Java Method

In Java, **methods** are blocks of code that perform specific tasks. They help with code reusability and modularity.

🔹 Method Declaration (Definition)

The **syntax** for declaring a method in Java:

```
java

modifier returnType methodName(parameters) {
    // method body
}
```

✅ Example:

```
java

public int addNumbers(int a, int b) {
    int sum = a + b;
    return sum;
}
```

Breakdown:

- **`public`** → Access modifier (who can access the method)
- **`int`** → Return type (what the method returns)
- **`addNumbers`** → Method name
- **`(int a, int b)`** → Parameters (inputs)
- **`return sum;`** → Output of the method

🔹 Method Calling (Invocation)

Once a method is defined, you can call it using the object of the class or directly if it's **`static`**.

✅ Example 1: Calling a Non-Static Method

```
java

public class Calculator {

    public int addNumbers(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        Calculator obj = new Calculator();  // Creating object
        int result = obj.addNumbers(5, 3);  // Calling method
```

```
        System.out.println("Sum: " + result);
    }
}
```

✅ Example 2: Calling a Static Method

```
java

public class Calculator {

    public static int multiplyNumbers(int a, int b) {
        return a * b;
    }

    public static void main(String[] args) {
        int result = multiplyNumbers(4, 6);  // No object needed for static method
        System.out.println("Product: " + result);
    }
}
```

🔹 Types of Methods in Java

1. **Predefined Methods** – Already available (e.g., `System.out.println()`)
2. **User-defined Methods** – Created by the user

🔹 Return Types

- `void` – Returns nothing
- `int`, `String`, `double`, `boolean`, etc. – Returns specific data types
- Can also return objects or arrays

Would you like a lab-based practice or quiz to test method concepts?

✅ Method with Arguments and Return Type in Java

A method with **arguments** takes input values (called **parameters**) and a **return type** means it sends back a value to the caller.

🔹 Syntax:

```
java

returnType methodName(dataType param1, dataType param2, ...) {
    // method logic
    return value;
}
```

✅ Example: Method with Arguments and Return Type

```java
public class Calculator {

    // Method with arguments and return type
    public int add(int a, int b) {
        int sum = a + b;
        return sum;  // returning an int value
    }

    public static void main(String[] args) {
        Calculator calc = new Calculator();  // create object
        int result = calc.add(10, 20);       // pass arguments
        System.out.println("Sum: " + result);  // print returned value
    }
}
```

🔹 Explanation:

- **`int add(int a, int b)`**
  → This method takes **two int arguments** and returns an **int result**.
- **`return sum;`**
  → Sends the result back to the caller.
- **`calc.add(10, 20);`**
  → Calling the method with values 10 and 20 as arguments.

🧠 Other Examples:

✅ String method example:

```java
public String greet(String name) {
    return "Hello, " + name + "!";
}
```

✅ Boolean return type example:

```java
public boolean isEven(int number) {
    return number % 2 == 0;
}
```

🔹 Summary:

Part	Example
Method Name	**`add`**
Arguments	**`int a, int b`**
Return Type	**`int`**

Part	Example
Call Statement	`calc.add(10, 20)`
Return Value	`30`

Would you like practice questions or hands-on tasks on this topic?

✅ Method Overloading in Java

Method Overloading means defining **multiple methods** in the same class **with the same name** but **different parameter lists** (number, type, or order of parameters).

It allows you to perform a similar operation in different ways.

🔹 Why Use Method Overloading?

- Improves **readability**
- Supports **polymorphism**
- Avoids creating multiple method names for similar operations

🔹 Rules of Method Overloading:

1. Methods **must have the same name**
2. They **must differ** in:
    - Number of parameters, or
    - Type of parameters, or
    - Order of parameters (only if types are different)

✅ Example:

```
java

public class Calculator {

    // Method with 2 int parameters
    public int add(int a, int b) {
        return a + b;
    }

    // Method with 3 int parameters
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method with 2 double parameters
    public double add(double a, double b) {
        return a + b;
    }

    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println("Sum 1: " + calc.add(10, 20));           // int + int
        System.out.println("Sum 2: " + calc.add(10, 20, 30));       // int + int + int
```

```
        System.out.println("Sum 3: " + calc.add(10.5, 20.3));        // double + double
    }
}
```

🔹 Output:

```
yaml

Sum 1: 30
Sum 2: 60
Sum 3: 30.8
```

❌ Invalid Overloading:

You **cannot overload** a method by only changing the return type.

```
java

// Invalid
public int add(int a, int b) { ... }
public double add(int a, int b) { ... } // ❌ Error: same method signature
```

🔹 Real-World Analogy:

Think of **`print()`** in Java – it's overloaded in the **`System.out`** class:

```
java

System.out.println("Hello");     // prints String
System.out.println(10);          // prints int
System.out.println(10.5);        // prints double
```

Would you like to try a coding exercise or quiz on method overloading?

✅ `main()` Method in Java

The **`main()` method** is the **entry point** of any standalone Java application. When you run a Java program, the JVM (Java Virtual Machine) looks for the **`main()`** method to begin execution.

🔹 Standard Syntax:

```
java

public static void main(String[] args) {
    // your code here
}
```

🔹 Explanation of Each Keyword:

Keyword	Meaning
`public`	Access modifier — JVM must be able to call it from outside the class
`static`	Allows JVM to call the method **without creating an object**
`void`	Return type — it doesn't return any value
`main`	The **name** recognized by JVM as the starting point
`String[] args`	Command-line arguments passed as a **String array**

✅ Example:

```
java


public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, Java!");
    }
}
```

Output:

```


Hello, Java!
```

🔹 What is `String[] args`?

It allows your program to accept **command-line arguments**.

Example run:

```
bash


java HelloWorld Hello GPT
```

Then:

```
java


System.out.println(args[0]);  // prints: Hello
System.out.println(args[1]);  // prints: GPT
```

🔹 Alternative (but valid) forms:

```
java


public static void main(String args[]) { }
public static void main(String... args) { }  // varargs
```

❗ Important Notes:

- You **cannot** change the method signature if you want JVM to recognize it as the starting point.
- A Java class **without a `main()` method** will throw a **"Main method not found" error** during execution (unless it's part of a framework like JUnit, Spring, etc.).

Would you like an example with command-line input or deeper explanation of static, args, or access modifiers?