Methods in Java

teacher: Майер С.Ф.
sfmayer@sfedu.ru

Why Do We Need Methods?

- Imagine you need to calculate the sum of integers from 1 to 10, then from 20 to 37, and finally from 35 to 49.
- Without methods, you would write almost identical code three times:

```
// Sum from 1 to 10
int sum = 0;
for (int i = 1; i <= 10; i++) sum += i;
System.out.println("Sum from 1 to 10 is " + sum);

sum = 0;
for (int i = 20; i <= 37; i++) sum += i;
System.out.println("Sum from 20 to 37 is " + sum);

// Sum from 35 to 49 (again)
sum = 0;
for (int i = 35; i <= 49; i++) sum += i;
System.out.println("Sum from 35 to 49 is " + sum);
```

Key Issue: Code duplication. Hard to maintain and error-prone.

Introducing Methods

Solution: Define a method once and reuse it.

- We create a common piece of code that accepts the starting and ending integers as parameters.
- This approach is **DRY** (Don't Repeat Yourself).

Improved Code:

```
// Method definition (once)
public static int sum(int i1, int i2) {
    int result = 0;
    for (int i = i1; i <= i2; i++) {
        result += i;
    }
    return result;
}
// Method invocations (reused)
public static void main(String[] args) {
    System.out.println("Sum from 1 to 10 is " + sum(1, 10));
    System.out.println("Sum from 20 to 37 is " + sum(20, 37));
    System.out.println("Sum from 35 to 49 is " + sum(35, 49));
}
```

Output:

```
Sum from 1 to 10 is 55
Sum from 20 to 37 is 513
Sum from 35 to 49 is 630
```

Method

- A **method** is a collection of statements grouped together to perform a specific operation.
- **Analogy:** A method is like a **recipe** or a **machine**:
  - You provide **inputs** (parameters).
  - It performs a **task** (body).
  - It may produce an **output** (return value).

Benefits:

- **Reusability:** Write once, use many times.
- **Modularity:** Break complex problems into smaller, manageable pieces.
- **Maintainability:** Fix a bug in one place, not everywhere.

Method Definition Syntax

The syntax for defining a method:

```
modifiers returnType methodName(parameterList) {
    // method body
    // statements to perform the task
    return value; // if returnType is not void
}
```

Components:

- **Modifiers:** public static (for now — controls access).
- **Return Type:** The data type of the value the method returns. Use void if nothing is returned.
- **Method Name:** A valid Java identifier (e.g., sum, max, printGrade).
- **Parameter List:** A comma-separated list of type-name pairs (e.g., int num1, int num2). Can be empty.
- **Method Body:** The block of code that implements the operation.

Anatomy of a Method (Example: max)

Let's examine a method that finds the maximum of two integers.

```
public static int max(int num1, int num2) {
    int result;
    if (num1 > num2) {
        result = num1;
    } else {
        result = num2;
    }
    return result;
}
```

Explanation:

- **Signature:** `max(int num1, int num2)` — the method name and parameter list.
- **Parameters:** num1 and num2 are **formal parameters**.
- **Return:** The method returns an **int** value.
- **Body:** Contains the logic to determine the larger number.

Anatomy of a Method (Example: max)

Let's examine a method that finds the maximum of two integers. But it will not return any value!

```
public static void max(int num1, int num2) {
    if (num1 > num2) {
        System.out.println("max number: " + num1);
    } else {
        System.out.println("max number: " + num2);
    }
}
```

Explanation:

- Use void if nothing is returned.

Method Invocation (Calling a Method)

- To execute a method, you must **invoke** (call) it.
- There are two ways, depending on the return type:
  - **Value-Returning Method:** The call is treated as a value.

```
int larger = max(3, 4);        // Assign result to variable

System.out.println(max(3, 4)); // Use result directly
```

  - **void Method:** The call must be a standalone statement.

```
printGrade(78.5);    // Valid
// int x = printGrade(78.5); // ERROR: no return value
```

Example max: comparison

```java
public class MaxExample1 {
    public static int max(int num1, int num2)
    {
        int result;
        if (num1 > num2) {
            result = num1;
        } else {
            result = num2;
        }
        return result;
    }

    public static void main(String[] args) {
        int a = 10;
        int b = 25;
        int maximum = max(a, b);
        System.out.println("max: " + maximum);

        System.out.println("Max(50,30)=" + max(50,
30));
        System.out.println("Max(7, 7) = " + max(7,
7));
    }
}
```

```java
public class MaxExample2 {
    public static void max(int num1, int num2) {
        if (num1 > num2) {
            System.out.println("max: " + num1);
        } else {
            System.out.println("max: " + num2);
        }
    }

    public static void main(String[] args) {
        int x = 10;
        int y = 25;
        max(x, y);

        max(50, 30);
        max(7, 7);
        max(100, 50);
    }
}
```

Arguments vs. Parameters:

- **Arguments (Actual Parameters):** The values passed during invocation (e.g., 3, 4).
- **Parameters (Formal Parameters):** The variables defined in the method header (e.g., num1, num2).

```
public static int max(int num1, int num2) {
    int result;
    if (num1 > num2) {
        result = num1;
    } else {
        result = num2;
    }
    return result;
}
…
int larger = max(3, 4);
System.out.println(max(3, 4));
```

Parameters (Formal Parameters)

Arguments (Actual Parameters)

Calling a Method

Calling a Method

When a method is invoked, program control transfers to that method.
After the method finishes, control returns to the caller.

Example:

```
public class TestMax {
    public static void main(String[] args) {
        int i = 5;
        int j = 2;
        int k = max(i, j);     // 1. Call max
        System.out.println(k); // 3. Continue here
    }

    public static int max(int num1, int num2) {
        int result;
        if (num1 > num2) result = num1;
        else result = num2;
        return result;         // 2. Return to caller
    }
}
```

The void Method in Action (Example: printGrade)

This method prints a letter grade based on a score (90-100: A, 80-89: B, 70-79: C, 60-69: D, 50-59: F). It returns nothing.

```
For 78.5  the grade is C
For 59.5  The grade is F
```

The void Method in Action (Example: printGrade)

This method prints a letter grade based on a score. It returns nothing.

```
public class TestVoidMethod {
    public static void main(String[] args) {
        System.out.print("The grade is ");
        printGrade(78.5);  // Call as a statement
        System.out.print("The grade is ");
        printGrade(59.5);
    }
    public static void printGrade(double score) {
        if (score >= 90.0) {
            System.out.println('A');
        } else if (score >= 80.0) {
            System.out.println('B');
        } else if (score >= 70.0) {
            System.out.println('C');
        } else if (score >= 60.0) {
            System.out.println('D');
        } else {
            System.out.println('F');
        }
    }
```

For 78.5 the grade is C
For 59.5 The grade is F

The return Statement in void Methods

A void method does not require a `return` statement, but you can use it to exit early. This is useful for validating input.

```
public static void printGrade(double score) {
    if (score < 0 || score > 100) {
        System.out.println("Invalid score");
        return;  // Exit method immediately
    }
    // Only executes if score is valid
    if (score >= 90.0) System.out.println('A');
    else if (score >= 80.0) System.out.println('B');
    else if (score >= 70.0) System.out.println('C');
    else if (score >= 60.0) System.out.println('D');
    else System.out.println('F');
}
```

Output (if called with printGrade(105)**):**

Invalid score

Simple example1

To do: create a method to add an exclamation mark to a string.

Solution 1: Method that returns a value (returns the modified string)

Solution 2: Method that doesn't return a value (void method)

Output:

```
Original: Hello
Modified: Hello!
```

Solution 1

```
public class StringModifier {
    public static void main(String[] args) {
        String original = "Hello";

        // Call the method and get the result
        String result = addExclamationMark(original);

        System.out.println("Original: " + original);
        System.out.println("Modified: " + result);
    }

    public static String addExclamationMark(String text) {
        return text + "!";
    }
}
```

Solution 2

```
public class StringPrinter {
    public static void main(String[] args) {
        String message = "Good morning";

        System.out.println("Before method call: " + message);

        // Call the method — it will print the result directly
        printWithExclamation(message);

        System.out.println("After method call: " + message);
    }

    public static void printWithExclamation(String text) {
        System.out.println(text + "!");
    }
}
```

Example

To do: create a method to calculate factorial.

Solution 1: Method that returns a value

Solution 2: Method that doesn't return a value (void method)

Output:

```
7! = 5040
```

Version 1 (returns long)

```
class FactorialExample {
    // Uses recursion: n! = n * (n-1)!
    public static long factorial(int n) {
        if (n < 0) {
            throw new IllegalArgumentException("Factorial not defined for
negative numbers");
        }
        if (n == 0 || n == 1) {
            return 1;
        }
        return n * factorial(n - 1);
    }

    public static void main(String[] args) {
        int number = 7;
        long result = factorial(number);
        System.out.println(number + "! = " + result); // Output: 7! = 5040
    }
}
```

Or without recursion

```
long result = 1;
        for (int i = 2; i <= n; i++) {
            result *= i;
        }
        System.out.println(n + "! = " +
result);
```

Version 2 (void — prints result with validation)

```
class FactorialVoid {
    // Prints factorial result directly, includes input validation
    public static void calculateFactorial(int n) {
        if (n < 0) {
            System.out.println("Error: Factorial not defined for negative numbers");
            return;
        }

        long result = 1;
        for (int i = 2; i <= n; i++) {
            result *= i;
        }
        System.out.println(n + "! = " + result);
    }

    public static void main(String[] args) {
        calculateFactorial(5);  // Output: 7! = 5040
        calculateFactorial(-1); // Output: Error: Factorial not defined...
    }
}
```

Pass-by-Value (Primitive Types)

- **Java uses pass-by-value** for primitive types.
- This means a **copy** of the argument's value is passed to the parameter.
- The method works with a copy; the original variable is **never modified**.
- **Example:**

```
public class Increment {
    public static void main(String[] args) {
        int x = 1;
        increment(x);
        System.out.println(x);  // Still 1
    }
    public static void increment(int n) {
        n++;  // n is a copy
        System.out.println("Inside method: " + n);
    }
}
```

Inside method: 2
1

Method Overloading(Example: max)

Let's examine a method that finds the maximum of two integers.

```
public static int max(int num1, int num2) {
    int result;
    if (num1 > num2) {
        result = num1;
    } else {
        result = num2;
    }
    return result;
}
```

- Only **integers** are possible.
- Only **two** integers are possible

Method Overloading

Overloading allows multiple methods with the **same name** but **different parameter lists**.
The compiler determines which method to call based on the arguments.

Example: Create method to find the maximum value of two integers, two real numbers, and three integers.

Output:

```
max(3, 4) = 4
max(3.0, 5.4)= 5.4
max(3.0, 5.4, 10.14) = 10.14
```

Rules of method overloading:

- Methods must have different parameter lists (number, type, or order).
- Cannot overload based only on return type or modifiers.

Method Overloading

Example: Create method to find the maximum value of two integers, two real numbers, and three integers.

```
public class TestOverloading {
    public static void main(String[] args) {
        System.out.println(max(3, 4));        // Calls max(int, int)
        System.out.println(max(3.0, 5.4));    // Calls max(double, double)
        System.out.println(max(3.0, 5.4, 10.14)); // Calls max(double, double, double)
    }
    public static int max(int num1, int num2) {
        return (num1 > num2) ? num1 : num2;
    }
    public static double max(double num1, double num2) {
        return (num1 > num2) ? num1 : num2;
    }
    public static double max(double num1, double num2, double num3) {
        return max(max(num1, num2), num3);
    }
}
```

Output:
4
5.4
10.14

Variable Scope

Scope is the part of the program where a variable can be accessed.

Rules:

A **local variable** is declared inside a method. Its scope begins at its declaration and ends at the closing brace of the block.

A **parameter** has scope throughout the entire method.

A variable declared in a for **loop header** has scope only within the loop.

Variable Scope

Common Mistake:

```
for (int i = 0; i < 10; i++) {
   // i is accessible here
}
System.out.println(i);  // ERROR: i is out of scope
```

Valid Example:

```
public static void example() {
    int x = 10;  // Scope: entire method
    if (x > 5) {
        int y = 20;  // Scope: only inside the if block
        System.out.println(y);
    }
    // System.out.println(y); // ERROR: y out of scope
}
```

Key Takeaways & Summary

1. **Methods** are reusable blocks of code that perform specific tasks. They make programs modular and maintainable.
2. **Method signature** consists of the method name and parameter list. It uniquely identifies the method.
3. **Value-returning methods** produce a result; void methods perform actions without returning a value.
4. **Pass-by-value** means primitive arguments are copied; the original is never changed.
5. **Method overloading** allows multiple methods with the same name but different parameters. Avoid ambiguous overloads.
6. **Variable scope** determines where a variable can be accessed. Local variables exist only within their declaring block.

Final Thought:

Methods are the building blocks of structured programming. Mastering them is essential for writing clean, efficient, and professional Java code.