Functional Programming and Java Streams: A Comprehensive Guide

Table of Contents

Introduction to Functional Programming

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions, emphasizing pure functions, immutability, and declarative programming. Unlike imperative programming, which focuses on explicit state changes and control flow, functional programming prioritizes "what to solve" over "how to solve it." It draws inspiration from lambda calculus and mathematical principles, promoting predictable and maintainable code.
Java introduced functional programming features in Java 8, including lambda expressions, the `java.util.function` package, and the Stream API. These additions enable developers to write concise, expressive, and parallelizable code.

Key Concepts of Functional Programming

1. **First-Class Functions**
   Functions are treated as first-class citizens, meaning they can be assigned to variables, passed as arguments, or returned from other functions. In Java, lambda expressions and method references enable this behavior.
   Example:
   ```
   // Lambda expression assigned to a variable
   Function<Integer, Integer> square = x -> x * x;
   System.out.println(square.apply(5)); // Output: 25
   ```

2. **Pure Functions**
   Pure functions produce outputs solely based on their inputs and have no side effects (e.g., no modification of external state). This predictability simplifies testing and debugging.
   Example:
   ```
   // Pure function
   int add(int a, int b) {
       return a + b;
   }
   ```

3. **Immutability**
   Immutable data cannot be changed once created, reducing bugs from unintended state modifications. Java supports immutability with `final` variables and immutable collections like `List.of()`.
   Example:
   ```
   final List<String> immutableList = List.of("A", "B", "C");
   // immutableList.add("D"); // UnsupportedOperationException
   ```

4. **Higher-Order Functions**
   These functions accept other functions as parameters or return them. Java's functional interfaces, such as `Function<T, R>`, `Predicate<T>`, and `Consumer<T>`, support higher-order functions.
   Example:
   ```
   Function<Integer, Integer> doubleIt = x -> x * 2;
   Function<Integer, Integer> squareIt = x -> x * x;
   Function<Integer, Integer> doubleThenSquare = doubleIt.andThen(squareIt);
   System.out.println(doubleThenSquare.apply(3)); // (3 * 2)^2 = 36
   ```

5. **Declarative vs. Imperative Style**
6.
   Imperative: Specifies step-by-step instructions for how to achieve a result.
   ```
   List<String> result = new ArrayList<>();
   for (String s : list) {
       if (s.length() > 3) {
           result.add(s);
       }
   }
   ```

   Declarative: Describes the desired outcome without detailing the steps.
   ```
   List<String> result = list.stream()
                             .filter(s -> s.length() > 3)
                             .collect(Collectors.toList());
   ```

Functional Interfaces and Lambda Expressions

A **functional interface** is an interface with a single abstract method (SAM), enabling the use of lambda expressions or method references to implement its behavior. Introduced in Java 8, functional interfaces are the backbone of Java's functional programming capabilities, allowing functions to be treated as first-class citizens. The `java.util.function` package provides common functional interfaces like `Function<T, R>`, `Predicate<T>`, `Consumer<T>`, and `Supplier<T>`.

Understanding Lambda Expressions

Lambda expressions are a key feature of Java 8 that provide a concise way to implement functional interfaces. They enable developers to define anonymous functions (functions without a name) inline, reducing boilerplate code compared to anonymous classes. Lambda expressions are essential for functional programming in Java, as they facilitate passing behavior as data, enabling patterns like function composition and stream processing.

Syntax of Lambda Expressions

A lambda expression consists of three parts:

- **Parameters**: A list of parameters (can be empty) in parentheses. Types can be omitted if inferable.
- **Arrow Operator**: `->` separates parameters from the body.
- **Body**: The expression or block of statements to execute.

General syntax:

```
(parameters) -> expression

// or

(parameters) -> { statements; }
```

Examples of valid lambda expressions:

```
// No parameters
() -> System.out.println("Hello");


// One parameter (parentheses optional for single parameter)
x -> x * 2
// Equivalent with explicit parentheses
(x) -> x * 2


// Multiple parameters
(x, y) -> x + y


// Block body with multiple statements
(x, y) -> {
    System.out.println("Processing: " + x);
    return x + y;
}
```

Type Inference

The Java compiler infers parameter types based on the target functional interface's method signature, making type declarations optional in most cases. For example:

```
Function<Integer, Integer> doubleIt = x -> x * 2;
// Compiler infers x is Integer based on Function<Integer, Integer>
```

Explicit types can be used for clarity or when the compiler cannot infer types:

```
Function<Integer, Integer> doubleIt = (Integer x) -> x * 2;
```

Scoping and Variable Capture

Lambda expressions can access variables from their enclosing scope, but the rules governing this access are designed to ensure predictability and thread safety. The behavior depends on the type of variable being accessed: local variables, method parameters, instance variables, or static variables. Understanding these rules is critical for writing correct and safe lambda expressions, especially in concurrent or parallel stream contexts.

Types of Variables Accessible

Lambda expressions can access the following types of variables from their enclosing scope:

1. **Local Variables and Method Parameters**:

    These are variables declared within the enclosing method or passed as parameters.

    They must be **effectively final**, meaning they are either explicitly declared final or not modified after initialization. This restriction ensures that the lambda's behavior remains consistent and thread-safe, as lambdas may be executed in different threads (e.g., in parallel streams).

    Reason for Effectively Final: Lambda expressions capture variables by reference, not by value. If a local variable could be modified after the lambda is created, it could lead to unpredictable behavior, especially in concurrent scenarios where multiple threads might access or modify the variable simultaneously. Requiring effectively final variables eliminates this risk.

2. **Instance Variables**:

   These are fields of the enclosing class.

   They can be modified unless explicitly declared final, as they are part of the object’s state and are inherently tied to the instance. However, modifying instance variables in a lambda is discouraged in functional programming, as it introduces side effects and violates purity.

3. **Static Variables**:

   These are static fields of the enclosing class.

   Like instance variables, they can be modified unless declared final. Modifying static variables in a lambda is also discouraged due to potential side effects and thread-safety concerns.

Effectively Final Explained

A variable is **effectively final** if it is not modified after its initialization, even if it is not explicitly declared with the final keyword. The Java compiler enforces this rule by checking that any local variable or method parameter used in a lambda expression is not reassigned after its initial value is set. If a variable is reassigned, the compiler will throw an error, preventing the lambda from compiling.

Examples of Variable Capture

Below are examples illustrating how lambda expressions capture different types of variables, including correct usage and common errors.

1. **Capturing an Effectively Final Local Variable**:

```
int factor = 2;

Function<Integer, Integer> multiply = x -> x * factor;

// factor is effectively final (not modified after initialization)

System.out.println(multiply.apply(5)); // Output: 10


// This would cause a compile-time error:

// factor = 3; // Error: Variable used in lambda expression should be final or
effectively final
```

   Explanation: factor is effectively final because it is not modified after being set to 2. The lambda captures factor by reference and uses it to compute x * factor. Attempting to reassign factor violates the effectively final rule, resulting in a compilation error.

2. **Capturing a Method Parameter**:

```
Function<Integer, Predicate<Integer>> createThreshold = threshold -> x -> x >
threshold;

Predicate<Integer> greaterThanFive = createThreshold.apply(5);

System.out.println(greaterThanFive.test(7)); // Output: true

System.out.println(greaterThanFive.test(3)); // Output: false
```

   Explanation: The lambda x -> x > threshold captures the method parameter threshold. Since method parameters are implicitly final in lambda contexts (they cannot be reassigned within the method), threshold is effectively final, and the lambda compiles successfully. This example also shows a higher-order function, where a lambda returns another lambda.

3. **Capturing an Instance Variable**:

```
class Counter {
    private int count = 0;
    Function<Integer, Integer> incrementAndGet = x -> {
        count += x; // Modifying instance variable
        return count;
    };
    void test() {
        System.out.println(incrementAndGet.apply(5)); // Output: 5
        System.out.println(incrementAndGet.apply(3)); // Output: 8
    }
}

public class Main {
    public static void main(String[] args) {
        Counter counter = new Counter();
        counter.test();
    }
}
```

Explanation: The lambda captures the instance variable count and modifies it. This is allowed because instance variables are part of the object's state and are not subject to the effectively final rule. However, modifying count introduces a side effect, which is generally discouraged in functional programming, especially in stream operations where purity is preferred.

4. **Capturing a Static Variable**:

```
class Logger {
    private static int logCount = 0;
    static Consumer<String> log = message -> {
        logCount++; // Modifying static variable
        System.out.println("Log [" + logCount + "]: " + message);
    };
}


public class Main {
    public static void main(String[] args) {
        Logger.log.accept("First message"); // Output: Log [1]: First message
        Logger.log.accept("Second message"); // Output: Log [2]: Second
message
    }
       }
```

Explanation: The lambda captures the static variable logCount and increments it. Like instance variables, static variables can be modified unless declared final. However, modifying static variables can lead to thread-safety issues in concurrent environments, so caution is advised.

5. **Common Pitfall: Non-Effectively Final Variable**:

```
int counter = 0;

Function<Integer, Integer> add = x -> x + counter;

// counter = 1; // Error: Variable used in lambda expression should be
final or effectively final

System.out.println(add.apply(5)); // Would work if counter remains
effectively final: Output: 5
```

Explanation: If counter is modified after the lambda is defined (e.g., by uncommenting counter = 1), the compiler throws an error because counter is no longer effectively final. To fix this, either declare counter as final or ensure it is not reassigned.

Thread Safety and Implications

The effectively final requirement for local variables and method parameters is rooted in thread safety:

- **Capture by Reference**: Lambda expressions capture variables by reference, not by value. This means the lambda accesses the current value of the variable at execution time, not the value it had when the lambda was created. Allowing modifications to captured local variables could lead to race conditions or inconsistent behavior in parallel streams or concurrent executions.
- **Immutable Behavior**: By enforcing effectively final variables, Java ensures that lambda expressions behave predictably, even when executed in different threads. This is particularly important in parallel stream operations, where multiple threads may process elements simultaneously.
- **Instance and Static Variables**: These are not subject to the effectively final rule because they are part of the object or class state, which is inherently shared. However, modifying them in a lambda can introduce thread-safety issues, such as race conditions, unless proper synchronization (e.g., using synchronized blocks or thread-safe data structures) is implemented.

Best Practices for Variable Capture

1. **Prefer Effectively Final Local Variables**: Use local variables or method parameters for capturing values in lambdas, ensuring they are effectively final to maintain thread safety and predictability.
2. **Avoid Modifying Instance or Static Variables**: Modifying these variables in a lambda introduces side effects, which violate functional programming principles. If modification is necessary, consider using thread-safe mechanisms or redesigning the logic to avoid state changes.
3. **Use Final Where Possible**: Explicitly declaring variables as final can improve code clarity and prevent accidental reassignments.
4. **Encapsulate State in Objects**: If state changes are needed, encapsulate the state in an object and pass it to the lambda, or use a stream's stateful operations (e.g., reduce) to manage state declaratively.
5. **Test for Concurrency**: When using lambdas in parallel streams or concurrent contexts, test for thread-safety issues, especially if instance or static variables are involved.

Practical Use Cases

- **Configuration Values**: Capture effectively final local variables to configure lambda behavior, such as thresholds or constants in predicates (e.g., x -> x > threshold).
- **Counters in Non-Concurrent Contexts**: Use instance variables for counters or accumulators in single-threaded applications, but avoid this in parallel streams.
- **Logging**: Capture static variables for logging purposes, but ensure thread-safe logging mechanisms (e.g., java.util.logging.Logger) in concurrent environments.
- **Stream Pipelines**: Use effectively final variables in stream operations to define filtering or mapping logic, ensuring purity and thread safety.

By understanding and adhering to these scoping and variable capture rules, developers can write robust and maintainable lambda expressions that align with functional programming principles and perform reliably in both sequential and parallel contexts.

Benefits of Lambda Expressions

1. **Conciseness**: Lambda expressions eliminate the boilerplate of anonymous classes, making code more readable.
2. **Flexibility**: They allow behavior to be passed as arguments, enabling functional patterns like higher-order functions.

3. **Expressiveness**: Combined with functional interfaces, lambdas make code more declarative, focusing on *what* to do rather than *how*.
4. **Stream Compatibility**: Lambda expressions are integral to the Stream API, enabling concise data processing pipelines.

Examples of Lambda Expressions

1. **Implementing a Custom Functional Interface**

```
@FunctionalInterface
interface MathOperation {
    int operate(int a, int b);
}

MathOperation add = (a, b) -> a + b;
MathOperation multiply = (a, b) -> a * b;

System.out.println(add.operate(5, 3)); // Output: 8
System.out.println(multiply.operate(5, 3)); // Output: 15
```

2. **Using Built-in Functional Interfaces**

```
// Function<T, R>: Takes input T, returns R
Function<String, Integer> length = s -> s.length();
System.out.println(length.apply("Hello")); // Output: 5

// Predicate<T>: Tests input, returns boolean
Predicate<Integer> isEven = n -> n % 2 == 0;
System.out.println(isEven.test(4)); // Output: true

// Consumer<T>: Performs action on input, no return
Consumer<String> print = s -> System.out.println(s);
print.accept("Hello, World"); // Output: Hello, World

// Supplier<T>: Produces output, no input
Supplier<Double> random = () -> Math.random();
System.out.println(random.get()); // Output: Random double between 0.0 and 1.0
```

3. **Lambda in a Stream Pipeline**

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
List<String> filtered = names.stream()
                             .filter(s -> s.length() > 3)
                             .map(s -> s.toUpperCase())
                             .collect(Collectors.toList());
System.out.println(filtered); // Output: [ALICE, CHARLIE]
```

Considerations and Best Practices

- **Readability**: Use lambdas for simple operations; for complex logic, consider named methods or method references to improve clarity.
- **Effective Final Variables**: Ensure captured variables are effectively final to avoid compilation errors and maintain thread safety.
- **Avoid Side Effects**: Lambda expressions should ideally be pure (no side effects) to align with functional programming principles, especially in stream operations.
- **Performance**: Lambdas are generally lightweight, but excessive use in performance-critical code may introduce overhead due to object creation. Profile if necessary.

Lambda expressions are a cornerstone of Java's functional programming model, enabling concise and expressive code. They work seamlessly with functional interfaces and the Stream API, making them indispensable for modern Java development.

Example: Defining and Using a Functional Interface

```
@FunctionalInterface
interface Greeting {
    void sayHello(String name);
}

public class Main {
    public static void main(String[] args) {
        Greeting greet = name -> System.out.println("Hello, " + name);
        greet.sayHello("World"); // Output: Hello, World
    }
}
```

Quiz: Functional Interfaces and Lambda Expressions

1. **What is a functional interface?**
   A functional interface is an interface with exactly one abstract method, used to represent a single function contract. It can be implemented using lambda expressions or method references.

2. **Why are lambda expressions useful in Java?**
   Lambda expressions provide a concise way to implement functional interfaces, reducing boilerplate code and enabling functional programming patterns like passing functions as arguments.

3. **What is the role of the `@FunctionalInterface` annotation?**
   The `@FunctionalInterface` annotation ensures that an interface has exactly one abstract method, providing compile-time checks to enforce compatibility with lambda expressions and method references.

4. **What are some common functional interfaces in `java.util.function`?**
   Common interfaces include:

   `Function<T, R>`: Represents a function that takes an input of type `T` and returns a result of type `R`.

   `Predicate<T>`: Tests an input and returns a boolean.

   `Consumer<T>`: Performs an action on an input without returning a result.

   `Supplier<T>`: Produces a result without taking input.

5. **How does Java's type inference work with lambda expressions?**
   Java's compiler infers parameter types in lambda expressions based on the target functional interface's method signature, allowing developers to omit type declarations for conciseness (e.g., `x -> x * 2` instead of `(Integer x) -> x * 2`).

Method References

Method references, introduced in Java 8, are a concise alternative to lambda expressions. They allow you to refer to existing methods or constructors by name, reducing boilerplate when a lambda expression simply delegates to an existing method. Method references are particularly useful in functional programming, as they enhance readability and reusability when working with functional interfaces, complementing the flexibility of lambda expressions.

Types of Method References

There are four types of method references in Java, each with a specific syntax and use case:

1. **Static Method Reference** (`Class::staticMethod`)
   Refers to a static method of a class.
   Syntax: `ClassName::staticMethodName`

2. **Instance Method Reference of a Particular Object** (`object::instanceMethod`)
   Refers to an instance method of a specific object.
   Syntax: `instance::methodName`
3. **Instance Method Reference of an Arbitrary Object** (`Class::instanceMethod`)
   Refers to an instance method applied to an object of a specified class, where the object is provided as an argument.
   Syntax: `ClassName::instanceMethodName`
4. **Constructor Reference** (`Class::new`)
   Refers to a constructor of a class.
   Syntax: `ClassName::new`

When to Use Method References

Method references are used when a lambda expression would simply call an existing method without additional logic. They improve code readability by replacing verbose lambda expressions with a more compact and descriptive syntax. Use method references when:

- The lambda expression directly invokes a single method.
- The method's signature matches the functional interface's abstract method.
- You want to make the code more concise and self-documenting.

Examples of Method References

Below are examples demonstrating each type of method reference, with comparisons to equivalent lambda expressions.

1. Static Method Reference

Suppose you have a static method to parse a string to an integer:

```
public class Utils {
    public static Integer parse(String s) {
        return Integer.parseInt(s);
    }
}
```

You can use a static method reference with a `Function<String, Integer>`:

```
Function<String, Integer> parser = Utils::parse;
// Equivalent lambda: s -> Utils.parse(s)
System.out.println(parser.apply("123")); // Output: 123
```

In a stream, this is common with methods like `Integer::parseInt`:

```
List<String> strings = Arrays.asList("1", "2", "3");
List<Integer> numbers = strings.stream()
                               .map(Integer::parseInt)
                               .collect(Collectors.toList());
System.out.println(numbers); // Output: [1, 2, 3]
```

2. Instance Method Reference of a Particular Object

If you have an object with an instance method, you can reference it. For example:

```
String prefix = "Hello, ";
Function<String, String> addPrefix = prefix::concat;
// Equivalent lambda: s -> prefix.concat(s)
```

```
System.out.println(addPrefix.apply("World")); // Output: Hello, World
```

This is useful when working with a specific instance, such as a configured object in a pipeline.

3. Instance Method Reference of an Arbitrary Object

This is common in streams when applying an instance method to each element. For example, to convert strings to uppercase:

```
List<String> words = Arrays.asList("apple", "banana", "pear");
List<String> upperWords = words.stream()
                               .map(String::toUpperCase)
                               // Equivalent lambda: s -> s.toUpperCase()
                               .collect(Collectors.toList());
System.out.println(upperWords); // Output: [APPLE, BANANA, PEAR]
```

Here, `String::toUpperCase` applies the `toUpperCase` method to each `String` element in the stream.

4. Constructor Reference

Constructor references are used to create new objects. For example, to create new `String` objects:

```
Supplier<String> stringCreator = String::new;
// Equivalent lambda: () -> new String()
System.out.println(stringCreator.get()); // Output: (empty string)
```

In a stream, constructor references are useful for creating collections:

```
List<String> words = Arrays.asList("a", "b", "c");
List<List<String>> lists = words.stream()
                                .map(Collections::singletonList)
                                // Equivalent lambda: s ->
Collections.singletonList(s)
                                .collect(Collectors.toList());
System.out.println(lists); // Output: [[a], [b], [c]]
```

Benefits and Considerations

- **Readability**: Method references like `String::toUpperCase` are more descriptive than `s -> s.toUpperCase()`, clearly indicating the method being used.
- **Reusability**: They leverage existing methods, reducing code duplication.
- **Limitations**: Method references can only be used when the lambda expression directly calls a method with a matching signature. If additional logic is needed (e.g., `s -> s.toUpperCase() + "!"`), a lambda expression is required.
- **Best Practice**: Use method references for clarity, but fall back to lambdas when the operation involves more complex logic or multiple steps.

Method references bridge the gap between Java's object-oriented and functional paradigms, allowing developers to reuse existing methods in a functional context. They are particularly powerful in stream pipelines, where they complement lambda expressions to make operations like mapping, filtering, and collecting more concise and expressive.

Java Streams and the Stream API

The **Stream API**, introduced in Java 8, provides a functional and fluent way to process collections of data. Streams enable developers to define data processing pipelines using declarative operations, leveraging lambda expressions and method references for concise and expressive code. This improves readability and performance, making streams ideal for modern data processing tasks.

Key Characteristics of Java Streams

1. **Laziness (Delayed Execution)**
   Intermediate operations (e.g., `map`, `filter`) are not executed until a terminal operation (e.g., `collect`, `count`) is invoked. This allows optimizations like short-circuiting.
   Example:

```
List<Integer> result = Stream.of(1, 2, 3, 4, 5)
                             .filter(x -> {
                                 System.out.println("Filtering " + x);
                                 return x > 3;
                             })
                             .collect(Collectors.toList());
// Output only when collect() is called
```

2. **Fusion (Operation Chaining)**
   Chained intermediate operations are fused into a single pass through the data, reducing overhead.
   Example:

```
List<Integer> result = Stream.of(1, 2, 3, 4, 5)
                             .filter(x Caldwell -> x > 2)
                             .map(x -> x * 2)
                             .sorted()
                             .collect(Collectors.toList());
// Single pass: [6, 8, 10]
```

3. **Stateless vs. Stateful Operations**
   - **Stateless**: Operations like `map` and `filter` process each element independently.
   - **Stateful**: Operations like `sorted` or `distinct` require knowledge of the entire stream, maintaining internal state.

4. **Non-Interference and No Side Effects**
   Stream operations should not modify the source data or introduce side effects. Lambda expressions used in streams should be pure to ensure predictable behavior.
   Example:

```
List<Integer> list = Arrays.asList(1, 2, 3);
// Good: Pure function
list.stream().map(x -> x * 2).forEach(System.out::println);
// Bad: Modifying source
list.stream().map(x -> { list.add(x); return x; }); // Avoid!
```

5. **Parallelizability**
   Streams support parallel processing with `.parallelStream()` or `.parallel()`, leveraging Java's Fork/Join framework. Operations must be stateless and associative for correct parallel execution.
   Example:

```
List<Integer> result = Arrays.asList(1, 2, 3, 4, 5)
                             .parallelStream()
                             .map(x -> x * 2)
                             .collect(Collectors.toList());
```

Stream Operations

Stream operations are categorized into three types: **stream sources**, **intermediate operations**, and **terminal operations**. Stream sources create a stream, intermediate operations transform or filter the stream, and terminal operations produce a result or side effect, triggering the pipeline's execution. Below is a detailed explanation of each operation, including its purpose, parameters, return type, and a code example.

Stream Sources

Stream sources generate a stream from various data structures or inputs. They are the starting point of a stream pipeline.

1. **`Collection.stream()`**

Purpose: Creates a sequential stream from a `Collection` (e.g., `List`, `Set`).

Parameters: None (called on a `Collection` instance).

Return Type: `Stream<T>`, where `T` is the type of elements in the collection.

Use Case: Process elements of a list or set declaratively.

Example:

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
Stream<String> stream = names.stream();
stream.forEach(System.out::println);
// Output:
// Alice
// Bob
// Charlie
```

2. **`Stream.of(T... values)`**

   Purpose: Creates a stream from a variable number of elements.

   Parameters: Varargs of type `T` (elements to include in the stream).

   Return Type: `Stream<T>`.

   Use Case: Quickly create a stream from a fixed set of values.

   Example:

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
long count = stream.count();
System.out.println("Count: " + count); // Output: Count: 5
```

3. **`Arrays.stream(T[] array)`**

   Purpose: Creates a stream from an array.

   Parameters: An array of type `T`.

   Return Type: `Stream<T>`.

   Use Case: Process array elements in a stream pipeline.

   Example:

```
    Integer[] numbers = {1, 2, 3, 4, 5};
    Stream<Integer> stream = Arrays.stream(numbers);
    List<Integer> list = stream.collect(Collectors.toList());
    System.out.println(list); // Output: [1, 2, 3, 4, 5]
```

4. **`BufferedReader.lines()`**

   Purpose: Creates a stream of lines from a `BufferedReader`, typically used for reading text files.

   Parameters: None (called on a `BufferedReader` instance).

   Return Type: `Stream<String>`.

   Use Case: Process lines of a file declaratively.

   Example:

```
String text = "Line 1\nLine 2\nLine 3";
BufferedReader reader = new BufferedReader(new StringReader(text));
Stream<String> stream = reader.lines();
stream.forEach(System.out::println);
```

```
// Output:
// Line 1
// Line 2
// Line 3
```

5. **`Pattern.splitAsStream(CharSequence input)`**

   Purpose: Creates a stream of substrings split by a regular expression pattern.

   Parameters: A `CharSequence` to split.

   Return Type: `Stream<String>`.

   Use Case: Split text into tokens for processing.

   Example:

```
Pattern pattern = Pattern.compile(",");
Stream<String> stream = pattern.splitAsStream("apple,banana,pear");
List<String> fruits = stream.collect(Collectors.toList());
System.out.println(fruits); // Output: [apple, banana, pear]
```

Intermediate Operations

Intermediate operations transform or filter a stream, producing another stream. They are lazy, meaning they are not executed until a terminal operation is invoked.

1. **`filter(Predicate<? super T> predicate)`**

   Purpose: Retains elements that satisfy the given predicate.

   Parameters: A `Predicate` (often a lambda) that returns `true` for elements to keep.

   Return Type: `Stream<T>`.

   Use Case: Select elements meeting specific criteria.

   Example:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> evens = numbers.stream()
                             .filter(n -> n % 2 == 0)
                             .collect(Collectors.toList());
System.out.println(evens); // Output: [2, 4]
```

2. **`map(Function<? super T, ? extends R> mapper)`**

   Purpose: Transforms each element using the given function.

   Parameters: A `Function` (lambda or method reference) that maps `T` to `R`.

   Return Type: `Stream<R>`.

   Use Case: Convert elements to a different type or format.

   Example:

```
List<String> words = Arrays.asList("apple", "banana", "pear");
List<String> upper = words.stream()
                          .map(String::toUpperCase)
                          .collect(Collectors.toList());
System.out.println(upper); // Output: [APPLE, BANANA, PEAR]
```

3. **`flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)`**

   Purpose: Transforms each element into a stream and flattens the resulting streams into a single stream.

Parameters: A `Function` that maps `T` to `Stream<R>`.

Return Type: `Stream<R>`.

Use Case: Handle nested collections or structures.

Example:

```
List<List<Integer>> nested = Arrays.asList(Arrays.asList(1, 2),
Arrays.asList(3, 4));
List<Integer> flat = nested.stream()
                           .flatMap(List::stream)
                           .collect(Collectors.toList());
System.out.println(flat); // Output: [1, 2, 3, 4]
```

4. **`distinct()`**

Purpose: Removes duplicate elements based on `equals()`.

Parameters: None.

Return Type: `Stream<T>`.

Use Case: Ensure unique elements in the stream.

Example:

```
List<Integer> numbers = Arrays.asList(1, 2, 2, 3, 3, 4);
List<Integer> unique = numbers.stream()
                              .distinct()
                              .collect(Collectors.toList());
System.out.println(unique); // Output: [1, 2, 3, 4]
```

5. **`sorted()`**

Purpose: Sorts elements in natural order (requires elements to implement `Comparable`).

Parameters: None (or a `Comparator` for custom sorting).

Return Type: `Stream<T>`.

Use Case: Order elements for processing or display.

Example:

```
List<Integer> numbers = Arrays.asList(4, 2, 5, 1, 3);
List<Integer> sorted = numbers.stream()
                              .sorted()
                              .collect(Collectors.toList());
System.out.println(sorted); // Output: [1, 2, 3, 4, 5]
```

6. **`sorted(Comparator<? super T> comparator)`**

Purpose: Sorts elements using a custom `Comparator`.

Parameters: A `Comparator` to define the sort order.

Return Type: `Stream<T>`.

Use Case: Sort elements by specific criteria.

Example:

```
List<String> words = Arrays.asList("apple", "banana", "pear");
List<String> sortedByLength = words.stream()
                                   .sorted(Comparator.comparing(String::l
ength))
```

```
                                   .collect(Collectors.toList());
System.out.println(sortedByLength); // Output: [pear, apple, banana]
```

7. **`peek(Consumer<? super T> action)`**

   Purpose: Performs an action on each element for debugging or side effects without modifying the stream.

   Parameters: A `Consumer` (lambda or method reference) to apply to each element.

   Return Type: `Stream<T>`.

   Use Case: Inspect elements during pipeline execution.

   Example:

```
List<Integer> numbers = Arrays.asList(1, 2, 3);
List<Integer> result = numbers.stream()
                              .peek(n -> System.out.println("Processing:
" + n))
                              .map(n -> n * 2)
                              .collect(Collectors.toList());
// Output:
// Processing: 1
// Processing: 2
// Processing: 3
System.out.println(result); // Output: [2, 4, 6]
```

8. **`limit(long maxSize)`**

   Purpose: Restricts the stream to the first `maxSize` elements.

   Parameters: A `long` specifying the maximum number of elements.

   Return Type: `Stream<T>`.

   Use Case: Process only a subset of elements.

   Example:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> firstThree = numbers.stream()
                                  .limit(3)
                                  .collect(Collectors.toList());
System.out.println(firstThree); // Output: [1, 2, 3]
```

9. **`skip(long n)`**

   Purpose: Skips the first `n` elements of the stream.

   Parameters: A `long` specifying the number of elements to skip.

   Return Type: `Stream<T>`.

   Use Case: Ignore initial elements in processing.

   Example:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> afterTwo = numbers.stream()
                                .skip(2)
                                .collect(Collectors.toList());
System.out.println(afterTwo); // Output: [3, 4, 5]
```

Terminal Operations

Terminal operations produce a result or side effect and trigger the execution of the stream pipeline. After a terminal operation, the stream is considered consumed and cannot be reused.

1. **`forEach(Consumer<? super T> action)`**

   Purpose: Applies an action to each element, typically for side effects like printing.

   Parameters: A `Consumer` (lambda or method reference) to apply to each element.

   Return Type: `void`.

   Use Case: Perform an action on each element, such as logging or updating external state.

   Example:

   ```
   List<String> words = Arrays.asList("apple", "banana", "pear");
   words.stream()
        .forEach(System.out::println);
   // Output:
   // apple
   // banana
   // pear
   ```

2. **`toArray()`**

   Purpose: Converts the stream to an array.

   Parameters: None (or an `IntFunction` for specific array types).

   Return Type: `Object[]` (or `T[]` with a generator).

   Use Case: Obtain an array from stream results.

   Example:

   ```
   List<Integer> numbers = Arrays.asList(1, 2, 3);
   Integer[] array = numbers.stream()
                         .toArray(Integer[]::new);
   System.out.println(Arrays.toString(array)); // Output: [1, 2, 3]
   ```

3. **`reduce(T identity, BinaryOperator<T> accumulator)`**

   Purpose: Combines elements into a single result using an initial value and an accumulator function.

   Parameters: An identity value (starting point) and a `BinaryOperator` to combine elements.

   Return Type: `T`.

   Use Case: Compute sums, products, or other aggregations.

   Example:

   ```
   List<Integer> numbers = Arrays.asList(1, 2, 3, 4);
   int sum = numbers.stream()
                 .reduce(0, Integer::sum);
   System.out.println("Sum: " + sum); // Output: Sum: 10
   ```

4. **`collect(Collector<? super T, A, R> collector)`**

   Purpose: Gathers elements into a collection or other data structure.

   Parameters: A `Collector` defining how to collect elements (e.g., `Collectors.toList()`).

   Return Type: `R` (type of the collected result).

   Use Case: Create lists, sets, maps, or custom collections from stream results.

Example:

```
List<String> words = Arrays.asList("apple", "banana", "pear");
List<String> collected = words.stream()
                              .collect(Collectors.toList());
System.out.println(collected); // Output: [apple, banana, pear]
```

5. **`min(Comparator<? super T> comparator)`**

   Purpose: Finds the smallest element based on a `Comparator`.

   Parameters: A `Comparator` to compare elements.

   Return Type: `Optional<T>`.

   Use Case: Identify the minimum value in a stream.

   Example:

```
List<Integer> numbers = Arrays.asList(3, 1, 4, 2);
Optional<Integer> min = numbers.stream()
                              .min(Integer::compare);
System.out.println("Min: " + min.orElse(0)); // Output: Min: 1
```

6. **`max(Comparator<? super T> comparator)`**

   Purpose: Finds the largest element based on a `Comparator`.

   Parameters: A `Comparator` to compare elements.

   Return Type: `Optional<T>`.

   Use Case: Identify the maximum value in a stream.

   Example:

```
List<Integer> numbers = Arrays.asList(3, 1, 4, 2);
Optional<Integer> max = numbers.stream()
                              .max(Integer::compare);
System.out.println("Max: " + max.orElse(0)); // Output: Max: 4
```

7. **`count()`**

   Purpose: Returns the number of elements in the stream.

   Parameters: None.

   Return Type: `long`.

   Use Case: Count elements after filtering or transformation.

   Example:

```
List<String> words = Arrays.asList("apple", "banana", "pear");
long count = words.stream()
                  .filter(s -> s.length() > 4)
                  .count();
System.out.println("Count: " + count); // Output: Count: 2
```

8. **`anyMatch(Predicate<? super T> predicate)`**

   Purpose: Checks if any element satisfies the given predicate (short-circuits).

   Parameters: A `Predicate` to test elements.

   Return Type: `boolean`.

Use Case: Verify if at least one element meets a condition.

Example:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4);
boolean hasEven = numbers.stream()
                         .anyMatch(n -> n % 2 == 0);
System.out.println("Has even: " + hasEven); // Output: Has even: true
```

9. **`allMatch(Predicate<? super T> predicate)`**

   Purpose: Checks if all elements satisfy the given predicate (short-circuits).

   Parameters: A `Predicate` to test elements.

   Return Type: `boolean`.

   Use Case: Verify if every element meets a condition.

   Example:

```
List<Integer> numbers = Arrays.asList(2, 4, 6);
boolean allEven = numbers.stream()
                         .allMatch(n -> n % 2 == 0);
System.out.println("All even: " + allEven); // Output: All even: true
```

10. **`noneMatch(Predicate<? super T> predicate)`**

   Purpose: Checks if no elements satisfy the given predicate (short-circuits).

   Parameters: A `Predicate` to test elements.

   Return Type: `boolean`.

   Use Case: Verify that no elements meet a condition.

   Example:

```
List<Integer> numbers = Arrays.asList(1, 3, 5);
boolean noEven = numbers.stream()
                        .noneMatch(n -> n % 2 == 0);
System.out.println("No even: " + noEven); // Output: No even: true
```

11. **`findFirst()`**

   Purpose: Returns the first element of the stream, if any (short-circuits).

   Parameters: None.

   Return Type: `Optional<T>`.

   Use Case: Retrieve the first element after filtering or sorting.

   Example:

```
List<String> words = Arrays.asList("apple", "banana", "pear");
Optional<String> first = words.stream()
                              .filter(s -> s.length() > 4)
                              .findFirst();
System.out.println("First: " + first.orElse("None")); // Output: First:
apple
```

12. **`findAny()`**

   Purpose: Returns any element of the stream, useful in parallel streams (short-circuits).

   Parameters: None.

Return Type: `Optional<T>`.

Use Case: Retrieve an arbitrary element, especially in parallel processing.

Example:

```
List<String> words = Arrays.asList("apple", "banana", "pear");
Optional<String> any = words.parallelStream()
                            .filter(s -> s.length() > 4)
                            .findAny();
System.out.println("Any: " + any.orElse("None")); // Output: Any: apple
or banana
```

Code Challenge: Java Streams

1. **Count the number of even numbers in a list using streams.**

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
long evenCount = numbers.stream()
                        .filter(n -> n % 2 == 0)
                        .count();
System.out.println("Even numbers: " + evenCount); // Output: Even numbers: 3
```

2. **Filter out strings with length > 5 from a list using a lambda expression.**

```
List<String> words = Arrays.asList("apple", "banana", "pear", "watermelon");
List<String> longWords = words.stream()
                              .filter(s -> s.length() > 5)
                              .collect(Collectors.toList());
System.out.println(longWords); // Output: [banana, watermelon]
```

Coding Problems

Below are ten coding problems to practice functional programming and Java Streams. Solutions are provided in a separate file.

1. **Problem**: Given a list of integers, find the sum of all numbers divisible by 3 using streams.
2. **Problem**: Convert a list of strings to uppercase and collect them into a new list using streams.
3. **Problem**: Given a list of strings, find the first string longer than 4 characters, or return "None" if none exists.
4. **Problem**: Remove duplicates from a list of integers and sort them in ascending order using streams.
5. **Problem**: Given a list of integers, compute the average of numbers greater than 10.
6. **Problem**: Flatten a list of lists of integers into a single list using `flatMap`.
7. **Problem**: Count the number of words in a sentence (split by spaces) using streams.
8. **Problem**: Given a list of integers, create a map where keys are numbers and values are their squares.
9. **Problem**: Check if all numbers in a list are positive using `allMatch`.
10. **Problem**: Parallelize a stream to compute the sum of squares of numbers in a large list.

Short Answer Questions

Short answers to reinforce understanding of functional programming and Java Streams. Detailed responses are provided in a separate file.

1. What is the primary benefit of using pure functions in functional programming?
2. How does the Stream API improve code readability compared to traditional loops?
3. What is the difference between intermediate and terminal operations in a stream?
4. Why is immutability important in functional programming?

5. How does lazy evaluation optimize stream processing?
6. What is a higher-order function, and how is it implemented in Java?
7. How does the `@FunctionalInterface` annotation assist developers?
8. What are the risks of introducing side effects in stream operations?
9. How can streams be parallelized, and what precautions should be taken?
10. What is the purpose of `flatMap` in the Stream API?