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