Method (الدوال بلغة جافا)

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 Main:

```
public class Main {

  static void myMethod() {

    // code to be executed

  }

}
```

Example Explained

myMethod() is the name of the method.

static means that the method belongs to the Main class and not an object of the Main class. You will learn more about objects and how to access methods through objects later.

void means that this method does not have a return value. You will learn more about return values later.

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

Example : A method can also be called multiple times:

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

output

```
// I just got executed!
// I just got executed!
// I just got executed!
```

```
  }
}
```

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:

Example

```
public class Main {
  static void myMethod(String fname) {
    System.out.println(fname + " Refsnes");
  }
  public static void main(String[] args) {
    myMethod("Liam");
    myMethod("Jenny");
    myMethod("Anja");
  }
}
```

output

```
// Liam Refsnes
// Jenny Refsnes
// Anja Refsnes
```

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

Multiple Parameters

You can have as many parameters as you like:

Example

```
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);
  }
}
```

Output

```
// Liam is 5
// Jenny is 8
// Anja is 31
```

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

Return Values

The **void** keyword, used in the examples above, indicates that the method should not return a value. If you want the method to return a value, you can use a

primitive data type (such as int, char, etc.) **instead of void**, and use the **return** keyword inside the method:

Example

```
public class Main {
  static int myMethod(int x) {
    return 5 + x;
  }
  public static void main(String[] args) {
    System.out.println(myMethod(3));
  }
}
```

Output

```
// Outputs 8 (5 + 3)
```

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

```
public class Main {
  static int myMethod(int x, int y) {
    return x + y;
  }
  public static void main(String[] args) {
    System.out.println(myMethod(5, 3));
  }
}
```

Output

```
// Outputs 8 (5 + 3)
```

You can also store the result in a variable (recommended, as it is easier to read and maintain):

Example

```
public class Main {
  static int myMethod(int x, int y) {
    return x + y;
  }
  public static void main(String[] args) {
    int z = myMethod(5, 3);
    System.out.println(z);
  }
}
```

Output

```
// Outputs 8 (5 + 3)
```

A Method with If...Else

It is common to use if...else statements inside methods:

Example

```
public class Main {
  // Create a checkAge() method with an integer variable called age
  static void checkAge(int age) {
    // If age is less than 18, print " younger "
    if (age < 18) {
      System.out.println("younger ");
```

```
    // If age is greater than, or equal to, 18, print "young"
    } else {
      System.out.println("young");
    }
  }

  public static void main(String[] args) {
    checkAge(20); // Call the checkAge method and pass along an age of 20
  }
}
```

Output

// Outputs “young "

Java Method Overloading

Method Overloading

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

Example

```
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:

Example

```
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);
}
```

Note :**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**:

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);

}
```

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

Java Recursion

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

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

Recursion Example

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

Example

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

```
public class Main {

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

```
    int result = sum(10);
    System.out.println(result);
  }
  public static int sum(int k) {
    if (k > 0) {
      return k + sum(k - 1);
    } else {
      return 0;
    }
  }
}
```

Example Explained

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

10 + sum(9)

10 + (9 + sum(8))

10 + (9 + (8 + sum(7)))

...

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

10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 0

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

Halting Condition

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

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

Example

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

```
public class Main {
  public static void main(String[] args) {
    int result = sum(5, 10);
    System.out.println(result);
  }
  public static int sum(int start, int end) {
    if (end > start) {
      return end + sum(start, end - 1);
    } else {
      return end;
```

```
    }
  }
}
```