**day11\_107856406\_dsdipt\_sudipto\_19june2025**

**Employee Code:** 107856406

**Login ID:** dsdipt

**Email :** dsdipt@amazon.com

**Name:** Sudipto Das

**Date:** 19 June 2025 (Day 11)

### ***Task 1: What do you understand about java streams?***

A **Stream** in Java is a **pipeline** that lets you process data (like a list or array) in a **declarative**, **functional-style** way — meaning you describe *what* you want to do, not *how* to do it.

**🧠 In My Own Words:**

Think of a stream like a **water pipeline**:

* We have a **source** (like a list of names).
* The water flows through **filters** (like only names starting with "A").
* We can apply **transformations** (like converting all names to uppercase).
* Finally, we do something at the end, like **collect** them or **print** them.

We don’t change the original data — instead, we create a **new result** by flowing through this pipeline.

**Example:**

List<String> names = Arrays.asList("Alice", "Bob", "Ankit", "David");

names.stream()

.filter(name -> name.startsWith("A")) // Keep only names starting with A

.map(String::toUpperCase) // Convert them to uppercase

.forEach(System.out::println); // Print each

**Output:**  
ALICE

ANKIT

**Advantages of Stream:**

**1. Cleaner and More Readable Code**

* Stream operations are often written in a single line.  
  Encourages declarative style — *what* to do, not *how* to do it.

list.stream().filter(x -> x > 10).sorted().forEach(System.out::println);

**2. No Side Effects (Functional Style)**

* Encourages *pure functions* — does not modify the original data.
* Makes code easier to debug and test.

**3. Efficient and Lazy Execution**

* Operations like filter(), map() are *lazy* — only executed when needed (e.g., forEach, collect).
* Can optimize performance by short-circuiting operations (like findFirst()).

**4. Supports Parallel Processing**

* Easily convert streams to **parallel streams** for multi-core processing:

list.parallelStream().filter(...).forEach(...);

* Improves performance with large data sets.

**5. Easy Chaining of Operations**

* You can chain multiple operations like filter(), map(), sorted() together for complex transformations without writing loops.

**6. Reduces Boilerplate Code**

* No need to write verbose for loops, if conditions, or iterator code.

**7. Powerful Terminal Operations**

* Collect results using collect(), toList(), toSet(), or group them using Collectors.groupingBy().

**8. Better for Aggregations and Queries**

* Stream operations like count(), min(), max(), reduce() make data aggregation simple.

### ***Task 2: What is Boilerplate Code?***

Boilerplate code refers to the **repetitive, often structural code** that’s required to set things up, but doesn’t contribute to core logic. It’s especially common in older or verbose languages like Java.

**❌ 1. Lack of Parallelism**

**Explanation:**

Boilerplate-heavy code often:

* **follows sequential and verbose patterns**,
* lacks **declarative or functional constructs**,
* doesn’t naturally support **parallel processing** like Streams or lambda expressions.

**Problem:**

* Traditional for-loops and data processing patterns make **parallelization hard to implement** or manage.
* You have to manually manage threads, locks, and synchronization, which is error-prone.

**Modern Alternative:**

Java 8 Streams:  
list.parallelStream().filter(...).map(...).collect(...);

* Less boilerplate, **automatic parallel execution**, and better performance.

**❌ 2. Lack of Composition**

**Explanation:**

Boilerplate code usually follows an **imperative style**:

* We manually wire everything (e.g., object creation, method calls).
* There's **low reusability**, and components are **tightly coupled**.
* It’s hard to **compose behavior dynamically**.

**Problem:**

* We can’t easily **chain behaviors**, reuse logic, or write clean code that’s **modular and flexible**.
* **Example:** repetitive getter/setter + logic buried in service layers = difficult to compose.

**Modern Alternative:**

Functional-style code encourages **composition**, such as:  
  
Function<String, String> trim = String::trim;

Function<String, String> toUpper = String::toUpperCase;

Function<String, String> composed = trim.andThen(toUpper);

System.out.println(composed.apply(" hello "));

* We can **build behavior** by composing smaller functions instead of writing procedural logic.

### ***Task 3: List of Intermediate and terminal operations***

In Java, **Stream API** operations are divided into two types:

**Intermediate Operations (return another Stream):**

* filter()
* map()
* flatMap()
* sorted()
* distinct()
* limit()
* skip()
* peek()

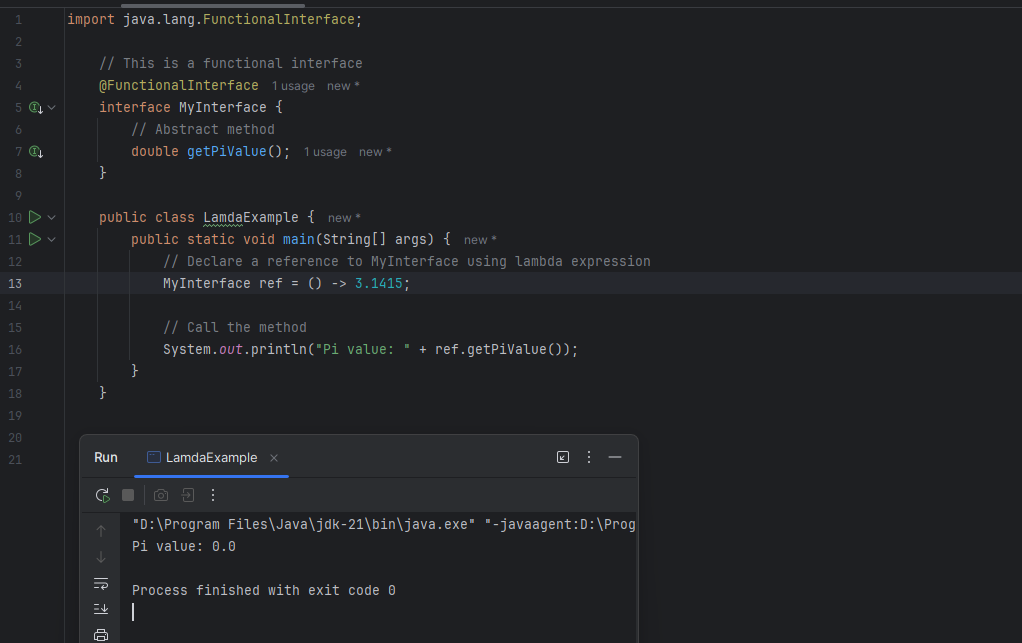
These are **lazy** — they don’t execute until a terminal operation is invoked.

**Terminal Operations (return a result or side-effect):**

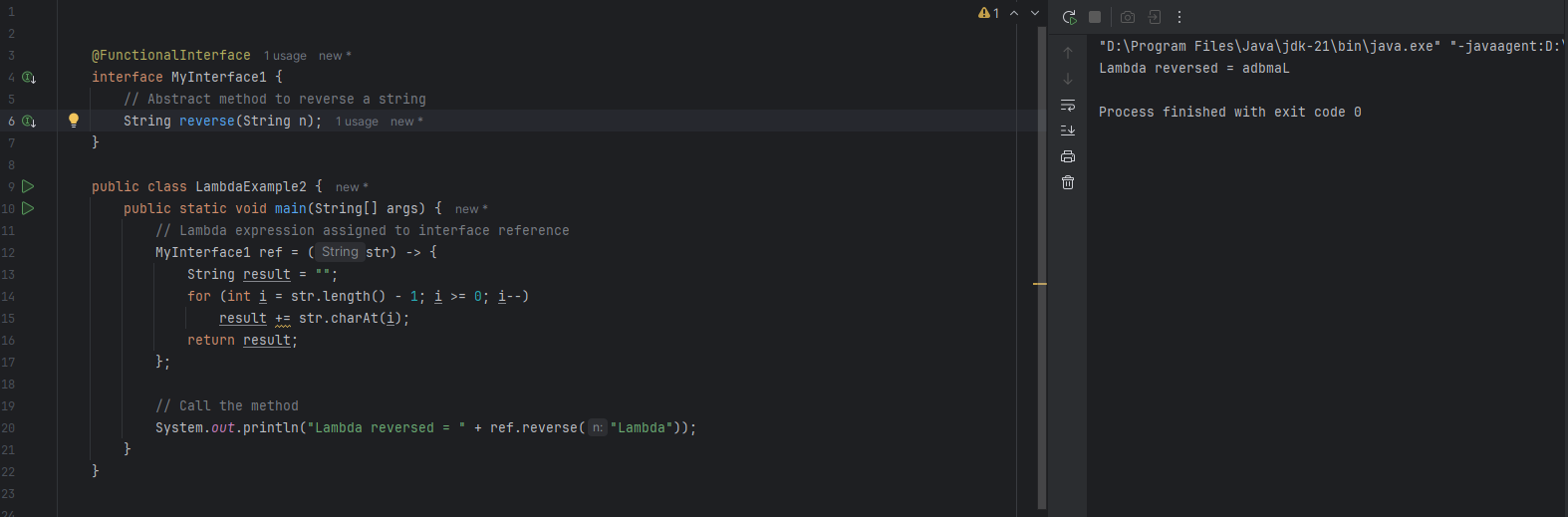
* forEach()
* collect()
* reduce()
* count()
* min(), max()
* anyMatch(), allMatch(), noneMatch()  
  toArray()

These trigger the processing of the stream pipeline.

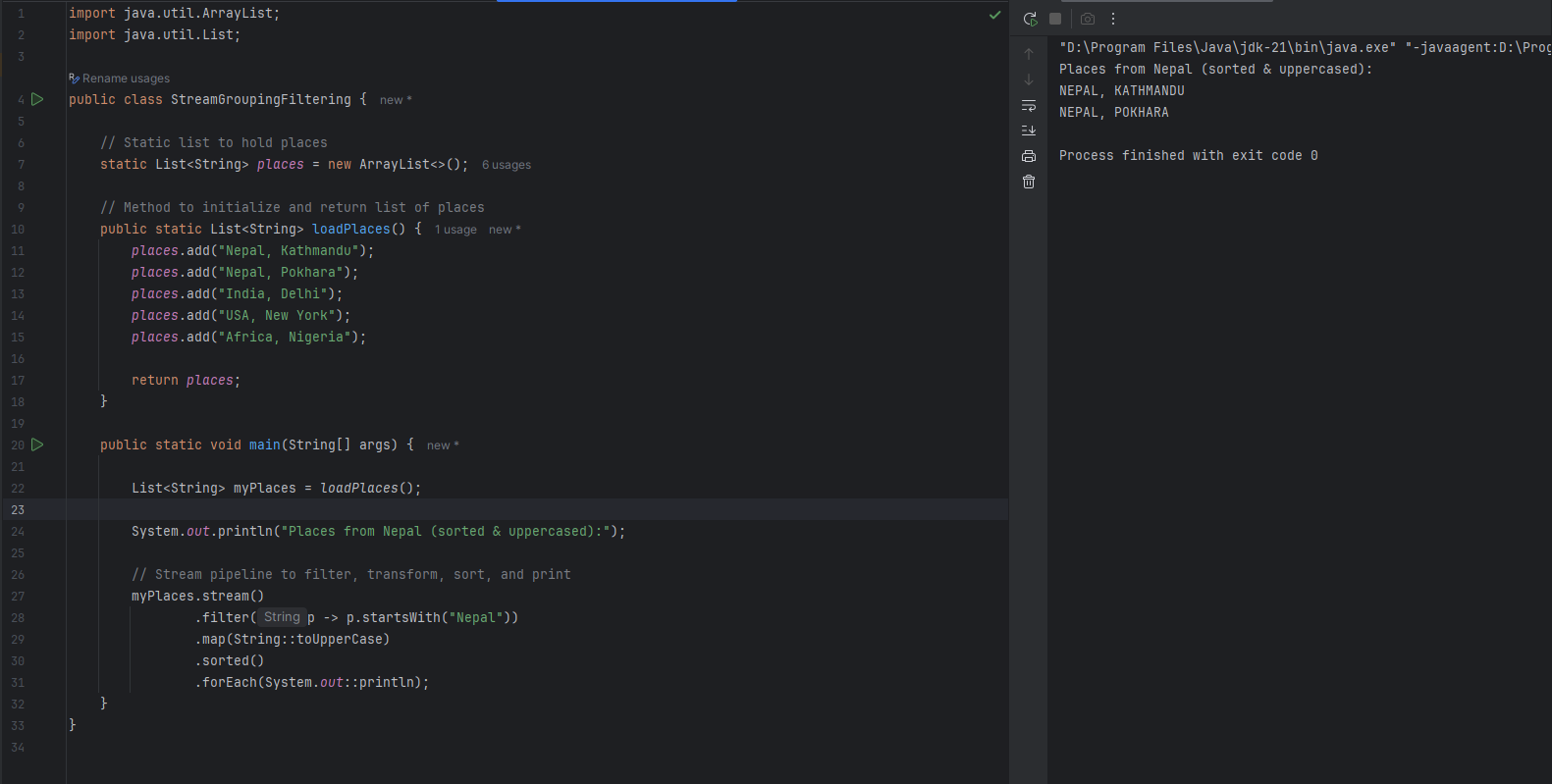
### ***Task 4: Lambda Expression with functional interface***



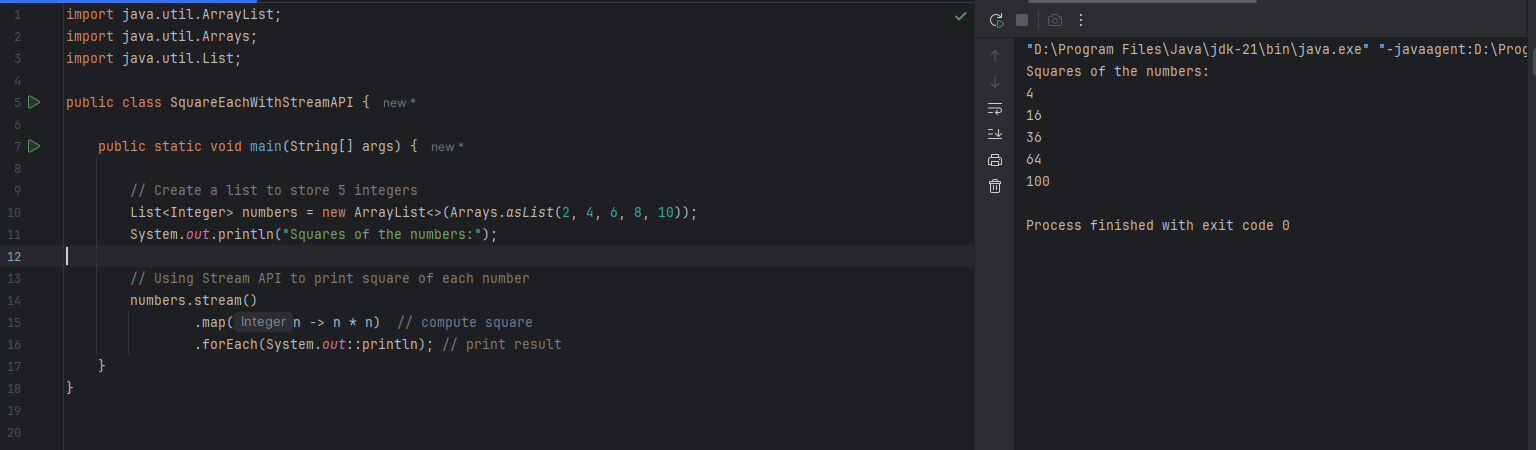
### ***Task 5: Lambda Expression with functional interface with single parameter***



### ***Task 6: Stream Grouping Filtering***



### ***Task 7: Do SquareEach using Stream API***



### ***Task 8: What do you understand by map() in Java Streams?***

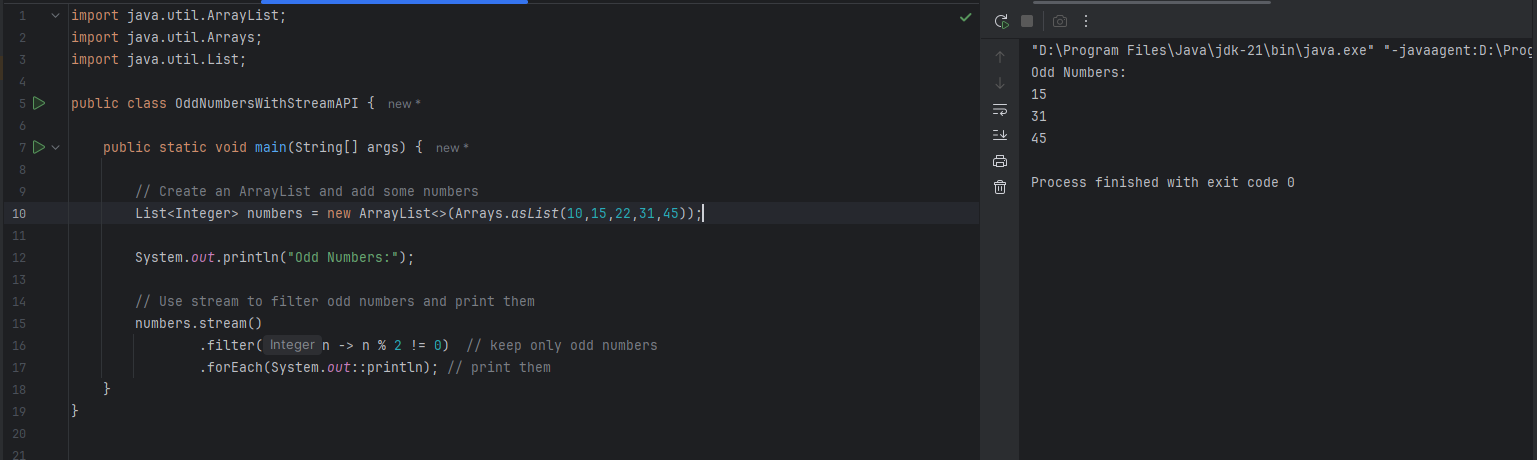
In Java Stream API, the map() method is used to **transform** or **convert** elements of a stream from one form to another.

**Syntax:**

stream.map(function)

* Takes a **Function** as an argument
* Applies the function to **each element** in the stream
* Returns a **new stream** with transformed values
* Does **not modify** the original list (functional style)
* Returns a new stream (immutable)

### ***Task 9: Find Odd Numbers using Stream API***



***Task 10: What do you understand by filter()?***

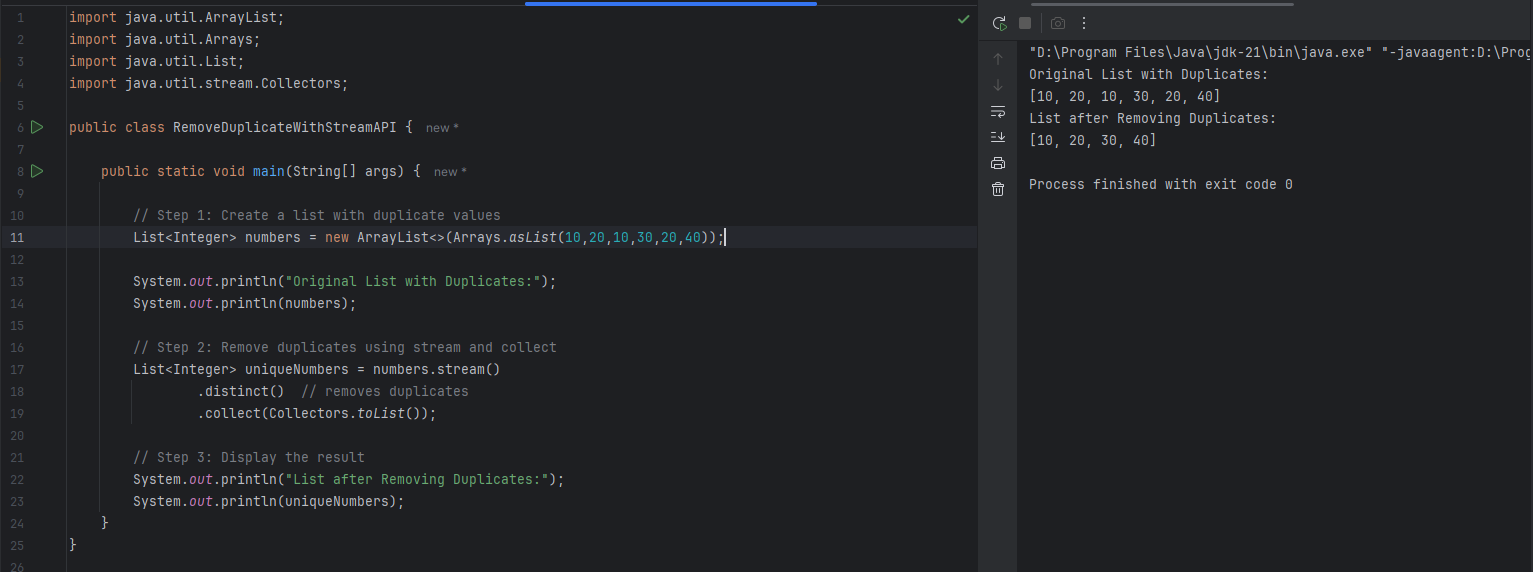
The filter() method in Java Stream API is used to **select elements** from a stream based on a **given condition (predicate)**.

**Syntax:**

stream.filter(predicate)

* Takes a function that returns true or false
* Returns a **new stream** that contains only the elements that satisfy the condition
* To **remove unwanted elements** from a stream and keep only those that meet certain criteria.
* filter() checks every item. If it matches the rule, keep it. If not, skip it.

### ***Task 11: Remove Duplicates using Stream API***



***Task 12: What do you understand by filter()?***

The distinct() method in the Java Stream API is used to **remove duplicate elements** from a stream.

**Syntax:**

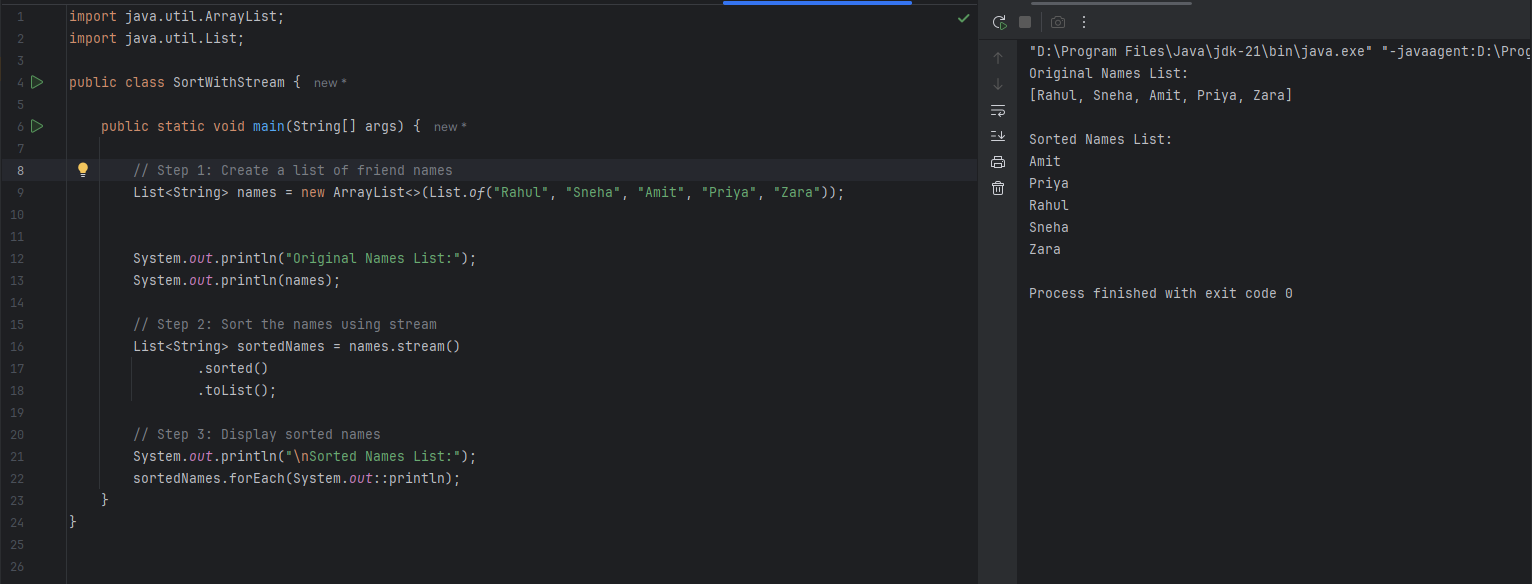
stream.distinct()

* Returns a **new stream** containing only **unique elements**.
* Preserves the **original order** of elements (first occurrence is kept).

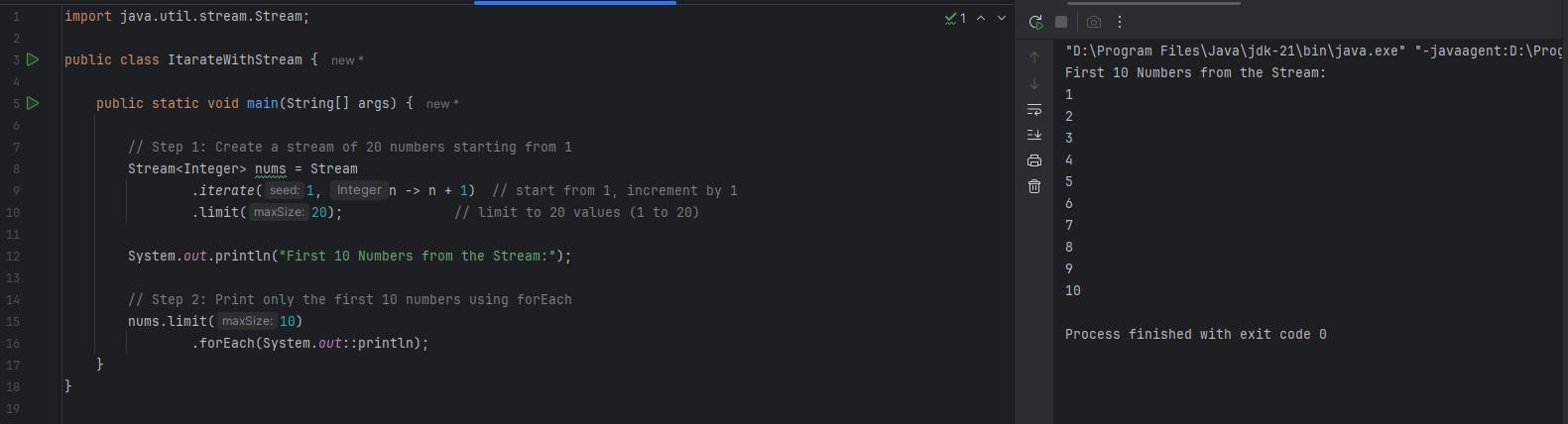
**How It Works:**

* Internally, distinct() uses **equals()** and **hashCode()** methods to determine uniqueness:
* For primitive wrappers (Integer, String, etc.), it works out of the box.
* For **custom objects**, we must **override** equals() and hashCode().

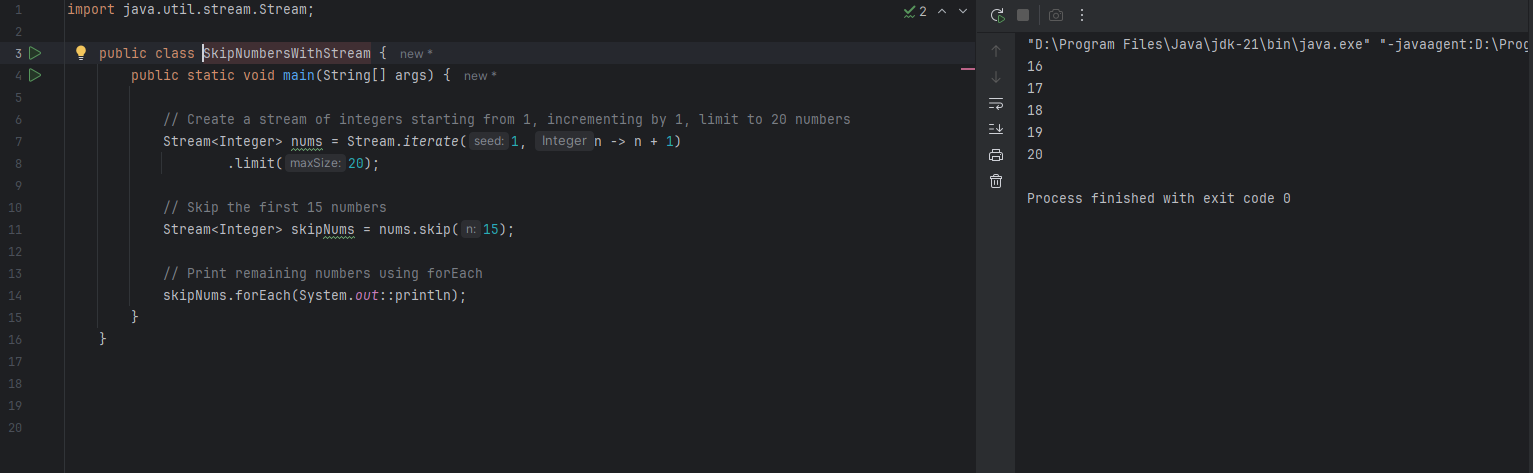
***Task 13: Sort an ArrayList of Friend Names and Display Using Stream API.***



***Task 14: Generate a Stream of Numbers, Limit to 20, and Print First 10 Using forEach()***



***Task 15: Create an ArrayList, Skip 15 Numbers & Print Using Stream***



***Task 16: Explain the limit() and skip() methods in Java's Stream API.***

* **limit() Method**

The limit(long maxSize) method returns a stream consisting of the **first maxSize elements** of the original stream.

**Syntax:**

Stream<T> limitedStream = originalStream.limit(n);

Used to **restrict** the number of elements processed in the stream.

**Example:**

Stream<Integer> stream = Stream.iterate(1, n -> n + 1)

.limit(5);

stream.forEach(System.out::println);

**Output:**

1

2

3

4

5

* **skip() Method**

The skip(long n) method returns a stream with the **first n elements skipped**.

**Syntax:**

Stream<T> skippedStream = originalStream.skip(n);

Used when you want to **ignore** the first few elements of a stream.

✅ Example:

Stream<Integer> stream = Stream.iterate(1, n -> n + 1)

.limit(10)

.skip(5);

stream.forEach(System.out::println);

**Output:**

6

7

8

9

10

***Task 17: Difference Between Mutable and Immutable***

🔄 Mutable Objects

* The **state (data)** of the object **can be changed** after it's created.
* Changes affect the **same object in memory**.
* **Examples:** ArrayList, HashMap, StringBuilder, Date.
* **Not thread-safe** by default – needs synchronization if used in multithreading.
* **Better performance** when frequent modifications are needed.
* Used when you **need to update data frequently**.
* Can cause **unexpected side effects** if shared across methods/threads.

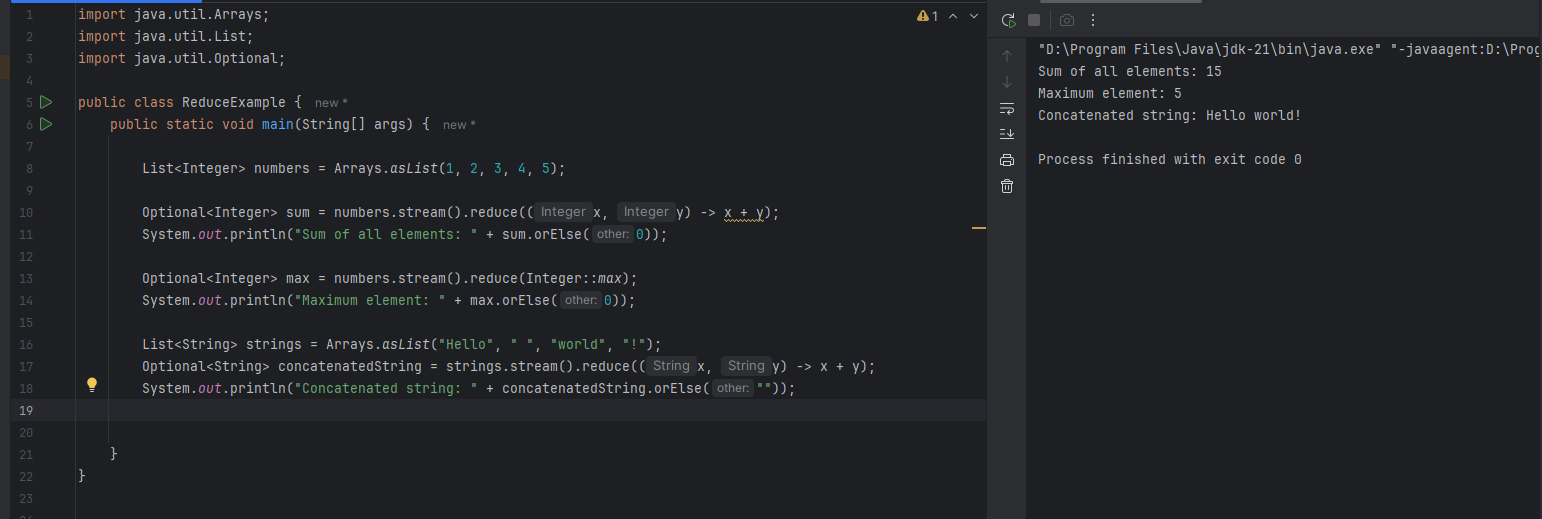
🔒 Immutable Objects

* The **state (data)** of the object **cannot be changed** once created.
* Any modification returns a **new object**, the original stays unchanged.
* **Examples:** String, Integer, LocalDate, BigDecimal.
* **Thread-safe** by design – safe to use in multithreaded environments.
* **Higher memory usage** due to creation of new objects for changes.
* Used when **data should not change**, like keys in a map or constants.
* Easier to **reason about, debug, and maintain**.

**Real-Life Analogy:**

* **Immutable**: A printed document — you can’t change the text; if you want to update, you need to **print a new copy**.
* **Mutable**: A whiteboard — you can **erase and rewrite** as many times as you want.

***Task 18: Using reduce() to Perform Aggregation Operations in Java Streams***



***Task 19: Error Messages in Java – Compile Time vs Run Time***

**Compile-Time Errors**

* These are **detected by the compiler** before the program runs.
* Caused by **syntax or grammatical mistakes** in the code.

**Examples:**

* Missing ;, {}, or ()
* Misspelled keywords (e.g., publc instead of public)  
  Using undeclared variables
* Type mismatch (e.g., assigning a String to an int)

**Characteristics:**

* **Prevents code from compiling**
* Easier to fix since the compiler usually shows the exact line and message

**Run-Time Errors (Exceptions)**

* Errors that occur **while the program is running**.
* The code compiles successfully but **fails during execution**.

**Examples:**

* **NullPointerException** – Accessing an object that is null
* **ArrayIndexOutOfBoundsException** – Accessing an invalid array index
* **ArithmeticException** – Division by zero
* **IOException** – Input/output failure (e.g., file not found)
* **StackOverflowError** – Too many nested method calls (infinite recursion)

**Characteristics:**

* Detected **at runtime**
* Can be **handled using try-catch** blocks

***Task 20: Stack Trace in Java***

A **stack trace** is a report that shows the **sequence of method calls** that the program was executing when an **exception** occurred.

It helps in **debugging** by showing **where** and **why** an error happened.

Identify the Error

* Shows the **type of exception** that was thrown.
* Example:  
   Exception in thread "main" java.lang.NullPointerException

Locate the Code

* Points to the **exact line number** and **class/method** where the error occurred.
* Example:  
   at TaskExample.main(TaskExample.java:12)

Analyze the Code Path

* Shows the **chain of method calls** that led to the error.
* Helps trace back to the **root cause** of the exception.

Find the Solution

* Once the issue is located and understood, you can:
  + Add **null checks**
  + Use **try-catch blocks**
  + Fix **logic errors**
  + Prevent **invalid inputs**

**Example Output:**

Exception in thread "main" java.lang.NullPointerException

at MyClass.myMethod(MyClass.java:10)

at MyClass.main(MyClass.java:5)