Java Methods

What Are Methods in Java?

- A **method** is a block of code that performs a **specific task**.
- Methods help us **reuse code**, improve **readability**, and make programs **modular**.
- In Java, **all methods must belong to a class**.
- Methods define the **behavior of objects** (what they can do).

Example: A Simple Method

```
public class CodeFrill{
   // User-defined method
   public void printMessage() {
      System.out.println("Welcome to CodeFrill!");
   }

   public static void main(String[] args) {
      CodeFrill obj = new CodeFrill ();  // Create object
      obj.printMessage();       // Call method
   }
}
```

Output:

```
Welcome to CodeFrill!
```

How It Works:

1. The program starts at main().
2. An object obj of class CodeFrill is created.
3. The method printMessage() is called using obj.

4. Control jumps into the method body → executes System.out.println("Welcome to CodeFrill!").
5. After execution, control returns to the caller (main).

---

3. Method Syntax

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

Components:

- **Modifier** → Access level (public, private, etc.).
- **Return Type** → Data type returned (int, String, or void if nothing).
- **Method Name** → Follows camelCase (e.g., calculateSum).
- **Parameters** → Optional inputs (e.g., int x, int y).
- **Body** → The logic that runs when the method is called.

---

4. Types of Methods in Java (with Explanations)

1. Predefined Methods (Built-in)

- Already provided by Java libraries.
- Example: Math.random(), System.out.println().
- **How it works:**
    - These methods are part of Java's **API (Application Programming Interface)**.
    - When you call Math.random(), the JVM executes pre-written code inside the Math class that generates a random number.

```
System.out.println(Math.random()); // Prints a random number between 0.0 and 1.0
```

Java Methods

---

2. User-Defined Methods

- Written by the programmer to perform specific tasks.
- **How it works:**
    - You define the method inside a class.
    - When called, the JVM creates a **stack frame** for it, executes the code, and then returns control.

```
class Demo {
   void sayHello() {
      System.out.println("Hello from user-defined method!");
   }

   public static void main(String[] args) {
      Demo d = new Demo();
      d.sayHello(); // Method call
   }
}
```

Output:

```
Hello from user-defined method!
```

---

3. Instance Methods

- Belong to an **object**.
- Called using the object name.
- **How it works:**
    - Each object has its own copy of instance variables.

- When you call an instance method, it can access and modify that object's data.

```
class Student {
   String name;

   void displayName() {
      System.out.println("Student name: " + name);
   }

   public static void main(String[] args) {
      Student s1 = new Student();
      s1.name = "Jhobin";
      s1.displayName(); // Accesses s1's data
   }
}
```

Output:

Student name: Jhobin

---

4. Static Methods

- Belong to the **class**, not the object.
- Called using the **class name**.
- **How it works:**
  - Stored in the **method area** of JVM memory.
  - Can only access **static variables** directly.

```
class MathUtils {
```

```
  static int square(int x) {
    return x * x;
  }

  public static void main(String[] args) {
    System.out.println(MathUtils.square(5)); // Call without object
  }
}
```

Output:

25

---

5. Abstract Methods

- Declared without a body.
- Must be implemented in a **subclass**.
- **How it works:**
  - The abstract class provides a **contract**.
  - The subclass provides the **implementation**.

```
abstract class Animal {
  abstract void sound(); // No body
}

class Dog extends Animal {
  void sound() {
    System.out.println("Bark");
```

```
    }

    public static void main(String[] args) {
        Dog d = new Dog();
        d.sound(); // Calls Dog's implementation
    }
}
```

Output:

Bark

---

6. Predefined Object Methods

- Every class in Java inherits from Object.
- Example: hashCode(), toString().
- **How it works:**
    - These methods are defined in the Object class.
    - When you call them, JVM executes the default implementation unless overridden.

```
public class Demo {
    public static void main(String[] args) {
        Demo obj = new Demo();
        System.out.println(obj.hashCode()); // Unique integer
    }
}
```

Output (example):

12345678

Java Methods

---

5. Method Call Stack (How Methods Execute)

- Java uses a **stack** to manage method calls.
- Each method call creates a **stack frame** with:
    - Local variables
    - Parameters
    - Return address

Example:

```
public class CallStackExample {
    public static void A() {
        B();
        System.out.println("In Method A");
    }
    public static void B() {
        C();
        System.out.println("In Method B");
    }
    public static void C() {
        System.out.println("In Method C");
    }
    public static void main(String[] args) {
        A();
    }
}
```

Java Methods

Output:

In Method C

In Method B

In Method A

Explanation:

1. main() calls A() → stack frame for A created.
2. A() calls B() → stack frame for B created.
3. B() calls C() → stack frame for C created.
4. C() finishes → its frame removed.
5. Control returns to B(), then A(), then main().

---

6. Method Signature

- Defined by **method name + parameter list**.
- Example:

int max(int x, int y)

- Name = max
- Parameters = (int, int)
- Return type (int) is **not** part of the signature.

---

7. Naming Methods

- Start with a **verb** in lowercase.
- Use **camelCase** for multi-word names.
- Examples: printMessage(), calculateSum(), getName().

Java Methods

Summary (Java Methods)

- **Methods** = Blocks of code that perform tasks.
- **Types** = Predefined, User-defined, Instance, Static, Abstract.
- **Call Stack** = Manages method execution in LIFO order.
- **Method Signature** = Name + parameters.
- **Naming Rules** = Verb-based, camelCase, descriptive.
- **Calling Methods** = Via object (instance), class (static), or subclass (abstract).