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.

Types of Java methods

Depending on whether a method is defined by the user, or available in the standard library, there are two types of methods in Java:

- Standard Library Methods
- User-defined Methods

Standard Library Methods

The standard library methods are built-in methods in Java that are readily available for use. These standard libraries come along with the Java Class Library (JCL) in a Java archive (*.jar) file with JVM and JRE.

Here's a working example:

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

        // using the sqrt() method
        System.out.print("Square root of 4 is: " + Math.sqrt(4));
    }
}
```

User-defined Method

We can also create methods of our own choice to perform some task. Such methods are called user-defined methods.

Here is how we can create a method in Java:

```
public static void myMethod() {
    System.out.println("My Function called");
}
```

Returning a Value from a Method

A method returns to the code that invoked it when it

- completes all the statements in the method,
- reaches a `return` statement, or
- throws an exception (covered later),

whichever occurs first.

You declare a method's return type in its method declaration. Within the body of the method, you use the `return` statement to return the value.

Any method declared `void` doesn't return a value. It does not need to contain a `return` statement, but it may do so. In such a case, a `return` statement can be used to branch out of a control flow block and exit the method and is simply used like this:

```
return;
```

If you try to return a value from a method that is declared `void`, you will get a compiler error.

Any method that is not declared `void` must contain a `return` statement with a corresponding return value, like this:

```
return returnValue;
```

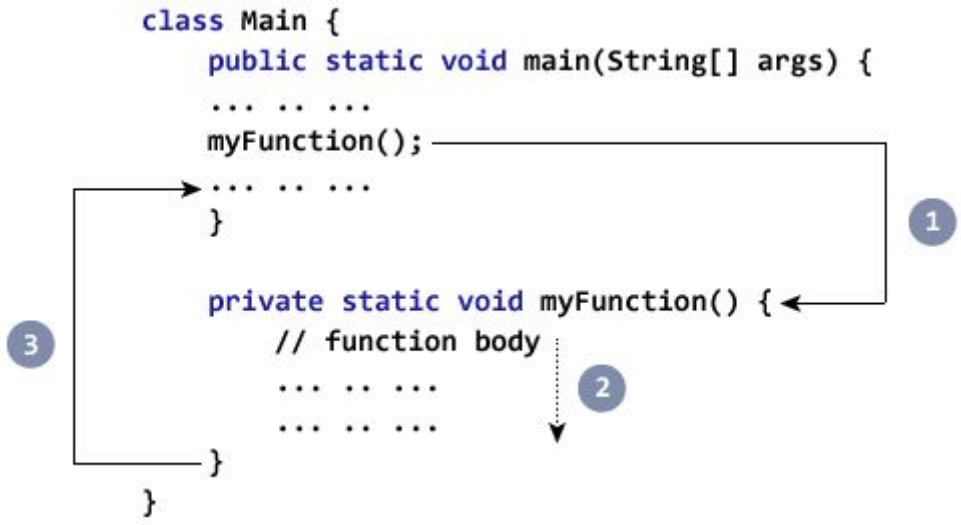
The data type of the return value must match the method's declared return type; you can't return an integer value from a method declared to return a boolean.

The `getArea()` method in the `Rectangle` `Rectangle` class that was discussed in the sections on objects returns an integer:

```
// a method for computing the area of the rectangle
 public int getArea() {
     return width * height;
 }
```

This method returns the integer that the expression `width*height` evaluates to.

How to call a Java Method?



1. While executing the program code, it encounters `myFunction();` in the code.
2. The execution then branches to the `myFunction()` method and executes code inside the body of the method.
3. After the execution of the method body, the program returns to the original state and executes the next statement after the method call.

Method Arguments and Return Value

As discussed earlier, a Java method can have zero or more parameters. And, it may also return some value.

Let's take an example of a method returning a value.

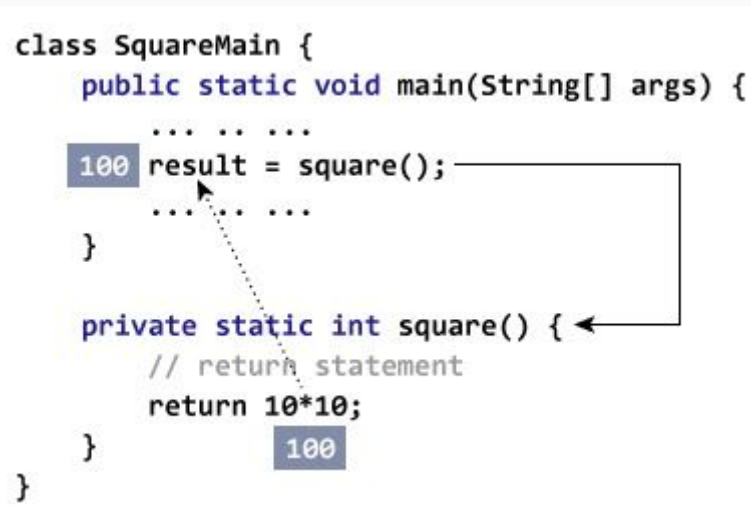
```
class SquareMain {
    public static void main(String[] args) {
        int result;

        // call the method and store returned value
        result = square();
        System.out.println("Squared value of 10 is: " + result);
    }

    public static int square() {
        // return statement
        return 10 * 10;
    }
}
```

Output:

```
Squared value of 10 is: 100
```

In the above program, we have created a method named `square()`. This method does not accept any arguments and returns value `10 *10`.

Here, we have mentioned the return type of the method as `int`. Hence, the method should always return an integer value.



Representation of a method returning a value

As we can see, the scope of this method is limited as it always returns the same value. Now, let's modify the above code snippet so that instead of always returning the squared value of 10, it returns the squared value of any integer passed to the method.

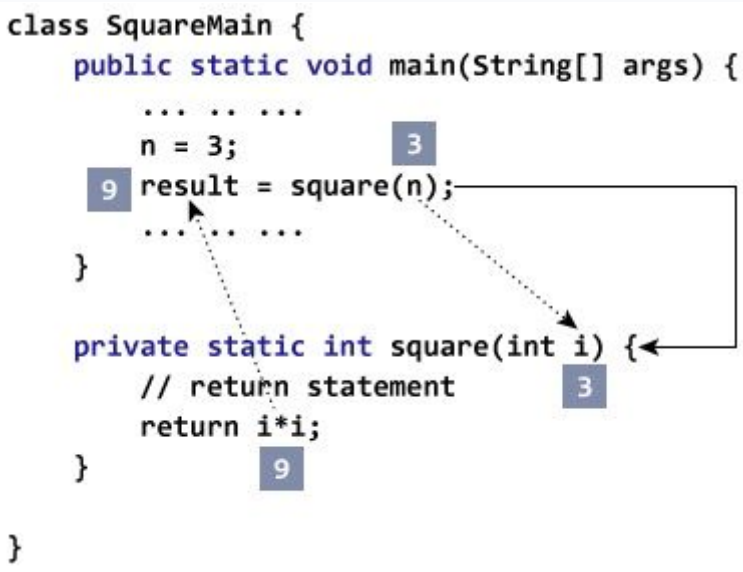
```
public class Main {

    public static void main(String[] args) {
        int result, n;

        n = 3;
        result = square(n);
        System.out.println("Square of 3 is: " + result);

        n = 4;
        result = square(n);
        System.out.println("Square of 4 is: " + result);
    }

    // method
    static int square(int i) {
```

```
        return i * i;
    }
}
```

Output:

```
Squared value of 3 is: 9
Squared value of 4 is: 16
```

Here, the `square()` method accepts an argument `i` and returns the square of `i`. The returned value is stored in the variable `result`.



Passing arguments and returning a value from a method in Java

If we pass any other data type instead of int, the compiler will throw an error. It is because Java is a strongly typed language.

The argument `n` passed to the `getSquare()` method during the method call is called an actual argument.

```
result = getSquare(n);
```

The argument `i` accepted by the method definition is known as a formal argument. The type of formal argument must be explicitly typed.

```
public static int square(int i) {...}
```

We can also pass more than one argument to the Java method by using commas. For example,

```
public class Main {
```

```
    // method definition
    public static int getIntegerSum (int i, int j) {
        return i + j;
    }

    // method definition
    public static int multiplyInteger (int x, int y) {
        return x * y;
    }

    public static void main(String[] args) {

        // calling methods
        System.out.println("10 + 20 = " + getIntegerSum(10, 20));
        System.out.println("20 x 40 = " + multiplyInteger(20, 40));
    }
}
```

Output:

```
10 + 20 = 30
20 x 40 = 800
```

> Note: The data type of actual and formal arguments should match, i.e., the data type of first actual argument should match the type of first formal argument. Similarly, the type of second actual argument must match the type of second formal argument and so on.

What are the advantages of using methods?

1. The main advantage is code reusability. We can write a method once, and use it multiple times. We do not have to rewrite the entire code each time. Think of it as, "write once, reuse multiple times". For example,

```
public class Main {

    // method defined
    private static int getSquare(int x){
        return x * x;
    }

    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
```

```
            // method call
            int result = getSquare(i);
            System.out.println("Square of " + i + " is: " + result);
        }
    }
}
```

Output:

```
Square of 1 is: 1
Square of 2 is: 4
Square of 3 is: 9
Square of 4 is: 16
Square of 5 is: 25
```

In the above program, we have created the method named getSquare() to calculate the square of a number. Here, the same method is used to calculate the square of numbers less than 6.

Hence, we use the same method again and again.

2. Methods make code more readable and easier to debug. For example, `getSquare()` method is so readable, that we can know what this method will be calculating the square of a number.