | double |
| copySign(x, y) | Returns the first floating point x with the sign of the second floating point y | double |
| cos(x) | Returns the cosine of x (x is in radians) | double |
| cosh(x) | Returns the hyperbolic cosine of a double value | double |
| exp(x) | Returns the value of Ex | double |
| expm1(x) | Returns ex -1 | double |
| floor(x) | Returns the value of x rounded down to its nearest integer | double |
| getExponent(x) | Returns the unbiased exponent used in x | int |
| hypot(x, y) | Returns sqrt(x2 +y2) without intermediate overflow or underflow | double |
| IEEEremainder(x, y) | Computes the remainder operation on x and y as prescribed by the IEEE 754 standard | double |
| log(x) | Returns the natural logarithm (base E) of x | double |
| log10(x) | Returns the base 10 logarithm of x | double |
| log1p(x) | Returns the natural logarithm (base E) of the sum of x and 1 | double |
| max(x, y) | Returns the number with the highest value | double|float|int|long |
| min(x, y) | Returns the number with the lowest value | double|float|int|long |
| nextAfter(x, y) | Returns the floating point number adjacent to x in the direction of y | double|float |
| nextUp(x) | Returns the floating point value adjacent to x in the direction of positive infinity | double|float |
| pow(x, y) | Returns the value of x to the power of y | double |
| random() | Returns a random number between 0 and 1 | double |
| round(x) | Returns the value of x rounded to its nearest integer | int |
| rint() | Returns the double value that is closest to x and equal to a mathematical integer | double |
| signum(x) | Returns the sign of x | double |
| sin(x) | Returns the sine of x (x is in radians) | double |
| sinh(x) | Returns the hyperbolic sine of a double value | double |
| sqrt(x) | Returns the square root of x | double |
| tan(x) | Returns the tangent of an angle | double |
| tanh(x) | Returns the hyperbolic tangent of a double value | double |
| toDegrees(x) | Converts an angle measured in radians to an approx. equivalent angle measured in degrees | double |
| toRadians(x) | Converts an angle measured in degrees to an approx. angle measured in radians | double |
| ulp(x) | Returns the size of the unit of least precision (ulp) of x | double|float |

**Note:** All Math methods are static.

public class MathTest{

public static void main(String[] args){

System.out.println(Math.abs(-1.43));

System.out.println(Math.acos(0.5)\*(180)/3.14);//returns radains in double but i'm converting to degress

System.out.println(Math.atan2(2,2)\*(180)/3.14);//angle btwn the two rectangular cordinates

System.out.println(Math.cbrt(8));//return cube root of x

System.out.println(Math.ceil(1.000001));//x>=x(whole number);

System.out.println(Math.copySign(3,-2));// 3 copies the sign of 2

System.out.println(90\*180/3.14);

System.out.println(90\*180/(22/7));

System.out.println(Math.cos(11/7));

System.out.println(11/7);

System.out.println(Math.exp(3));//returns e(2.718281828459045) pow x

System.out.println(Math.floor(2.8));

System.out.println(Math.hypot(4,3));

System.out.println(Math.rint(1.5));//return bouble/float [.0-.5],(0.5-1.0)

System.out.println(Math.toRadians(45));

}

}

## Boolean Values

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

However, it is more common to return boolean values from boolean expressions, for conditional testing (see below).

## Boolean Expression

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

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

The Boolean value of an expression is the basis for all Java comparisons and conditions.

# **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:

### **Example**

Create a method inside MyClass:

public class MyClass {

static void myMethod() {

// code to be executed

}

}

#### **Example Explained**

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

## Call a Method

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

In the following example, myMethod() is used to print a text (the action), when it is called:

### **Example**

Inside main, call the myMethod() method:

public class MyClass {

static void myMethod() {

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

}

public static void main(String[] args) {

myMethod();

}

}

// Outputs "I just got executed!"

## Parameters and Arguments

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

### **Example**

public class MyClass {

static void myMethod(String fname) {

System.out.println(fname + " Refsnes");

}

public static void main(String[] args) {

myMethod("Liam");

myMethod("Jenny");

myMethod("Anja");

}

}

// Liam Refsnes

// Jenny Refsnes

// Anja Refsnes

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**.

## Method Overloading

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

### **Example**

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);

}

## 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.

## 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. A block of code may exist on its own or it may 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.

### **Example**

public class MyClass {

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

}

}

## 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 when encountering it for the first time, 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:

### **Example**

Use recursion to add all of the numbers up to 10.

public class MyClass {

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;

}

}

}

[Run example »](https://www.w3schools.com/java/showjava.asp?filename=demo_recursion)

### **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

Java Oop

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:

class

Fruit

objects

Apple

Banana

Mango

Another example:

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.

## Create a Class

To create a class, use the keyword class:

### **MyClass.java**

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

public class MyClass {

int x = 5;

}

Remember from the [Java Syntax chapter](https://www.w3schools.com/java/java_syntax.asp) that a class should always start with an uppercase first letter, and that the name of the java file should match the class name.

## Create an Object

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

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

### **Example**

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

public class MyClass {

int x = 5;

public static void main(String[] args) {

MyClass **myObj** = new MyClass();

System.out.println(myObj.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:

* MyClass.java
* OtherClass.java

#### **MyClass.java**

public class MyClass {

int x = 5;

}

#### **OtherClass.java**

class OtherClass {

public static void main(String[] args) {

MyClass **myObj** = new MyClass();

System.out.println(myObj.x);

}

}

When both files have been compiled:

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

Run the OtherClass.java file:

C:\Users\Your Name>java OtherClass

And the output will be:

5

## 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:

Another term for class attributes is **fields**. 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).

## Java Class Methods

## Static vs. Non-Static

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:

### **Example**

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

public class MyClass {

// 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 compile an error

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

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

}

}

## 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:

### **Example**

Create a constructor:

// Create a MyClass class

public class MyClass {

int x; // Create a class attribute

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

public MyClass() {

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

}

public static void main(String[] args) {

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

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

}

}

// Outputs 5

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.

### **Example**

public class Car {

int modelYear;

String modelName;

public Car(int year, String name) {

modelYear = year;

modelName = name;

}

public static void main(String[] args) {

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

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

}

}

// Outputs 1969 Mustang

## Modifiers

By now, you are quite familiar with the public keyword that appears in almost all of our examples:

**public** class MyClass

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/showjava.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/showjava.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/showjava_classes.asp?filename=demo_mod_public2) |
| private | The code is only accessible within the declared class | [Try it »](https://www.w3schools.com/java/showjava.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/showjava.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/showjava.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/showjava.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/showjava_classes.asp?filename=demo_mod_abstract) |

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" |

## 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:

### **Example**

public class Person {

private String name; // private = restricted access

// Getter

public String getName() {

return name;

}

// Setter

public void setName(String newName) {

this.name = newName;

}

}

## 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)

## Built-in Packages

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

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

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

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

### **Syntax**

import package.name.Class; // Import a single class

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

## Import a Class

If you find a class you want to use, for example, the Scanner class, **which is used to get user input**, write the following code:

### **Example**

import java.util.Scanner;

In the example above, java.util is a package, while Scanner is a class of 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 a complete line:

### **Example**

Using the Scanner class to get user input:

import java.util.Scanner;

class MyClass {

public static void main(String[] args) {

Scanner myObj = new Scanner(System.in);

System.out.println("Enter username");

String userName = myObj.nextLine();

System.out.println("Username is: " + userName);

}

}

## User-defined Packages

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

### **Example**

└── root

└── mypack

└── MyPackageClass.java

To create a package, use the package keyword:

### **MyPackageClass.java**

package mypack;

class MyPackageClass {

public static void main(String[] args) {

System.out.println("This is my package!");

}

}

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

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

Then compile the package:

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

This forces the compiler to create the "mypack" package.

The -d keyword specifies the destination for where to save the class file. You can use any directory name, like c:/user (windows), or, if you want to keep the package within the same directory, you can use the dot sign ".", like in the example above.

**Note:** The package name should be written in lower case to avoid conflict with class names.

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

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

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

The output will be:

This is my package!

Java Inheritance (Subclass and Superclass)

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):

### **Example**

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 {

...

}

The output will be something like this:

Car.java:8: error: cannot inherit from final Vehicle  
class Car extends Vehicle {  
                  ^  
1 error)

## 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.):

### **Example**

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");

}

}

Remember from the [Inheritance chapter](https://www.w3schools.com/java/java_inheritance.asp) that we use the extends keyword to inherit from a class.

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

### **Example**

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 MyMainClass {

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();

}

}

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

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

## Java Inner Classes

In Java, it is also possible to nest classes (a class within a class). The purpose of nested classes is to group classes that belong together, which makes your code more readable and maintainable.

To access the inner class, create an object of the outer class, and then create an object of the inner class:

### **Example**

class OuterClass {

int x = 10;

class InnerClass {

int y = 5;

}

}

public class MyMainClass {

public static void main(String[] args) {

OuterClass myOuter = new OuterClass();

OuterClass.InnerClass myInner = myOuter.new InnerClass();

System.out.println(myInner.y + myOuter.x);

}

}

// Outputs 15 (5 + 10)

## Private Inner Class

Unlike a "regular" class, an inner class can be private or protected. If you don't want outside objects to access the inner class, declare the class as private:

### **Example**

class OuterClass {

int x = 10;

**private** class InnerClass {

int y = 5;

}

}

public class MyMainClass {

public static void main(String[] args) {

OuterClass myOuter = new OuterClass();

OuterClass.InnerClass myInner = myOuter.new InnerClass();

System.out.println(myInner.y + myOuter.x);

}

}

If you try to access a private inner class from an outside class (MyMainClass), an error occurs:

MyMainClass.java:12: error: OuterClass.InnerClass has private access in OuterClass  
    OuterClass.InnerClass myInner = myOuter.new InnerClass();  
              ^

## Static Inner Class

An inner class can also be static, which means that you can access it without creating an object of the outer class:

### **Example**

class OuterClass {

int x = 10;

static class InnerClass {

int y = 5;

}

}

public class MyMainClass {

public static void main(String[] args) {

OuterClass.InnerClass myInner = new OuterClass.InnerClass();

System.out.println(myInner.y);

}

}

// Outputs 5

**Note:** just like static attributes and methods, a static inner class does not have access to members of the outer class.

## Access Outer Class From Inner Class

One advantage of inner classes, is that they can access attributes and methods of the outer class:

### **Example**

class OuterClass {

int x = 10;

class InnerClass {

public int myInnerMethod() {

return x;

}

}

}

public class MyMainClass {

public static void main(String[] args) {

OuterClass myOuter = new OuterClass();

OuterClass.InnerClass myInner = myOuter.new InnerClass();

System.out.println(myInner.myInnerMethod());

}

}

// Outputs 10

}

// 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 MyMainClass {

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.

**Note:** Abstraction can also be achieved with [Interfaces](https://www.w3schools.com/java/java_interface.asp), which you will learn more about in the next chapter.

## Interfaces

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

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

### **Example**

// 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:

### **Example**

// 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 MyMainClass {

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 MyMainClass {

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:

### **Example**

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 MyClass {

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 MyClass {

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

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:

### **Example**

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

class MyClass {

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

}

}

If you don't know what a package is, read our [Java Packages Tutorial](https://www.w3schools.com/java/java_packages.asp).

## 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 MyClass {

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 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:

### **Example**

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

public class MyClass {

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:

2020-09-11

## Display Current Time

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

### **Example**

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

public class MyClass {

public static void main(String[] args) {

LocalTime myObj = LocalTime.now();

System.out.println(myObj);

}

}

The output will be:

20:48:27.286711

## Display Current Date and Time

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

### **Example**

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

public class MyClass {

public static void main(String[] args) {

LocalDateTime myObj = LocalDateTime.now();

System.out.println(myObj);

}

}

The output will be:

2020-09-11T20:48:27.395219

## 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:

### **Example**

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

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

public class MyClass {

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: 2020-09-11T20:48:27.396175  
After Formatting: 11-09-2020 20:48:27

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

## Java ArrayList

The ArrayList class is a resizable [array](https://www.w3schools.com/java/java_arrays.asp), which can be found in the java.util package.

The difference between a built-in array and an ArrayList in Java, is that the size of an array cannot be modified (if you want to add or remove elements to/from an array, you have to create a new one). While elements can be added and removed from an ArrayList whenever you want. The syntax is also slightly different:

### **Example**

Create an ArrayList object called **cars** that will store strings:

import java.util.ArrayList; // import the ArrayList class

ArrayList<String> cars = new ArrayList<String>(); // Create an ArrayList object

## Add Items

The ArrayList class has many useful methods. For example, to add elements to the ArrayList, use the add() method:

### **Example**

import java.util.ArrayList;

public class MyClass {

public static void main(String[] args) {

ArrayList<String> cars = new ArrayList<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

System.out.println(cars);

}

}

## Access an Item

To access an element in the ArrayList, use the get() method and refer to the index number:

### **Example**

cars.get(0);

## Change an Item

To modify an element, use the set() method and refer to the index number:

### **Example**

cars.set(0, "Opel");

## Remove an Item

To remove an element, use the remove() method and refer to the index number:

### **Example**

cars.remove(0);

To remove all the elements in the ArrayList, use the clear() method:

### **Example**

cars.clear();

## ArrayList Size

To find out how many elements an ArrayList have, use the size method:

### **Example**

cars.size();

## Loop Through an ArrayList

Loop through the elements of an ArrayList with a for loop, and use the size() method to specify how many times the loop should run:

### **Example**

public class MyClass {

public static void main(String[] args) {

ArrayList<String> cars = new ArrayList<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

for (int i = 0; i < cars.size(); i++) {

System.out.println(cars.get(i));

}

}

}

You can also loop through an ArrayList with the **for-each** loop:

### **Example**

public class MyClass {

public static void main(String[] args) {

ArrayList<String> cars = new ArrayList<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

for (String i : cars) {

System.out.println(i);

}

}

}

## Other Types

Elements in an ArrayList are actually objects. In the examples above, we created elements (objects) of type "String". Remember that a String in Java is an object (not a primitive type). To use other types, such as int, you must specify an equivalent [wrapper class](https://www.w3schools.com/java/java_wrapper_classes.asp): Integer. For other primitive types, use: Boolean for boolean, Character for char, Double for double, etc:

### **Example**

Create an ArrayList to store numbers (add elements of type Integer):

import java.util.ArrayList;

public class MyClass {

public static void main(String[] args) {

ArrayList<Integer> myNumbers = new ArrayList<Integer>();

myNumbers.add(10);

myNumbers.add(15);

myNumbers.add(20);

myNumbers.add(25);

for (int i : myNumbers) {

System.out.println(i);

}

}

}

## Sort an ArrayList

Another useful class in the java.util package is the Collections class, which include the sort() method for sorting lists alphabetically or numerically:

### **Example**

Sort an ArrayList of Strings:

import java.util.ArrayList;

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

public class MyClass {

public static void main(String[] args) {

ArrayList<String> cars = new ArrayList<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

Collections.sort(cars); // Sort cars

for (String i : cars) {

System.out.println(i);

}

}

}

## Java LinkedList

In the previous chapter, you learned about the [ArrayList](https://www.w3schools.com/java/java_arraylist.asp) class. The LinkedList class is almost identical to the ArrayList:

### **Example**

// Import the LinkedList class

import java.util.LinkedList;

public class MyClass {

public static void main(String[] args) {

LinkedList<String> cars = new LinkedList<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

System.out.println(cars);

}

}

## ArrayList vs. LinkedList

The LinkedList class is a collection which can contain many objects of the same type, just like the ArrayList.

The LinkedList class has all of the same methods as the ArrayList class because they both implement the List interface. This means that you can add items, change items, remove items and clear the list in the same way.

However, while the ArrayList class and the LinkedList class can be used in the same way, they are built very differently.

### **How the ArrayList works**

The ArrayList class has a regular array inside it. When an element is added, it is placed into the array. If the array is not big enough, a new, larger array is created to replace the old one and the old one is removed.

### **How the LinkedList works**

The LinkedList stores its items in "containers." The list has a link to the first container and each container has a link to the next container in the list. To add an element to the list, the element is placed into a new container and that container is linked to one of the other containers in the list.

### **When To Use**

It is best to use an ArrayList when:

* You want to access random items frequently
* You only need to add or remove elements at the end of the list

It is best to use a LinkedList when:

* You only use the list by looping through it instead of accessing random items
* You frequently need to add and remove items from the beginning or middle of the

list

## LinkedList Methods

For many cases, the ArrayList is more efficient as it is common to need access to random items in the list, but the LinkedList provides several methods to do certain operations more efficiently:

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Try it** |
| addFirst() | Adds an item to the beginning of the list. | [Try it »](https://www.w3schools.com/java/showjava.asp?filename=demo_linkedlist_addfirst) |
| addLast() | Add an item to the end of the list | [Try it »](https://www.w3schools.com/java/showjava.asp?filename=demo_linkedlist_addlast) |
| removeFirst() | Remove an item from the beginning of the list. | [Try it »](https://www.w3schools.com/java/showjava.asp?filename=demo_linkedlist_removefirst) |
| removeLast() | Remove an item from the end of the list | [Try it »](https://www.w3schools.com/java/showjava.asp?filename=demo_linkedlist_removelast) |
| getFirst() | Get the item at the beginning of the list | [Try it »](https://www.w3schools.com/java/showjava.asp?filename=demo_linkedlist_getfirst) |
| getLast() | Get the item at the end of the list |  |

## Java HashMap

In the [ArrayList](https://www.w3schools.com/java/java_arraylist.asp) chapter, you learned that Arrays store items as an ordered collection, and you have to access them with an index number (int type). A HashMap however, store items in "**key**/**value**" pairs, and you can access them by an index of another type (e.g. a String).

One object is used as a key (index) to another object (value). It can store different types: String keys and Integer values, or the same type, like: String keys and String values:

### **Example**

Create a HashMap object called **capitalCities** that will store String **keys** and String **values**:

import java.util.HashMap; // import the HashMap class

HashMap<String, String> capitalCities = new HashMap<String, String>();

## Add Items

The HashMap class has many useful methods. For example, to add items to it, use the put() method:

### **Example**

// Import the HashMap class

import java.util.HashMap;

public class MyClass {

public static void main(String[] args) {

// Create a HashMap object called capitalCities

HashMap<String, String> capitalCities = new HashMap<String, String>();

// Add keys and values (Country, City)

capitalCities.put("England", "London");

capitalCities.put("Germany", "Berlin");

capitalCities.put("Norway", "Oslo");

capitalCities.put("USA", "Washington DC");

System.out.println(capitalCities);

}

}

## Access an Item

To access a value in the HashMap, use the get() method and refer to its key:

### **Example**

capitalCities.get("England");

## Remove an Item

To remove an item, use the remove() method and refer to the key:

To remove all items, use the clear() method:

To find out how many items there are, use the size method

## Loop Through a HashMap

Loop through the items of a HashMap with a **for-each** loop.

**Note:** Use the keySet() method if you only want the keys, and use the values() method if you only want the values:

### **Example**

// Print keys

for (String i : capitalCities.keySet()) {

System.out.println(i);

}

## Java Iterator

An Iterator is an object that can be used to loop through collections, like [ArrayList](https://www.w3schools.com/java/java_arraylist.asp) and [HashSet](https://www.w3schools.com/java/java_hashset.asp). It is called an "iterator" because "iterating" is the technical term for looping.

To use an Iterator, you must import it from the java.util package.

## Getting an Iterator

The iterator() method can be used to get an Iterator for any collection:

### **Example**

// Import the ArrayList class and the Iterator class

import java.util.ArrayList;

import java.util.Iterator;

public class MyClass {

public static void main(String[] args) {

// Make a collection

ArrayList<String> cars = new ArrayList<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

// Get the iterator

Iterator<String> it = cars.iterator();

// Print the first item

System.out.println(it.next());

}

}

## Looping Through a Collection

To loop through a collection, use the hasNext() and next() methods of the Iterator:

### **Example**

while(it.hasNext()) {

System.out.println(it.next());

}

## Removing Items from a Collection

Iterators are designed to easily change the collections that they loop through. The remove() method can remove items from a collection while looping.

### **Example**

Use an iterator to remove numbers less than 10 from a collection:

import java.util.ArrayList;

import java.util.Iterator;

public class MyClass {

public static void main(String[] args) {

ArrayList<Integer> numbers = new ArrayList<Integer>();

numbers.add(12);

numbers.add(8);

numbers.add(2);

numbers.add(23);

Iterator<Integer> it = numbers.iterator();

while(it.hasNext()) {

Integer i = it.next();

if(i < 10) {

it.remove();

}

}

System.out.println(numbers);

}

}

**Note:** Trying to remove items using a **for loop** or a **for-each loop** would not work correctly because the collection is changing size at the same time that the code is trying to loop.

## Java Wrapper Classes

Wrapper classes provide a way to use primitive data types (int, boolean, etc..) as objects.

The table below shows the primitive type and the equivalent wrapper class:

|  |  |
| --- | --- |
| **Primitive Data Type** | **Wrapper Class** |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |
| boolean | Boolean |
| char | Character |

Sometimes you must use wrapper classes, for example when working with Collection objects, such as ArrayList, where primitive types cannot be used (the list can only store objects):

### **Example**

ArrayList<int> myNumbers = new ArrayList<int>(); // Invalid

ArrayList<Integer> myNumbers = new ArrayList<Integer>(); // Valid

## Creating Wrapper Objects

To create a wrapper object, use the wrapper class instead of the primitive type. To get the value, you can just print the object:

### **Example**

public class MyClass {

public static void main(String[] args) {

Integer myInt = 5;

Double myDouble = 5.99;

Character myChar = 'A';

System.out.println(myInt);

System.out.println(myDouble);

System.out.println(myChar);

}

}

[Run example »](https://www.w3schools.com/java/showjava.asp?filename=demo_wrapper_create)

Since you're now working with objects, you can use certain methods to get information about the specific object.

For example, the following methods are used to get the value associated with the corresponding wrapper object: intValue(), byteValue(), shortValue(), longValue(), floatValue(), doubleValue(), charValue(), booleanValue().

This example will output the same result as the example above:

### **Example**

public class MyClass {

public static void main(String[] args) {

Integer myInt = 5;

Double myDouble = 5.99;

Character myChar = 'A';

System.out.println(myInt.intValue());

System.out.println(myDouble.doubleValue());

System.out.println(myChar.charValue());

}

}

[Run example »](https://www.w3schools.com/java/showjava.asp?filename=demo_wrapper_methods)

Another useful method is the toString() method, which is used to convert wrapper objects to strings.

In the following example, we convert an Integer to a String, and use the length() method of the String class to output the length of the "string":

### **Example**

public class MyClass {

public static void main(String[] args) {

Integer myInt = 100;

String myString = myInt.toString();

System.out.println(myString.length());

}

}

[Run example »](https://www.w3schools.com/java/showjava.asp?filename=demo_wrapper_string)

## Java Exceptions

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

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

## Java try and catch

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

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

The try and catch keywords come in pairs:

### **Syntax**

try {

// *Block of code to try*

}

catch(Exception e) {

// *Block of code to handle errors*

}

Consider the following example:

This will generate an error, because **myNumbers[10]** does not exist.

public class MyClass {

public static void main(String[ ] args) {

int[] myNumbers = {1, 2, 3};

System.out.println(myNumbers[10]); // error!

}

}

The output will be something like this:

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10  
        at MyClass.main(MyClass.java:4)

[Run example »](https://www.w3schools.com/java/showjava.asp?filename=demo_try_error)

If an error occurs, we can use try...catch to catch the error and execute some code to handle it:

### **Example**

public class MyClass {

public static void main(String[ ] args) {

try {

int[] myNumbers = {1, 2, 3};

System.out.println(myNumbers[10]);

} catch (Exception e) {

System.out.println("Something went wrong.");

}

}

}

The output will be:

Something went wrong.

[Run example »](https://www.w3schools.com/java/showjava.asp?filename=demo_try_catch)

## Finally

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

### **Example**

public class MyClass {

public static void main(String[] args) {

try {

int[] myNumbers = {1, 2, 3};

System.out.println(myNumbers[10]);

} catch (Exception e) {

System.out.println("Something went wrong.");

} finally {

System.out.println("The 'try catch' is finished.");

}

}

}

The output will be:

Something went wrong.  
The 'try catch' is finished.

[Run example »](https://www.w3schools.com/java/showjava.asp?filename=demo_try_catch_finally)

## The throw keyword

The throw statement allows you to create a custom error.

The throw statement is used together with an **exception type**. There are many exception types available in Java: ArithmeticException, FileNotFoundException, ArrayIndexOutOfBoundsException, SecurityException, etc:

### **Example**

Throw an exception if **age** is below 18 (print "Access denied"). If age is 18 or older, print "Access granted":

public class MyClass {

static void checkAge(int age) {

if (age < 18) {

throw new ArithmeticException("Access denied - You must be at least 18 years old.");

}

else {

System.out.println("Access granted - You are old enough!");

}

}

public static void main(String[] args) {

checkAge(15); // Set age to 15 (which is below 18...)

}

}

The output will be:

Exception in thread "main" java.lang.ArithmeticException: Access denied - You must be at least 18 years old.  
        at MyClass.checkAge(MyClass.java:4)  
        at MyClass.main(MyClass.java:12)

[Run example »](https://www.w3schools.com/java/showjava.asp?filename=demo_throw)

If **age** was 20, you would **not** get an exception:

### **Example**

checkAge(20);

The output will be:

Access granted - You are old enough!

## What is a Regular Expression?

A regular expression is a sequence of characters that forms a search pattern. When you search for data in a text, you can use this search pattern to describe what you are searching for.

A regular expression can be a single character, or a more complicated pattern.

Regular expressions can be used to perform all types of **text search** and **text replace** operations.

Java does not have a built-in Regular Expression class, but we can import the java.util.regex package to work with regular expressions. The package includes the following classes:

* Pattern Class - Defines a pattern (to be used in a search)
* Matcher Class - Used to search for the pattern
* PatternSyntaxException Class - Indicates syntax error in a regular expression pattern

### **Example**

Find out if there are any occurrences of the word "w3schools" in a sentence:

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class MyClass {

public static void main(String[] args) {

Pattern pattern = Pattern.compile("w3schools", Pattern.CASE\_INSENSITIVE);

Matcher matcher = pattern.matcher("Visit W3Schools!");

boolean matchFound = matcher.find();

if(matchFound) {

System.out.println("Match found");

} else {

System.out.println("Match not found");

}

}

}

// Outputs Match found

[Run example »](https://www.w3schools.com/java/showjava.asp?filename=demo_regex)

### **Example Explained**

In this example, The word "w3schools" is being searched for in a sentence.

First, the pattern is created using the Pattern.compile() method. The first parameter indicates which pattern is being searched for and the second parameter has a flag to indicates that the search should be case-insensitive. The second parameter is optional.

The matcher() method is used to search for the pattern in a string. It returns a Matcher object which contains information about the search that was performed.

The find() method returns true if the pattern was found in the string and false if it was not found.

## Flags

Flags in the compile() method change how the search is performed. Here are a few of them:

* Pattern.CASE\_INSENSITIVE - The case of letters will be ignored when performing a search.
* Pattern.LITERAL - Special characters in the pattern will not have any special meaning and will be treated as ordinary characters when performing a search.
* Pattern.UNICODE\_CASE - Use it together with the CASE\_INSENSITIVE flag to also ignore the case of letters outside of the English alphabet

## Regular Expression Patterns

The first parameter of the Pattern.compile() method is the pattern. It describes what is being searched for.

Brackets are used to find a range of characters:

|  |  |
| --- | --- |
| **Expression** | **Description** |
| [abc] | Find one character from the options between the brackets |
| [^abc] | Find one character NOT between the brackets |
| [0-9] | Find one character from the range 0 to 9 |

## Metacharacters

Metacharacters are characters with a special meaning:

|  |  |
| --- | --- |
| **Metacharacter** | **Description** |
| | | Find a match for any one of the patterns separated by | as in: cat|dog|fish |
| . | Find just one instance of any character |
| ^ | Finds a match as the beginning of a string as in: ^Hello |
| $ | Finds a match at the end of the string as in: World$ |
| \d | Find a digit |
| \s | Find a whitespace character |
| \b | Find a match at the beginning of a word like this: \bWORD, or at the end of a word like this: WORD\b |
| \uxxxx | Find the Unicode character specified by the hexadecimal number xxxx |

## Quantifiers

Quantifiers define quantities:

|  |  |
| --- | --- |
| **Quantifier** | **Description** |
| n+ | Matches any string that contains at least one n |
| n\* | Matches any string that contains zero or more occurrences of n |
| n? | Matches any string that contains zero or one occurrences of n |
| n{x} | Matches any string that contains a sequence of *X* *n*'s |
| n{x,y} | Matches any string that contains a sequence of X to Y *n*'s |
| n{x,} | Matches any string that contains a sequence of at least X *n*'s |

**Note:** If your expression needs to search for one of the special characters you can use a backslash ( \ ) to escape them. In Java, backslashes in strings need to be escaped themselves, so two backslashes are needed to escape special characters. For example, to search for one or more question marks you can use the following expression: "\\?"

## Java Threads

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

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

## Creating a Thread

There are two ways to create a thread.

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

### **Extend Syntax**

public class MyClass extends Thread {

public void run() {

System.out.println("This code is running in a thread");

}

}

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

### **Implement Syntax**

public class MyClass implements Runnable {

public void run() {

System.out.println("This code is running in a thread");

}

}

## Running Threads

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

### **Extend Example**

public class MyClass extends Thread {

public static void main(String[] args) {

MyClass thread = new MyClass();

thread.start();

System.out.println("This code is outside of the thread");

}

public void run() {

System.out.println("This code is running in a thread");

}

}

[Run example »](https://www.w3schools.com/java/showjava.asp?filename=demo_threads)

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

### **Implement Example**

public class MyClass implements Runnable {

public static void main(String[] args) {

MyClass obj = new MyClass();

Thread thread = new Thread(obj);

thread.start();

System.out.println("This code is outside of the thread");

}

public void run() {

System.out.println("This code is running in a thread");

}

}

[Run example »](https://www.w3schools.com/java/showjava.asp?filename=demo_threads_runnable)

**Differences between "extending" and "implementing" Threads**

The major difference is that when a class extends the Thread class, you cannot extend any other class, but by implementing the Runnable interface, it is possible to extend from another class as well, like: class MyClass extends OtherClass implements Runnable.

## Concurrency Problems

Because threads run at the same time as other parts of the program, there is no way to know in which order the code will run. When the threads and main program are reading and writing the same variables, the values are unpredictable. The problems that result from this are called concurrency problems.

### **Example**

A code example where the value of the variable **amount** is unpredictable:

public class MyClass extends Thread {

public static int amount = 0;

public static void main(String[] args) {

MyClass thread = new MyClass();

thread.start();

System.out.println(amount);

amount++;

System.out.println(amount);

}

public void run() {

amount++;

}

}

[Run example »](https://www.w3schools.com/java/showjava.asp?filename=demo_threads_concurrency)

To avoid concurrency problems, it is best to share as few attributes between threads as possible. If attributes need to be shared, one possible solution is to use the isAlive() method of the thread to check whether the thread has finished running before using any attributes that the thread can change.

### **Example**

Use isAlive() to prevent concurrency problems:

public class MyClass extends Thread {

public static int amount = 0;

public static void main(String[] args) {

MyClass thread = new MyClass();

thread.start();

// Wait for the thread to finish

while(thread.isAlive()) {

System.out.println("Waiting...");

}

// Update amount and print its value

System.out.println("Main: " + amount);

amount++;

System.out.println("Main: " + amount);

}

public void run() {

amount++;

}

}

## Java Lambda Expressions

Lambda Expressions were added in Java 8.

A lambda expression is a short block of code which takes in parameters and returns a value. Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.

## Syntax

The simplest lambda expression contains a single parameter and an expression:

parameter -> expression

To use more than one parameter, wrap them in parentheses:

*(*parameter1*,* parameter2*)* -> expression

Expressions are limited. They have to immediately return a value, they cannot contain variables, assignments or statements such as if or for. In order to do more complex operations, a code block can be used with curly braces. If the lambda expression needs to return a value then the code block should have a return statement.

*(*parameter1*,* parameter2*)* -> { code block }

## Using Lambda Expressions

Lambda expressions are usually passed as parameters to a function:

### **Example**

Use a lamba expression in the ArrayList's forEach() method to print every item in the list:

import java.util.ArrayList;

public class MyClass {

public static void main(String[] args) {

ArrayList<Integer> numbers = new ArrayList<Integer>();

numbers.add(5);

numbers.add(9);

numbers.add(8);

numbers.add(1);

numbers.forEach( (n) -> { System.out.println(n); } );

}

}

[Run example »](https://www.w3schools.com/java/showjava.asp?filename=demo_lambda)

Lambda expressions can be stored in variables if the variable's type is an interface which has only one method. The lambda expression should have the same number of parameters and the same return type as that method. Java has many of these kinds of interfaces built in, such as the Consumer interface (found in the java.util package) used by lists.

### **Example**

Use Java's Consumer interface to store a lambda expression in a variable:

import java.util.ArrayList;

import java.util.function.Consumer;

public class MyClass {

public static void main(String[] args) {

ArrayList<Integer> numbers = new ArrayList<Integer>();

numbers.add(5);

numbers.add(9);

numbers.add(8);

numbers.add(1);

Consumer<Integer> method = (n) -> { System.out.println(n); };

numbers.forEach( method );

}

}

[Run example »](https://www.w3schools.com/java/showjava.asp?filename=demo_lambda2)

To use a lambda expression in a method, the method should have a parameter with a single-method interface as its type. Calling the interface's method will run the lambda expression:

### **Example**

Create a method which takes a lambda expression as a parameter:

interface StringFunction {

String run(String str);

}

public class MyClass {

public static void main(String[] args) {

StringFunction exclaim = (s) -> s + "!";

StringFunction ask = (s) -> s + "?";

printFormatted("Hello", exclaim);

printFormatted("Hello", ask);

}

public static void printFormatted(String str, StringFunction format) {

String result = format.run(str);

System.out.println(result);

}

}