**Java Methods**

* A **method** is a block of code which only runs when it is called.
* You can pass data, known as parameters, into a method.
* Methods are used to perform certain actions, and they are also known as **functions**.
* Why use methods? To reuse code: define the code once, and use it many times.

**Create a Method**

A method must be declared within a class. It is defined with the name of the method, followed by parentheses **()**. Java provides some pre-defined methods, such as System.out.println(), but you can also create your own methods to perform certain actions:

public class Main {

static void myMethod() {

// code to be executed

}

}

Example explained

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

**Call a Method**

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

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

public class Main {

static void myMethod() {

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

}

public static void main(String[] args) {

myMethod();

}

}

// Outputs "I just got executed!"

We can call the same method multiple times in a program.

**Java Method Parameters**

**Parameters and Arguments**

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

Parameters are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

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

public class Main {

static void myMethod(String fname, int age) {

System.out.println(fname + " is " + age);

}

public static void main(String[] args) {

myMethod("Liam", 5);

myMethod("Jenny", 8);

myMethod("Anja", 31);

}

}

// Liam is 5

// Jenny is 8

// Anja is 31

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**.

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

**Return Value**

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

public class Main {

static **int** myMethod(int x) { // passed x Parameter

**return** 5 + x;

}

public static void main(String[] args) {

System.out.println(myMethod(3));

}

}

// Outputs 8

**Method Overloading**

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

int myMethod(int x)

float myMethod(float x)

double myMethod(double x, double y)

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

static int plusMethodInt(int x, int y) {

return x + y;

}

static double plusMethodDouble(double x, double y) {

return x + y;

}

public static void main(String[] args) {

int myNum1 = plusMethodInt(8, 5);

double myNum2 = plusMethodDouble(4.3, 6.26);

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

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

}

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

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

static int plusMethod(int x, int y) {

return x + y;

}// method overloading

static double plusMethod(double x, double y) {

return x + y;

}

public static void main(String[] args) {

int myNum1 = plusMethod(8, 5);

double myNum2 = plusMethod(4.3, 6.26);

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

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

}

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

**Java Scope**

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

**Method Scope**

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

Java execute the code line by line

public class Main {

public static void main(String[] args) {

// Code here CANNOT use x

int x = 100;

// Code here can use x

System.out.println(x);

}

}

**Block Scope**

A block of code refers to all of the code between curly braces {}. Variables declared inside blocks of code are only accessible by the code between the curly braces, which follows the line in which the variable was declared:

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

public class Main {

public static void main(String[] args) {

// Code here CANNOT use x

// This is a block

// Code here CANNOT use x

int x = 100;

// Code here CAN use x

System.out.println(x);

// The block ends here

}

}

**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.

**Example**

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

public class Main {

public static void main(String[] args) {

int result = sum(10);// sum function called here

System.out.println(result);

}

public static int sum(int k) {

if (k > 0) {

return k + sum(k - 1);

} else {

return 0;

}

}

}

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

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

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

**Halting condition**

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

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

public class Main {

public static void main(String[] args) {

int result = sum(5, 10);

System.out.println(result);

}

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

if (end > start) {

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

} else {

return end;

}

}

}

**Infinite Recursion**

If the given condition is running infinite the recursive function will stop calling itself. The will call itself again and again hence java will throw an Exception in thread "main" java.lang.StackOverflowError

package OOPs;

public class Recursion {

public static void main(String[] args) {

*r*(); //method called

}

public static void r() {

*r*(); //function calling itself

}

}