# Imperative vs Declarative Programming in Java

## Introduction

In this tutorial, we will implement a simple example to showcase the difference between \*\***imperative**\*\* and \*\***declarative**\*\* **programming styles**.

### \*\*Use Case\*\*

- **Perform the summation of numbers from \*\*0 to 100\*\* and display the result.**

- We will implement this in both imperative and declarative styles.

## Project Setup

- The example is implemented using \*\*IntelliJ IDEA\*\*.

- Workspace setup and project creation will be covered separately in the course.

- This tutorial focuses on demonstrating the coding differences between the two approaches.

## Imperative Programming Approach

1. Create a class named \*\*ImperativeVsDeclarativeExample\*\*.

2. Make the class executable by adding \*\*public static void main\*\* method.

3. Implement the logic using a \*\*for loop\*\*:

- Declare an integer variable `sum = 0`.

- Iterate through numbers \*\*0 to 100\*\* using a for loop.

- Increment the sum variable by the current loop value.

- Print the sum after loop execution.

### \*\*Code Implementation:\*\*

public class ImperativeVsDeclarativeExample {  
 public static void main(String[] args) {  
 int sum = 0;  
 for (int i = 0; i <= 100; i++) {  
 sum += i;  
 }  
 System.out.println("Sum using Imperative approach: " + sum);  
 }  
}

### \*\*Output:\*\*

- The result will be \*\*5050\*\*.

## Declarative Programming Approach

1. Use \*\*IntStream\*\* introduced in Java 8.

2. Apply \*\*rangeClosed(0,100)\*\* to generate a stream from \*\*0 to 100\*\*.

3. Use \*\*sum()\*\* method to perform summation internally.

4. Print the result.

### \*\*Code Implementation:\*\*

import java.util.stream.IntStream;  
  
public class DeclarativeExample {  
 public static void main(String[] args) {  
 int sum = IntStream.rangeClosed(0, 100).sum();  
 System.out.println("Sum using Declarative approach: " + sum);  
 }  
}

### \*\*Output:\*\*

- The result will be \*\*5050\*\*, same as the imperative approach.

## Comparison: Imperative vs Declarative

### \*\*Imperative Approach (How-style programming)\*\*

- Defines \*how\* the summation should be performed.

- Uses \*\*explicit for-loop\*\* to iterate over numbers.

- Uses \*\*mutable variable (`sum`)\*\*, modifying its state with each iteration.

- Not ideal for \*\*multi-threading\*\*, as it may produce inconsistent results.

- Code is longer and requires reading line by line to understand the logic.

### \*\*Declarative Approach (What-style programming)\*\*

- Defines \*what\* needs to be done rather than \*how\* it is done.

- Uses \*\*functional programming\*\* concepts introduced in Java 8.

- Uses \*\*internal iteration\*\* (`IntStream.rangeClosed()`) instead of external iteration.

- Avoids mutability, making it \*\*thread-safe\*\*.

- More concise and readable.

## Declarative Approach in Multi-threading

One major advantage of the declarative approach is its suitability for \*\*multi-threading\*\*.

- In imperative programming, manual thread creation is needed for parallel execution.

- In declarative programming, \*\*parallel execution\*\* can be enabled easily using `.parallel()` method.

Example:

int parallelSum = IntStream.rangeClosed(0, 100).parallel().sum();  
System.out.println("Sum using Parallel Stream: " + parallelSum);

- \*\*How does it work?\*\* `.parallel()` splits the data and distributes the workload among multiple threads.

- Details about thread creation and execution will be covered in the \*\*Parallel Programming\*\* section of the course.

## Conclusion

- Java 8 enables writing \*\*cleaner, concise, and multi-threading friendly\*\* code using declarative programming.

- The \*\*imperative approach\*\* involves \*\*explicit\*\* iteration and \*\*mutability\*\*.

- The \*\*declarative approach\*\* provides \*\*internal iteration\*\*, \*\*immutability\*\*, and \*\*better readability\*\*.

- The next tutorial will cover another \*\*use case\*\* implementing Java 8 features.

### \*\*End of the Tutorial\*\*

Thank you for watching!