Functions/Methods

Difference Between Methods and Functions in Java

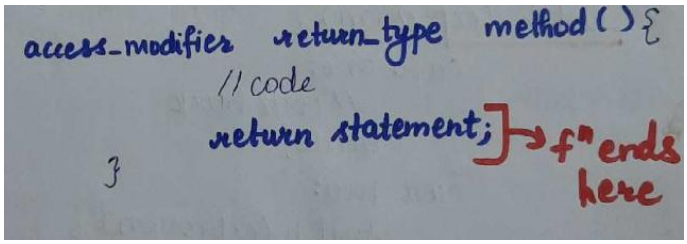
In Java, the terms **method** and **function** are sometimes used interchangeably, but they have distinct meanings in programming.

Feature	Method	Function
Definition	A method is a block of code associated with an object that performs a specific task.	A function is a block of reusable code that performs a specific task, which may or may not be associated with an object.
Belongs To	Always belongs to a class (i.e., methods are defined within a class in Java).	Functions can exist independently in some languages (like Python, C, etc.), but in Java, every function is a method because Java is an object-oriented language.
Call Type	A method is called on an object (e.g., `object.methodName()`).	A function can be called independently (e.g., `functionName()`, but in Java, this applies only to `static` methods).
Usage in Java	All functions in Java are methods since Java does not support standalone functions.	Java does not support standalone functions like C/C++, so all functions must be part of a class (as methods).
Static vs Instance	Can be `static` (belonging to the class) or non-static (belonging to an object).	In Java, functions are always methods; they can be `static` (class-level) or instance methods.
Example	`java class Demo { void show() { System.out.println("This is a method"); } }`	In Java, this is also a method. But in C: `c int add(int a, int b) { return a + b; }`

- In Java, all functions are methods because they must be defined inside a class.
- A method is associated with an object, while a function can be independent in some other programming languages.
- Java does not support standalone functions like C, Python, or JavaScript.
- A method can be static (class-level) or instance-based (object-level).

Methods in Java

A **method** in Java is a block of code that performs a specific task. It helps in code reusability and improves modularity in a program.



1. Syntax of a Method in Java

A method in Java consists of the following components:

1. **Access Modifier** – Defines the visibility of the method (public, private, protected, or default).
2. **Return Type** – Specifies the data type of the value returned by the method (e.g., int, double, void).
3. **Method Name** – Follows Java naming conventions (camelCase).
4. **Parameter List** – Contains input values enclosed in parentheses () (optional).
5. **Method Body** – Contains statements defining the method's behavior.
6. **Return Statement** – Returns a value if the method has a return type other than void.

2. Types of Methods in Java

(a) Predefined Methods (Built-in Methods)

These are methods provided by Java libraries, such as length(), substring(), equals(), etc.

(b) User-Defined Methods

These are methods created by the user to perform specific tasks.

(c) Static Methods

- Declared using the static keyword.
- Belongs to the class rather than an instance.
- Can be called without creating an object of the class.

(d) Instance Methods

- Requires an object of the class to be called.
- Does not use the static keyword.

(e) Method Overloading

- Multiple methods with the same name but different parameter lists.
- Allows the same method name to perform different functions based on the number or type of arguments.

(f) Method Overriding

- When a subclass provides a specific implementation of a method already defined in its parent class.
- Uses the @Override annotation for better readability.

3. Return Types in Java Methods

A method can have different return types:

- **void** – No return value.
- **Primitive Data Types** – int, double, boolean, etc.
- **Reference Data Types** – String, Array, Object, etc.

4. Calling a Method

(a) Calling an Instance Method

- Requires creating an object of the class.
- Called using objectName.methodName().

(b) Calling a Static Method

- Can be called directly using ClassName.methodName().
- No need to create an object.

---

5. Method Parameters

(a) Method with Parameters

- Takes input values and performs operations based on them.

(b) Method with Return Value

- Returns a value to the calling function.

(c) Pass by Value in Java

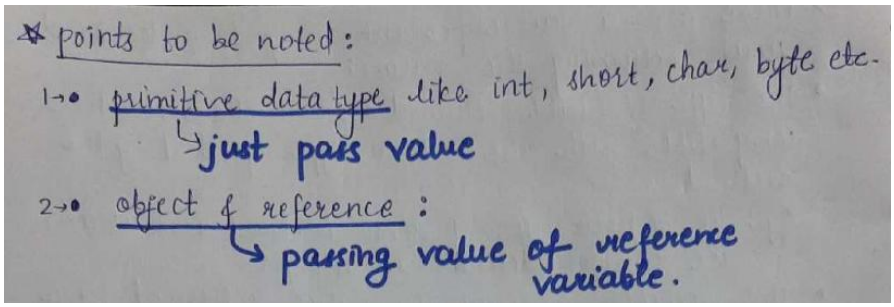
- Java uses **pass by value**, meaning that a copy of the variable is passed to the method.
- Changes made inside the method do not affect the original value outside the method.

6. Main Method in Java

- The main() method is the entry point of every Java program.
- It has the following signature: public static void main(String[] args).
- The String[] args parameter allows command-line arguments to be passed.

7. Importance of Methods in Java

- **Code Reusability** – Reduces duplication by reusing code.
- **Modularity** – Divides the program into smaller, manageable parts.
- **Improved Readability** – Makes the code more organized and easy to understand.
- **Easier Debugging** – Errors can be easily traced within a method.



Scope in Java

Scope in Java refers to the visibility and accessibility of variables, methods, and classes within different parts of a program. It defines where a variable can be used and for how long it exists in memory.

Types of Scope

(a) Local Scope

- A variable declared inside a method, constructor, or block is called a **local variable**.
- It can only be accessed within the block where it is declared.
- It gets destroyed once the block execution is complete.
- Cannot have access modifiers like public, private, or protected.

(b) Instance Scope (Object-Level Scope)

- Instance variables are declared inside a class but outside any method.

- These variables belong to an **instance of the class** and are created when the object is instantiated.
- Each object has its own copy of instance variables.
- Can be accessed using this keyword inside the class.

(c) Static Scope (Class-Level Scope)

- Static variables are declared using the static keyword inside a class.
- They belong to the **class itself**, not any specific instance.
- Shared among all objects of the class.
- Accessed using the class name (ClassName.variableName).

(d) Block Scope

- A variable declared within {} inside an if, for, while, or any other block is only accessible within that block.
- The variable gets destroyed once the block execution is completed.

(e) Method Scope

- Method parameters are only accessible within the method they are declared in.
- They are destroyed when the method execution completes.

2. Variable Shadowing

Shadowing occurs when a variable in a local scope (e.g., inside a method or block) has the **same name** as a variable in an outer scope (e.g., instance or class level). The local variable temporarily **hides** (shadows) the outer variable, making it inaccessible within that scope.

Rules of Shadowing

1. **The local variable takes precedence** over the instance or class variable.
2. **Shadowing does not change the value of the original variable** outside the local scope.
3. **Can be resolved using the this keyword** to access the instance variable when shadowed by a local variable.
4. **Static variables can be accessed using the class name** to avoid shadowing.

1. Local Variable Shadowing an Instance Variable

When a local variable inside a method has the same name as an instance variable, the local variable takes precedence.

Example:

```
class Test {
   int x = 10;  // Instance variable
   void display() {
      int x = 20; // Local variable shadows the instance variable
      System.out.println(x); // Prints 20 (local variable takes precedence)
   }
}
```

🔹 **Explanation:**

- x = 20 inside the display() method **shadows** the instance variable x = 10.
- The instance variable x is still **unchanged**, and only the local x is used inside the method.

Solution to Access Instance Variable: Use this keyword

```
System.out.println(this.x); // Prints 10
```

2. Local Variable Shadowing a Class (Static) Variable

If a local variable inside a method has the same name as a static variable, the local variable shadows the static variable.

Example:

```
class Test {
   static int x = 100;  // Static variable
   void display() {
      int x = 200; // Local variable shadows static variable
      System.out.println(x); // Prints 200
   }
}
```

Solution to Access Static Variable: Use Class Name

System.out.println(Test.x); // Prints 100

3. Method Parameter Shadowing an Instance Variable

If a method parameter has the same name as an instance variable, it shadows the instance variable within that method.

Example:

```
class Test {
   int x = 30; // Instance variable
   void setX(int x) { // Parameter x shadows instance variable x
      x = x; // This does nothing because both refer to the local variable
   }
   void printX() {
      System.out.println(x); // Prints 30 (instance variable remains unchanged)
   }
}
```

Issue: x = x; is assigning the parameter x to itself, leaving the instance variable unchanged.

Solution: Use this keyword

this.x = x; // Assigns method parameter to instance variable

4. Shadowing Inside an Inner Class (Nested Class Shadowing Outer Class Variable)

An inner class can define a variable with the same name as an outer class, which shadows the outer class variable inside the inner class.

Example:

```
class Outer {
   int x = 50; // Outer class variable
   class Inner {
      int x = 60; // Inner class variable shadows outer class variable
```

```
    void display() {
        System.out.println(x); // Prints 60 (inner class variable)
        System.out.println(Outer.this.x); // Prints 50 (access outer class variable)
    }
  }
}
```

Solution to Access Outer Class Variable: Use Outer.this.x

5. Shadowing in Static Context (Static Variable Shadowing Class-Level Variable)

Static methods cannot access instance variables directly, so if a static variable has the same name, it will shadow instance variables.

Example:

```
class Test {
  int x = 70; // Instance variable
  static int x = 80; // Static variable shadows instance variable (Compilation Error)
}
```

Compilation Error: Static and instance variables **cannot have the same name** in the same class.

Solution: Use different names for static and instance variables.

6. Block Scope Shadowing a Variable from an Outer Scope

A variable declared inside a block (if, for, etc.) shadows the variable declared outside the block.

Example:

```
class Test {
  int x = 90; // Instance variable

  void display() {
    System.out.println(x); // Prints 90 (instance variable)
```

```
        {
            int x = 100; // Block-scoped variable shadows instance variable
            System.out.println(x); // Prints 100 (block variable takes precedence)
        }

        System.out.println(x); // Prints 90 (block variable is out of scope)
    }
}
```

- **Explanation:**
  - Inside the block {}, x = 100 shadows the instance variable.
  - Outside the block, the instance variable x = 90 is accessible again.

Varargs (Variable Arguments)

1. Introduction

- Varargs (Variable Arguments) allow a method to accept **a variable number of arguments** of the same type.
- Introduced in **Java 5**.
- Declared using **three dots (...)** after the data type.
- Helps in reducing method overloading when multiple versions of a method handle different argument counts.

2. Syntax

```
returnType methodName(dataType... variableName) {
   // Method body
}
```

- The **three dots (...)** indicate that the method can take **zero or more arguments** of the specified data type.
- Internally, Java **treats varargs as an array**.

3. Key Rules for Using Varargs

Rule 1: Only One Varargs Parameter Allowed

- A method cannot have multiple varargs parameters.

Invalid:

```
void test(int... a, String... b) { }  // Compilation error
```

Valid:

```
void test(int... a) { }  // Works fine
```

Rule 2: Varargs Must Be the Last Parameter

- If a method has **both regular parameters and varargs**, the varargs **must come last**.

Valid:

```
void display(String name, int... numbers) { }  // Works fine
```

Invalid:

```
void display(int... numbers, String name) { }  // Compilation error
```

Rule 3: Can Pass No Arguments, One Argument, or Multiple Arguments

- A varargs method can be called with:
    - **No arguments** → methodName();
    - **One argument** → methodName(10);
    - **Multiple arguments** → methodName(10, 20, 30);

4. Use Cases of Varargs

Use Case 1: Sum of Numbers

- Instead of defining multiple overloaded methods, we can use varargs to handle a variable number of arguments.

```
int sum(int... numbers) {
   int total = 0;
   for (int num : numbers) {
      total += num;
   }
   return total;
}
```

Can be called as:

```
sum();            // Returns 0
sum(5);           // Returns 5
sum(5, 10, 15);   // Returns 30
```

Use Case 2: Finding Maximum Number

- We can find the **maximum value** using varargs.

```
int findMax(int... numbers) {
   int max = Integer.MIN_VALUE;
   for (int num : numbers) {
      if (num > max) {
         max = num;
      }
```

```
    return max;
}
```

Can be called as:

```
findMax(1, 2, 3, 10, 5);  // Returns 10
```

Use Case 3: String Formatting

- Varargs can be used for handling multiple **string inputs** dynamically.

```
void printMessages(String... messages) {
    for (String msg : messages) {
        System.out.println(msg);
    }
}
```

Can be called as:

```
printMessages("Hello", "Welcome", "Java");
```

5. When to Use Varargs?

- When the **number of arguments is unknown or varies**.
- When we **want to reduce method overloading**.
- When we need a **flexible method that can accept multiple values**.

6. Performance Considerations

- Varargs **creates an array internally**, which may cause **performance overhead** when handling large data.
- If the number of arguments is **fixed**, prefer **normal method parameters** for better efficiency.

7. Summary

Feature	**Details**
Declared using	dataType... variableName
Internal representation	Treated as an **array**
Number of parameters	**Zero, one, or multiple** arguments
Position in method	Must be the **last parameter**
Overloading alternative	Reduces method overloading complexity

8. Conclusion

Varargs simplify method design by allowing **variable-length arguments**, reducing method overloading. However, they should be **used carefully** to avoid performance issues due to internal array creation.

Performance Issues with Varargs

Varargs (int... args) let you pass multiple values into a method, but they create an **array** behind the scenes. This can lead to **performance issues**, especially if the method is called frequently or with large data. Here's why:

---

1. Extra Memory Usage

Whenever you call a varargs method, Java **creates a new array** to store the arguments. This can **waste memory** if the method is called many times.

Example Problem:
If a method with varargs is called **a million times**, **a million arrays** will be created, which increases memory usage and slows down the program.

---

2. Slower Performance for Large Inputs

Since varargs store arguments in an array, **iterating over large data takes time**. If your method processes **thousands of numbers**, it will **take longer** compared to using a normal fixed parameter.

Problem:
If you pass **10,000 numbers** to a varargs method, Java needs to:

1. Create an **array of 10,000 elements**
2. Iterate through all **10,000 numbers**

This takes **more time** compared to a method that takes a **pre-existing array or list**.

---

3. More CPU Work

Every time you call a varargs method, Java **allocates memory** for a new array and **manages garbage collection** (removing unused memory). This **increases CPU usage**, making your program **slower** if used too often.

---

4. Not Always Safe

Varargs allow **any number of arguments**, which can sometimes cause **unexpected issues** at runtime.

Example Problem:

```
void printValues(int... values) {
    System.out.println(Arrays.toString(values));
}


printValues(null);  //  Error: NullPointerException
```

- The above code crashes because null cannot be stored in a varargs array.

How to Improve Performance?

1. Avoid Calling Varargs in Loops

Instead of:

```
for (int i = 0; i < 1000000; i++) {
    processNumbers(1, 2, 3, 4, 5);  // Creates 1 million arrays!
}
```

Use:

```
int[] data = {1, 2, 3, 4, 5};
for (int i = 0; i < 1000000; i++) {
    processNumbers(data);  // Reuses 1 array
}
```

- **Why?** This avoids creating unnecessary arrays.

---

2. Use Fixed Parameters If You Know the Argument Count

Bad Practice: (Unnecessary varargs)

```
void sum(int... numbers) { }
```

Better Approach: (Faster, no extra array)

```
void sum(int a, int b, int c) { }
```

- **Why?** No array creation = **faster performance**.

---

3. Use Lists Instead of Varargs for Large Data

Bad Practice:

```
void process(int... numbers) { }  // Creates new array every time
```

Better Approach:

```
void process(List<Integer> numbers) { }  // No extra array
```

- **Why?** Lists are **faster** for handling large data dynamically.

Summary: When to Use and Avoid Varargs

Scenario	Use Varargs?	Better Alternative
Small number of arguments	Yes	-
Large number of arguments	No	List<T> or fixed array
Performance-sensitive code	No	Fixed parameters
Unknown number of inputs	Yes	-

Final Thoughts

- Varargs are **useful but can slow down performance** when used incorrectly.
- They create **new arrays every time** they are called, which increases **memory usage and CPU workload**.
- **Best Practice:** Use varargs **only for small, flexible arguments**. For large data, **use lists or arrays**.

What happens if you modify an instance variable inside a method?

- If a method modifies an **instance variable**, the change is **reflected across all method calls** because instance variables belong to the object.
- However, if the variable is passed as a method parameter, only a **copy** of the value is changed (pass-by-value concept).