# Methods in Java

## What is a Method?

A method in Java is a block of code that performs a specific task. Think of a method as a subprogram designed to execute a particular operation, like calculating a sum, printing a message, or sorting a list. Methods help to break down complex problems into smaller, manageable parts.

### Key Concepts:

- **Modularity:** Methods allow you to organize code into logical sections, making it easier to read, maintain, and debug.
- **Code Reusability:** Once a method is defined, you can use it multiple times without rewriting the code, enhancing efficiency.
- **Parameterization:** Methods can accept inputs (parameters) and return outputs, making them versatile for various operations.

## Method Declaration and Definition

A method in Java is declared and defined using the following structure:

```
returnType methodName(parameterList) {
    // Method body
}
```

- **returnType:** Specifies the type of value the method returns. If the method does not return any value, the return type is `void`.
- **methodName:** The name of the method, which should be meaningful and follow Java naming conventions (e.g., camelCase).
- **parameterList:** A list of input parameters, enclosed in parentheses and separated by commas. If no parameters are needed, leave the parentheses empty.

### Example of a Simple Method

```
class SimpleMethod {
    // Method declaration and definition
    public void greet() {
        System.out.println("Hello, welcome to Java!");
    }

    public static void main(String[] args) {
        SimpleMethod obj = new SimpleMethod(); // Creating an object of the class
        obj.greet(); // Calling the method
    }
}
```

**Expected Output:**

```
Hello, welcome to Java!
```

In this example:

- The method `greet` is declared with the return type `void`, meaning it doesn't return any value.
- The method simply prints a welcome message to the console when called.

## Method Parameters and Return Types

Methods can be designed to accept inputs (parameters) and produce outputs (return values). This allows methods to perform a wide range of tasks based on the inputs provided.

### Example with Parameters and Return Type

```java
class Addition {
    // Method with parameters and return type
    public int add(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        Addition obj = new Addition();
        int sum = obj.add(10, 20); // Passing arguments and storing the result
        System.out.println("Sum: " + sum);
    }
}
```

**Expected Output:**

```
Sum: 30
```

In this example:

- The method `add` accepts two integer parameters, `a` and `b`, and returns their sum.
- The return type of the method is `int`, indicating that the method returns an integer value.

## Method Overloading

Method overloading allows you to define multiple methods with the same name but different parameter lists. This is useful when you want the same method to handle different types of inputs.

### Example of Method Overloading

```
class MethodOverloading {
    // Method with two int parameters
    public int add(int a, int b) {
        return a + b;
    }

    // Overloaded method with three int parameters
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // Overloaded method with two double parameters
    public double add(double a, double b) {
        return a + b;
    }

    public static void main(String[] args) {
        MethodOverloading obj = new MethodOverloading();
        System.out.println("Sum of 10 and 20: " + obj.add(10, 20));
        System.out.println("Sum of 10, 20, and 30: " + obj.add(10, 20, 30));
        System.out.println("Sum of 10.5 and 20.5: " + obj.add(10.5, 20.5));
    }
}
```

**Expected Output:**

```
Sum of 10 and 20: 30
Sum of 10, 20, and 30: 60
Sum of 10.5 and 20.5: 31.0
```

In this example:

- The `add` method is overloaded to handle different types and numbers of parameters.
- The appropriate method is selected based on the arguments passed during the method call.

## Recursion in Java

Recursion is a technique where a method calls itself to solve a problem. Recursive methods are often used to solve problems that can be broken down into smaller, similar sub-problems, such as calculating a factorial, generating Fibonacci numbers, or solving the Tower of Hanoi problem.

### Example of Recursion: Calculating Factorial

```
class Factorial {
    // Recursive method to calculate factorial
    public int factorial(int n) {
        if (n == 0) {
            return 1; // Base case: factorial of 0 is 1
        } else {
            return n * factorial(n - 1); // Recursive call
        }
    }

    public static void main(String[] args) {
        Factorial obj = new Factorial();
        int result = obj.factorial(5); // Calculating factorial of 5
        System.out.println("Factorial of 5 is: " + result);
    }
}
```

**Expected Output:**

```
Factorial of 5 is: 120
```

### Explanation:

- **Base Case:** The simplest, smallest instance of the problem. In the case of factorial, the base case is when `n` is 0, and the factorial is 1.
- **Recursive Call:** The method calls itself with a modified argument, working towards the base case. Here, the method calls itself with `n-1` until `n` becomes 0.

### Example of Recursion: Fibonacci Series

```
class Fibonacci {
    // Recursive method to find nth Fibonacci number
    public int fibonacci(int n) {
        if (n == 0) {
            return 0; // Base case 1: Fibonacci(0) is 0
        } else if (n == 1) {
            return 1; // Base case 2: Fibonacci(1) is 1
        } else {
            return fibonacci(n - 1) + fibonacci(n - 2); // Recursive call
        }
    }

    public static void main(String[] args) {
        Fibonacci obj = new Fibonacci();
        int n = 7; // Find the 7th Fibonacci number
        int result = obj.fibonacci(n);
        System.out.println("Fibonacci number at position " + n + " is: " +
result);
    }
}
```

**Expected Output:**

```
Fibonacci number at position 7 is: 13
```

## Practice Questions

1. **Problem:** Implement a recursive method to solve the Tower of Hanoi problem for `n` disks.

   **Solution:**

   ```java
   class TowerOfHanoi {
       // Recursive method to solve Tower of Hanoi
       public void solve(int n, char fromRod, char toRod, char auxRod) {
           if (n == 1) {
               System.out.println("Move disk 1 from " + fromRod + " to " +
   toRod);
               return;
           }
           solve(n - 1, fromRod, auxRod, toRod); // Move n-1 disks from fromRod
   to auxRod
           System.out.println("Move disk " + n + " from " + fromRod + " to " +
   toRod);
           solve(n - 1, auxRod, toRod, fromRod); // Move n-1 disks from auxRod
   to toRod
       }

       public static void main(String[] args) {
           TowerOfHanoi obj = new TowerOfHanoi();
           int n = 3; // Number of disks
           obj.solve(n, 'A', 'C', 'B'); // A, B, C are rod names
       }
   }
   ```

   **Expected Output:**

   ```
   Move disk 1 from A to C
   Move disk 2 from A to B
   Move disk 1 from C to B
   Move disk 3 from A to C
   Move disk 1 from B to A
   Move disk 2 from B to C
   Move disk 1 from A to C
   ```

2. **Problem:** Write a recursive method to reverse a string.

   **Solution:**

```
class ReverseString {
    // Recursive method to reverse a string
    public String reverse(String str) {
        if (str.isEmpty()) {
            return str; // Base case: an empty string is already reversed
        } else {
            return reverse(str.substring(1)) + str.charAt(0); // Recursive
call
        }
    }

    public static void main(String[] args) {
        ReverseString obj = new ReverseString();
        String original = "hello";
        String reversed = obj.reverse(original);
        System.out.println("Original string: " + original);
        System.out.println("Reversed string: " + reversed);
    }
}
```

**Expected Output:**

```
Original string: hello
Reversed string: olleh
```

3. **Problem:** Write a method to calculate the Greatest Common Divisor (GCD)

of two numbers using recursion.

**Solution:**

```
class GCD {
    // Recursive method to calculate GCD
    public int gcd(int a, int b) {
        if (b == 0) {
            return a; // Base case: GCD(a, 0) is a
        } else {
            return gcd(b, a % b); // Recursive call
        }
    }

    public static void main(String[] args) {
        GCD obj = new GCD();
        int a = 56;
        int b = 98;
        int result = obj.gcd(a, b);
        System.out.println("GCD of " + a + " and " + b + " is: " + result);
    }
}
```

**Expected Output:**

```
GCD of 56 and 98 is: 14
```

## Summary

Methods in Java are fundamental to writing organized, reusable, and efficient code. They help to break down complex tasks into simpler, manageable parts, making your code easier to read and maintain. Understanding how to declare, define, and call methods, as well as how to work with parameters and return types, is crucial for effective Java programming. Recursion is a powerful tool for solving problems that can be broken down into smaller, similar sub-problems.