**Functional Programming in Java**

Programming Paradigm is a way or style of programming. The programming paradigm can be classified into two types:

* Imperative programming paradigm
* Declarative programming paradigm

**Imperative Programming Paradigm**

Imperative programming paradigm consists of sequence of statements that changes the program's state. Eg. Procedural programming, Object-Oriented programming, etc. Imperative programming paradigm focuses on **how is it done?**

public class Main {

public static void main(String[] args) {

int sum = 0;

for (int i = 1; i <= 5; i ++) {

sum += i;

}

System.out.println(sum);

}

}

**Declarative Programming Paradigm**

Declarative programming is a paradigm in which we define what needs to be accomplished without defining how it has to be implemented. Eg. **Functional programming**, **Logical programming**, etc. Declarative programming paradigm focuses on ***what we want?***

public class Main {

public static void main(String[] args) {

int[] array = new int[] {1, 2, 3, 4, 5};

int[] evenArray = Arrays.stream(array)

.filter(a -> a % 2 == 0)

.toArray();

System.out.println(Arrays.toString(evenArray));

}

}

1,2,3,4,5

Stream -> 1 2 3 4 5

**Explanation:**

In this example, the input array is converted to *stream*. The filter() method is invoked for each value in the stream and it is evaluated with the expression a -> a % 2 == 0.

filter(1) => false

filter(2) => true

filter(3) => false

filter(4) => true

filter(5) => false

The **filter()** method passes the value to the next operation (**toArray()**) only if the expression inside it evaluates to true. So, in this case, only 2 and 4 are passed to the next **toArray**() operation, and hence the output is [2, 4].

Here, we didn't define the steps to filter the even numbers. Instead we told the compiler what we want via the expression **filter(a -> a % 2 == 0).**

**What is Functional Programming in Java?**

Functional programming is a paradigm where the **basic unit of computation is a function**. Here functions are not the methods we write in programming. Those are called procedures where we write a list of instructions to tell the computer what to do. In functional programming, functions are treated like mathematical functions where we **map inputs to outputs to produce a result**. Eg. f(x) = 3x, *f*(*x*)=3*x+2*.

In functional programming, the software is developed around the evaluation of functions. The functions are isolated and independent, and they **depend only on the arguments passed to them and do not alter the program's state like modifying a global variable**. Functions operate on immutable data and have no side effects on the program.

Functional programming is implemented using the concepts, **First-Class Functions**, **Pure Functions**, **Immutability** and **Referential Transparency** by avoiding shared states and side effects.

**FP Examples**

**Example 1 - Anonymous Class**

public class Main {

public static void main(String[] args) {

List<Integer> list = new ArrayList<>(

Arrays.asList(1, 3, 4, 5, 2)

);

Comparator<Integer> comparator = new Comparator<Integer>() {

@Override

public int compare(Integer o1, Integer o2) {

return o1 - o2;

}

};

list.sort(comparator);

System.out.println(Arrays.toString(list.toArray()));

}

}

**Explanation**

In this example, we created an anonymous inner class (new Comparator<Integer>() {}) that implements the Comparator interface and overrides the compare() method. Anonymous inner class is a class without a name.

**Example 2 - Lambda Expression**

public class Main {

public static void main(String[] args) {

List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));

int sum = list.stream().reduce(0, (a, b) -> a + b);

System.out.println(sum);

}

}

**Explanation**

In this example, we used the **reduce**() method to find the sum of elements in the list. **reduce**() accepts two parameters, an initial value, and a lambda expression. Here we passed the lambda expression **(a, b) -> a + b.**

Lambda expression is an anonymous function that takes in parameters and returns a value. It is called an anonymous function because it doesn't require a name. The syntax for lambda expressions are:

**(parameter1, parameter2, ...) -> expression**

**(parameter1, parameter2, ...) -> { body }**

**Example 3 - Method Reference**

public class Main {

public static void main(String[] args) {

List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));

list.forEach(System.out::print);

}

}

**Explanation**

In this example, instead of using the lambda value -> System.out.print(value) as parameter to the forEach() method, we used **System.out::print** . This is called ***Method Reference***. During each iteration, each value of the list will be passed to the print() method of System.out.

**Characteristics of Functional Programming Language**

**Functions are First Class Citizens**

In Functional programming, functions are considered first-class citizens. A function is first-class if it can be **stored in a variable** and **passed as an argument** to a function like the other primitive types, and it is achieved via **Functional Interface**.

**What is a functional interface?**

A functional interface is an interface that contains **exactly one abstract method**. It is also called **Single Abstract Method (SAM) Interfaces**. It can have any number of default methods. In Java, an interface is annotated with @FunctionalInterface to make it a functional interface. @FunctionalInterface annotation ensures that the interface can't have more than one abstract method. Functional interfaces are meant to be used with lambdas.

@FunctionalInterface

**interface** Concatenator {

String concat(String s1, String s2);

}

**public** **class** FP {

**public** **static** **void** execute(Concatenator concatenator) {

String res = concatenator.concat("A", "B");

System.***out***.println(res);

}

**public** **static** **void** main(String[] args) {

// Function stored in a variable

Concatenator concatenator = (s1, s2) -> s1 + s2;

// Function passed as an argument

*execute*(concatenator);

}

}

**Explanation**

* We defined a functional interface Concatenator with only the concat() method. The concat() method accepts two strings as input and returns the concatenation of those strings.
* We stored the implementation of concat as a lambda expression (s1, s2) -> s1 + s2 in the variable concatenator.
* The lambda expression will be treated as the abstract method concat()'s implementation.
* The concatenator is passed as an argument to the method execute() that calls concatenator.concat().
* Here, (s1, s2) -> s1 + s2 is an anonymous function and it is stored in a variable (concatenator) and passed as an argument (execute() method) just like other primitive types.

**Lazy Evaluation**

Functional programming supports lazy evaluation (aka) call-by-need evaluation.

Lazy evaluation is an evaluation strategy that delays the evaluation of an expression until its value is needed (non-strict evaluation) and which also avoids repeated evaluations.

Lazy evaluation is based on two principles

* Don't evaluate an expression if the value is not needed.
* Don't re-evaluate an expression if the value is needed more than once.

public class Main {

public static void main(String[] args) {

List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));

Stream<Integer> stream = list.stream().filter(value -> {

System.out.println("Checking " + value);

return value % 2 == 0;

});

System.out.println("Started filtering ...");

System.out.println("Result: " + stream.collect(Collectors.toList()));

}

}

**Explanation:**

When we see the above code, we think that *Checking* will be printed first, then *Started filtering ...*, and then *Result: [2, 4]*. But, Java streams use lazy evaluation, and it doesn't evaluate the expression until its result is needed.

The result of the **filter**() method is needed only for the **stream.collect()** method. So, the filter() method is evaluated only when the compiler executes the steam.collect() method. This is the reason behind seeing the output *Checking* after *Started filtering ...*.

**Stateless**

Functional programming doesn't contain any state. It doesn't alter the state of a program during execution.

public static int factorialImperative(int n) {

int ans = 1;

for (int i = 2; i <= n; i++) {

ans \*= i;

}

return ans;

}

public static int factorialDeclarative(int n) {

if (n == 0 || n == 1) return 1;

return n \* factorialDeclarative(n - 1);

}

5 \* fd(4)

5 \* 4 \* fd(3)

5 \* 4 \* 3 \* fd(2)

5 \* 4 \* 3 \* 2 \* fd(1)

5 \* 4 \* 3 \* 2 \* 1 = 120

Consider the above methods factorialImperative() and factorialDeclarative()  for calculating factorial of a number.

The factorialImperative() method uses the variable ans to assign the values during each iteration repeatedly. So, it is changing the state of the program.

This factorialDeclarative() method uses **recursion** to compute the factorial of a number. It doesn't use any variables, only uses method arguments, and doesn't alter the program's state.

**No Flow Controls**

Functional programming doesn't use flow controls like if..else, switch, for, while, do..while. It is completely based on function calls.

**Principles and Concepts of Functional Programming in Java**

**First-Class Functions**

A function is said to be **First-Class** if treated like other variables in a programming language. A First-Class function satisfies the below criteria.

1. A function can be assigned to a variable.
2. A function can be passed as an argument to another function.
3. A function can be returned from another function.

**Higher-Order Functions**

A function is said to be **Higher-Order** if it satisfies any criteria below.

1. A function receives another function as an argument.
2. A function that returns a new function.

**public** **class** FCHO {

// Higher-Order 1: Function accepts a function as an argument

**private** **static** **int** execute(Multiplier<Integer, Integer> multiplier){

**return** multiplier.multiply(5, 10);

}

// Higher-Order 2: Function returns another function

**private** **static** Multiplier<Integer, Integer> getMultiplier() {

// First-Class 1: Function assigned to a variable

Multiplier<Integer, Integer> multiplier = (a, b) -> a \* b;

// First-Class 3: Function returned from a function

**return** multiplier;

}

**public** **static** **void** main(String[] args) {

Multiplier<Integer, Integer> multiplier = *getMultiplier*();

// First-Class 2: Function passed as an argument

**int** ans = *execute*(multiplier);

System.***out***.println(ans);

}

}

@FunctionalInterface

**interface** Multiplier<T, U> {

**int** multiply(T a, U b);

}

**Pure Functions**

A function is said to be pure if it satisfies the below criteria.

1. The return value of a function operates only on the arguments passed.
2. A function returns the same output, given the same input.
3. A function doesn't have any side effects (i.e.) modifying global or local state.

public int sum(int a, int b) {

return a + b;

}

The function sum() is pure because it operates only on the arguments a and b. The sum() function returns the same output, given the same input, and doesn't modify any local or global state.

**Immutability**

Immutability means that an object cannot be modified once created. Java's immutability can be achieved using the final keyword and private access modifier. Immutable class is created in Java using the following rules.

1. The class must be declared as final, which prevents it from subclassing.
2. All fields must be private and final.
3. The class should have one or more public constructors for instantiation.
4. The class should have only getters and not setters.

**Example**

*// 1. Class is final*

final class ImmutableStudent {

*// 2. All fields are private and final*

private final String id;

private final String name;

*// 3. Public constructor for instantiation*

public ImmutableStudent(String id, String name) {

this.id = id;

this.name = name;

}

*// 4. Only getters. No setters.*

public String getId() {

return this.id;

}

public String getName() {

return this.name;

}

}

**Referential Transparency**

An expression is *referentially transparent* if it can be replaced with its corresponding value without changing the program's behavior.

public class Main {

private static int sum(int a, int b) {

System.out.printf("sumReferential: Adding %d and %d%n", a, b);

return a + b;

}

private static int sumReferential(int a, int b) {

return a + b;

}

public static void main(String[] args) {

sumReferential(1, sumReferential(2, 3));

sum(1, sum(2, 3));

}

}

**Output:**

sumReferential: Adding 2 and 3

sumReferential: Adding 1 and 5

**Explanation:**

In this example replacing sumReferential(2, 3) with the value 5 will give the same result 6. But if we replace sum(2,3) with 5, we will miss the printf statement inside sum() method. So, sum() is not referentially transparent.

**Functional Programming Techniques**

**Function Composition**

Function composition is the process of composing bigger functions by combining smaller functions. Function composition can be achieved in Java by Predicate and Function functional interfaces.

**Predicate**

Predicate is a functional interface that **accepts exactly one input and can return a boolean output**. It provides built-in methods like and(), or(), negate(), and isEqual() methods to compose multiple functions. The composed function can be executed using the test() method. The syntax is Predicate<T> where T is the input type. Java also provides BiPredicate that accepts two inputs.

**and()**

Returns true only if **all** the predicates return true, else false. It behaves like logical **AND**.

predicateA.and(predicateB).and(predicateC)...test(value);

**or()**

Returns true if **any** of the predicates return true, else false. It behaves like logical **OR**.

predicateA.or(predicateB).or(predicateC)...test(value);

**negate()**

Returns true if the predicate returns false and vice versa. It behaves like logical **NOT**.

predicateA.negate()...test(value)

**isEqual()**

Returns a predicate that checks if two values are equal. The equality is decided via Object.equals() method.

Predicate.isEqual(value1).test(value2);

Example:

**public** **class** PredicateTesting {

**public** **static** **void** main(String[] args) {

Predicate<Integer> isDivisibleByTwo = (value) -> value % 2 == 0;

Predicate<Integer> isDivisibleByFive = (value) -> value % 5 == 0;

/\*-------------- and() --------------\*/

System.***out***.println("--- and() ---");

Predicate<Integer> isDivisibleByTwoAndFive = isDivisibleByTwo.and(isDivisibleByFive);

System.***out***.println("10 divisible by 2 and 5: " + isDivisibleByTwoAndFive.test(10));

System.***out***.println("4 divisible by 2 and 5: " + isDivisibleByTwoAndFive.test(4));

System.***out***.println();

/\*-------------- or() --------------\*/

System.***out***.println("--- or() ---");

Predicate<Integer> isDivisibleByTwoOrFive = isDivisibleByTwo.or(isDivisibleByFive);

System.***out***.println("4 divisible 2 or 5: " + isDivisibleByTwoOrFive.test(4));

System.***out***.println("3 divisible 2 or 5: " + isDivisibleByTwoOrFive.test(3));

System.***out***.println();

/\*-------------- negate() --------------\*/

System.***out***.println("--- negate() ---");

System.***out***.println("2 divisible by 2: " + isDivisibleByTwo.negate().test(2));

System.***out***.println();

/\*-------------- isEqual() --------------\*/

System.***out***.println("--- isEqual() ---");

System.***out***.println("Tony equal to Stark: " + Predicate.*isEqual*("Tony").test("Stark"));

}

}

**Explanation:**

* isDivisibleByTwo and isDivisibleByFive predicates are composed using and() to form isDivisibleByTwoAndFive predicate. isDivisibleByTwoAndFive returns *true* only if both the predicates return *true*, else *false*.
* isDivisibleByTwo and isDivisibleByFive predicates are composed using or() to form isDivisibleByTwoOrFive predicate. isDivisibleByTwoOrFive returns *true* if either of the predicate returns *true*, else *false*.
* isDivisibleByTwo.negate() returns *true* if the value is not divisible by 2, else *false*. This is because the actual result is negated.
* Predicate.isEqual() returns *true* only if the parameters to isEqual() and test() method are equal. The result is *false* in the example because *Tony* and *Stark* are not equal.

**Function**

Function is also a functional interface that **accepts exactly one input and can return any output**. Function provides built-in methods like compose() and andThen() to compose multiple functions. The composed function can be executed using the apply() method. The syntax is Function<T, U> where T and U are the type of input and output, respectively. Java also provides BiFunction that accepts two inputs.

**compose()**

predicateA.compose(predicateY).test(value);

compose() method first executes the predicateA with value and then executes predicateB with the result of predicateA. The above statement can be expanded as below.

temp = predicateA.apply(value);

result = predicateB.apply(temp);

**andThen()**

predicateA.andThen(predicateY).test(value);

andThen() method firt executes the predicateB with value and then executes predicateA with the result of predicateB. The above statement can be expanded as below.

temp = predicateB.apply(value);

result = predicateA.apply(temp);

Example:

**public** **class** AndThenCompose {

**public** **static** **void** main(String[] args) {

Function<String, String> appendX = (value) -> value + "-X";

Function<String, String> appendY = (value) -> value + "-Y";

// Executes appendX first and then appendY

Function<String, String> appendXAndThenY = appendX.andThen(appendY);

// Executes appendY first and then appendX

Function<String, String> appendYAndThenX = appendX.compose(appendY);

System.***out***.println("andThen: " + appendXAndThenY.apply("A"));

System.***out***.println("compose: " + appendYAndThenX.apply("A"));

}

}

**Explanation:**

* appendXAndThenY.apply("A") returns *A-X-Y* because it is composed with andThen() and appendX is executed before appendY.
* appendYAndThenX.apply("A") returns *A-Y-X* because it is composed with compose() and appendY is executed before appendX.

**Monads**

Monad is a technique which allows us to wrap a value and apply series of transformation to it. Mondas should follow the three laws, **left identity**, **right identity** and **associativity**. Optional is a good example of monad in Java.

public class Main {

public static void main(String[] args) {

Optional<String> concat = Optional.of("A")

.flatMap(a -> Optional.of("B")

.flatMap(b -> Optional.of(a + b)));

System.out.println(concat.get());

}

}

**Explanation:**

In this example, we wrapped the value using the of method and applied a series of transformations using the flatMap method. Each flatMap accepts an input and returns an Optional.

**Currying**

Currying is converting a function with multiple arguments into multiple functions with a single argument.

public class Main {

public static String concatTraditional(String a, String b) {

return a + b;

}

public static void main(String[] args) {

Function<String, Function<String, String>> concatCurried

= a -> b -> a + b;

System.out.println("concatTraditional: "

+ concatTraditional("A", "B"));

System.out.println("concatCurried: "

+ concatCurried.apply("A").apply("B"));

}

}

**Explanation:**

In this example, we created two methods, concatTraditional() and concatCurried(). concatTraditional() is the standard concat method that accepts two strings, whereas concatCurried() has a sequence of methods that takes a single string as an argument.

Currying is incorporated via the lambda expression **a -> b -> a + b**. We can split this expression into two different expressions as below.

1. a is input and b -> a + b is output.
2. b is input and a + b is output.

A picture containing table

Description automatically generated

**Recursion**

Recursion is the technique of making a function call itself. Since the function is called repeatedly, a base condition is required to break the loop.

public static int factorial(int n) {

if (n == 0 || n == 1) return 1;

return n \* factorial(n - 1);

}

class Main {

public static void main(String[] args) {

System.out.println(factorial(5));

}

}

**Explanation:**

The factorial() method is initially called with the value **5** from the main() method. During each recursive call, the value of **n** is decremented by **1**. When the value of **n** becomes **1**, the recursive call breaks and the value is returned to the main() method. The base condition to break the recursive call is **if (n == 0 || n == 1) return 1;**

**Why Functional Programming Matters?**

Our ability to decompose a problem into parts depends directly on our ability to glue solutions together. To support modular programming, a language must provide good glue. Functional programming languages provide two new kinds of glue — higher-order functions and lazy evaluation.

**Why functional programming matters by John Hughes**

* Functional programming operates on functions, and a function is responsible for accomplishing a particular task. This encourages the concept of **modularity**.
* A program can be divided into several modules where each module is responsible for a specific task.
* Each module can be subdivided into several modules by delegating the responsibilities. In functional programming, we define **a function as a module**.

Diagram

Description automatically generated

We can achieve the modularity by the two clues -- **higher-order functions** and **lazy evaluation**. By providing granular responsibilities, we can write code that is **readable**, **maintainable**, **easily debuggable**, and **less error-prone**. These are the factors of a successful programming paradigm.

**Functional Programming Paradigm vs. Object-Oriented Programming Paradigm**

| **Functional Programming Paradigm** | **Object-Oriented Programming Paradigm** |
| --- | --- |
| Functional programming follows **declarative programming paradigm**. | Object-Oriented programming follows **imperative programming paradigm**. |
| Functional programming uses **immutable data**. | Object-Oriented programming uses **mutable data**. |
| Functional programming supports parallel programming. | Object-Oriented programming doesn't support parallel programming. |
| Functional programming operates on **variables** and **functions**. | Object-Oriented programming operates on **objects** and **methods**. |
| Functional programming uses **recursion** for iteration. | Object-Oriented programming uses **loops** for iteration. |
| Functional programming is **stateless**. | Object-Oriented programming is **stateful**. |

**Advantages of the Functional Programming**

* Functional programming uses **pure functions**, which only depends on the input parameters passed to the function and always produces the same output for the given input.
* Functional programming deals with **immutable data** (i.e.) the state doesn't change, so it is easier to debug.
* Functional programming supports **parallel programming** because pure functions don't change the state and operate exclusively on the input parameters.
* Functional programming supports **lazy evaluation**, which avoids unwanted and repeated computations and saves time.
* Functional programming is **thread-safe** because pure functions don't modify the state of a program, and there is no shared state.
* In functional programming, functions can be stored in variables and passed to other functions. This enhances the **readability** of code.

**Conclusion**

* Programming Paradigm is a way or style of programming and can be classified into imperative and declarative.
* Functional programming is a type of declarative programming paradigm.
* Functional programming operates on immutable data and is stateless, meaning it doesn't alter the program's state.
* Functional programming supports lazy evaluation, parallel-programming and it is thread-safe.
* Functional programming is based on First-Class Functions, Pure Functions, Immutability, and Referential Transparency.
* Function composition, currying, monads, and recursion are functional programming techniques.
* Functional programming encourages modularity where complex functions are subdivided into simpler ones.

**Streams in Java**

**Introduction of Streams in Java**

Consider Java stream as a pipeline that consists of a stream source followed by zero or more intermediate operations and a terminal operation. Stream is not a collection or a data structure where we can store data.

* A **stream source** is a Stream instance that contains the initial data.
* **Intermediate operations** are used to perform actions on stream data and return another stream as output.
* **Terminal operations** produce the result of the stream after all the intermediate operations are applied.

Basically we pass input to the stream and apply zero or more intermediate operations to manipulate the data and finally, the result can be collected using a terminal operation.

Stream.of(1, 2, 3, 4, 5) *// Stream source*

.filter(x -> x % 2 == 0) *// Intermediate operation*

.collect(Collectors.toList()) *// Terminal operation*

Diagram

Description automatically generated

**Different Ways to Create Streams in Java**

Java streams can be created in many ways, including creating an empty stream, creating a stream from an existing array, creating a stream from specified values, etc.

**Stream.empty()**

Stream.empty() creates an empty stream without any values. This avoids null pointer exceptions when calling methods with stream parameters. We create empty streams when we want to add objects to the stream in mZ program.

**Syntax**

Stream<T> stream = Stream.empty();

**Example**

public class Main {

public static void main(String[] args) {

Stream<Integer> stream = Stream.empty();

System.out.println("Size: " + stream.count());

}

}

**Explanation**

The Stream.empty() method created an empty stream without any values in it. Hence calling count() method on the stream returned 0.

**Stream.builder()**

Stream.builder() returns a builder (a design pattern that allows us to construct an object step-by-step) that can be used to add objects to the stream. The objects are added to the builder using the add() method. Calling build() method on the builder creates an instance of the Stream. This implementation of Stream creation is based on the famous **Builder Design Pattern**.

**Syntax**

Stream.Builder<T> builder = Stream.builder();

Stream<T> stream = builder.build();

public class Main {

public static void main(String[] args) {

Stream.Builder<String> builder = Stream.builder();

builder.add("Tony Stark")

.add("Steve Rogers")

.add("Thor Odinson");

Stream<String> stream = builder.build();

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

}

}

**Explanation**

The Stream.builder() method returned a Stream.Builder instance and it is used to add strings to the builder instance. The Stream instance is created from the builder by calling the build() method.

**Stream.of()**

Stream.of() method creates a stream with the specified values. This method accepts both single and multiple values. It does the work of declaring and initializing a stream.

**Syntax**

*// Single value*

Stream<T> stream = Stream.of(T value);

*// Multiple values*

Stream<T> stream = Stream.of(T... values);

**Example**

public class Main {

public static void main(String[] args) {

Stream<Integer> stream1 = Stream.of(1);

System.out.println("Size of Stream1: " + stream1.count());

Stream<Integer> stream2 = Stream.of(1, 2, 3);

System.out.println("Size of Stream2: " + stream2.count());

}

}

**Explanation**

We created two streams stream1 and stream2 by passing single (1) and multiple values (1, 2, 3) to the Stream.of() method.

**Arrays.stream()**

Arrays.stream() method creates a stream instance from an existing array. The resulting stream instance will have all the elements of the array.

**Syntax**

Stream<T> stream = Arrays.stream(T[] array);

**Example**

public class Main {

public static void main(String[] args) {

Integer[] array = new Integer[] {1, 2, 3, 4, 5};

Stream<Integer> stream = Arrays.stream(array);

System.out.println("Size: " + stream.count());

}

}

**Explanation**

The stream instance is created from the values of the array passed to the Arrays.stream() method. The created stream instance will have all the values of the array.

**Stream.concat()**

Stream.concat methods combine two existing streams to produce a new stream. The resultant stream will have all the elements of the first stream followed by all the elements of the second stream.

**Syntax**

Stream<T> stream = Stream.concat(Stream<T> stream1, Stream<T> stream2);

**Example**

public class Main {

public static void main(String[] args) {

Stream<Integer> stream1 = Stream.of(1, 2, 3);

Stream<Integer> stream2 = Stream.of(4, 5);

Stream<Integer> stream3 = Stream.concat(stream1, stream2);

System.out.println("Elements of concatenated stream");

stream3.forEach(System.out::println);

}

}

**Explanation**

In this example, stream3 is created by combining stream1 and stream2. stream3 contains all the elements of stream1 followed by all the elements of stream2.

**Stream.generate()**

Stream.generate() returns an **infinite** sequential **unordered** stream where the values are generated by the provided **Supplier**.

Let's understand the important terms.

**Infinite** The values in the stream are infinite (i.e.) unlimited.

**Unordered** Repeated execution of the stream might produce different results.

**Supplier** A functional interface in Java represents an operation that takes no argument and returns a result.

Stream.generate() is useful to create infinite values like random integers, UUIDs (Universally Unique Identifiers), constants, etc. Since the resultant stream is infinite, it can be limited using the limit() method to make it run infinite time.

**Syntax**

Stream<T> stream = Stream.generate(Supplier<T> supplier);

**Example**

public class Main {

public static void main(String[] args) {

List<UUID> uuids = Stream.generate(UUID::randomUUID)

.limit(5)

.collect(Collectors.toList());

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

}

}

**Explanation**

* In this example, we created an infinite stream of UUIDs using the Stream.generate() method.
* The Supplier is UUID::randomUUID that generates random UUIDs.
* Since the stream is infinite, it is limited using the limit() method.

**Stream.iterate()**

Stream.iterate() returns a **infinite** sequential **ordered** stream stream where the values are generated by the provided **UnaryOperator**. Lets understand the important terms.

**Infinite** The values in the stream are infinite (i.e.) unlimited.

**Ordered** Repeated execution of the stream produces the same results.

**UnaryOperator** A functional interface in Java that takes one argument and returns a result that is of the same type as its argument.

Stream.iterate() methods accept two arguments, an initial value called the **seed value** and a unary operator. The first element of the resulting stream will be the seed value. The following elements are created by applying the unary operator on the previous element.

Stream<T> stream = Stream.iterate(T seed, UnaryOperator<T> unaryOperator

public class Main {

public static void main(String[] args) {

List<Integer> values = Stream.iterate(1, n -> n \* 2)

.limit(5)

.collect(Collectors.toList());

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

}

}

**Explanation**

* In this example, we created an infinite ordered stream using the Stream.iterate() method that generates multiples of 2.
* The first element of the stream is 1, which is the seed value passed. The following elements are created by applying the unary operator n -> n \* 2 on the previous element.
* Since the stream is infinite, the results are limited using the limit() method.

**Different Operations on Streams**

Stream provides various operations that can be chained together to produce results. Stream operations can be classified into two types.

* Intermediate Operations
* Terminal Operations

**Intermediate Operations**

Intermediate operations return a stream as the output, and intermediate operations are not executed until a terminal operation is invoked on the stream. This is called lazy evaluation, and it is discussed in detail in the later section (Lazy Evaluation).

**filter()**

The filter() method returns a stream with the stream's elements that match the given predicate. *Predicate is a functional interface in Java that accepts a single input and can return a boolean value.*

public class Main {

public static void main(String[] args) {

final List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));

final List<Integer> ans = list.stream()

.filter(value -> value % 2 == 0)

.collect(Collectors.toList());

System.out.println(Arrays.toString(ans.toArray()));

}

}

**Explanation** This example filters the even values based on the predicate (value -> value % 2 == 0) passed to it.

**map()**

The map() method returns a stream with the resultant elements after applying the given function on the stream elements.

**Example**

public class Main {

public static void main(String[] args) {

final List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));

final List<Integer> ans = list.stream()

.map(value -> value \* 10)

.collect(Collectors.toList());

System.out.println(Arrays.toString(ans.toArray()));

}

}

**Explanation**

In the example the map() method is called with the function value -> value \* 10 on the stream. The function is called for all values of the stream, and hence the result contains all stream values multiplied by 10.

**sorted()**

The sorted() method returns a stream with the elements of the stream sorted according to natural order or the provided Comparator.

**Example**

public class Main {

public static void main(String[] args) {

final List<Integer> list = new ArrayList<>(Arrays.asList(5, 1, 3, 4, 2));

System.out.println("Ascending Order");

list.stream().sorted()

.forEach(System.out::println);

System.out.println("\nDescending Order");

list.stream().sorted(Comparator.reverseOrder())

.forEach(System.out::println);

}

}

**Explanation**

* The sorted() method without any parameters sorts the elements in ascending order.
* The sorted() method with the comparator Comparator.reverseOrder() sorts the element in the descending order.

**distinct()**

This distinct() method returns a stream consisting of distinct elements of the stream (i.e.) it removes duplicate elements.

**Example**

public class Main {

public static void main(String[] args) {

final List<Integer> list = new ArrayList<>(Arrays.asList(1, 1, 2, 2, 3));

final List<Integer> ans = list.stream()

.distinct()

.collect(Collectors.toList());

System.out.println("Distinct List: " + Arrays.toString(ans.toArray()));

}

}

**Explanation**

The distinct() method removes the duplicate values 1 and 2.

**peek()**

The peek() method returns a stream consisting of the elements of the stream after performing the provided action on each element. This is useful when we want to print values after each intermediate operation.

**Example**

public class Main {

public static void main(String[] args) {

final List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));

final List<Integer> ans = list.stream()

.filter(value -> value % 2 == 0)

.peek(value -> System.out.println("Filtered " + value))

.map(value -> value \* 10)

.collect(Collectors.toList());

System.out.println(Arrays.toString(ans.toArray()));

}

}

**Explanation**

We printed the filtered values using the peek() method after the intermediate filter() operation.

**limit()**

The limit() method returns a stream with the stream elements limited to the provided size.

**Example**

public class Main {

public static void main(String[] args) {

final List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));

final List<Integer> ans = list.stream()

.limit(3)

.collect(Collectors.toList());

System.out.println("Limited List: " + Arrays.toString(ans.toArray()));

}

}

**Explanation**

We limited the size of the stream to *3* using the limit(3) method.

**skip()**

This skip() method returns a stream consisting of the stream after discarding the provided first *n* elements.

**Example**

public class Main {

public static void main(String[] args) {

final List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));

final List<Integer> ans = list.stream()

.skip(2)

.collect(Collectors.toList());

System.out.println("Skipped List: " + Arrays.toString(ans.toArray()));

}

}

**Explanation** The first two elements are skipped using the skip(2) method.

**Terminal Operations**

Terminal operations produce the results of the stream after all the intermediate operations are applied, and we can no longer use the stream once the terminal operation is performed.

**forEach()**

The forEach() method iterates and performs the specified action for each stream element. **For parallel stream, it doesn't guarantee to maintain the order of the stream**.

public class Main {

public static void main(String[] args) {

final List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));

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

}

}

**Explanation**

The forEach() method iterates and prints all the stream values.

**forEachOrdered()**

The forEachOrdered() method iterates and performs the specified action for each stream element. This is similar to the forEach() method, and the only difference is that **it maintains the order when the stream is parallel**.

public class Main {

public static void main(String[] args) {

Stream.of("A","B","C")

.parallel()

.forEach(x -> System.out.println("forEach: " + x));

Stream.of("A","B","C")

.parallel()

.forEachOrdered(x -> System.out.println("forEachOrdered: " + x));

}

}

**Explanation**

From the output, we can see that the forEach() method doesn't maintain the order of the stream, whereas the forEachOrdered() method maintains the order of the stream. **collect()** The collect() method performs a mutable reduction operation on the elements of the stream using a Collector.

**Mutable Reduction**

A mutable reduction is an operation in which the reduced value is a mutable result container, like an ArrayList.

**Collector**

A Collector is a class in Java that implements various reduction operations such as accumulating elements into collections, summarizing elements, etc.

public class Main {

public static void main(String[] args) {

List<Integer> evenList = Stream.of(1, 2, 3, 4, 5)

.filter(x -> x % 2 == 0)

.collect(Collectors.toList());

System.out.println(evenList);

}

}

**Explanation**

The filter(x -> x % 2 == 0) method return a stream that contains even numbers. The stream is then converted to a list via collect(Collectors.toList()).

**count()**

The count() method returns the total number of elements in the stream.

public class Main {

public static void main(String[] args) {

final List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));

final long count = list.stream().count();

System.out.println("Count: " + count);

}

}

**Explanation**

The count() method returns the total number of elements in the stream, *5*.

**reduce()**

The reduce() method performs a reduction on the elements of the stream and returns the value.

**Example**

public class Main {

public static void main(String[] args) {

final List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));

final int ans = list.stream().reduce(0, (value, sum) -> sum += value);

System.out.println("Sum: " + ans);

}

}

**Explanation**

* The reduce() method is called with two arguments, an initial value (*0*) and the accumulator method (value, sum) -> sum += value).
* Each stream element will be added to the previous result to produce the sum.

**toArray()**

The toArray() method returns an array that contains the elements of the stream.

**Example**

public class Main {

public static void main(String[] args) {

Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);

System.out.println(Arrays.toString(stream.toArray()));

}

}

**Explanation**

The toArray() method returns an array with the elements of the stream.

**min() and max()**

The min() and max() methods return an Optional that contains the minimum and maximum elements of the stream, respectively, according to the provided comparator.

**Example**

public class Main {

public static void main(String[] args) {

Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);

int minimum = stream.min((a, b) -> Integer.compare(a, b)).get();

System.out.println("Minimum: " + minimum);

stream = Stream.of(1, 2, 3, 4, 5);

int maximum = stream.max((a, b) -> Integer.compare(a, b)).get();

System.out.println("Maximum: " + maximum);

}

}

**Explanation**

The min() and max() methods return optionals that contain the minimum and maximum elements of the stream respectively. The comparator used to compare the elements is (a, b) -> Integer.compare(a, b). This comparator returns **0** if a == b, a **negative value** if a < b and, a **positive value** if a > b.

**findFirst()**

The findFirst() method returns an Optional that contains the first element of the stream or an empty Optional if the stream is empty.

**Example**

public class Main {

public static void main(String[] args) {

final List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));

Optional<Integer> ans = list.stream().findFirst();

ans.ifPresent(value -> System.out.println("First Value: " + value));

}

}

**Explanation** The findFirst() method returns an Optional with the first element of the stream, which is *1*.

**findAny()**

The findAny() method returns an Optional containing some element of the stream or an empty Optional if the stream is empty.

**Example**

public class Main {

public static void main(String[] args) {

final List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));

Optional<Integer> ans = list.stream().findAny();

ans.ifPresent(value -> System.out.println("Any Value: " + value));

}

}

**Explanation**

The findAny() method can return any element in this stream. In this example, it returns 1. **noneMatch()** When no stream elements match the specified predicate, the noneMatch() method returns true, otherwise false. If the stream is empty, it returns true.

**Example**

public class Main {

public static void main(String[] args) {

final List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));

boolean containsTwo = list.stream().noneMatch(value -> value == 2);

boolean containsZero = list.stream().noneMatch(value -> value == 0);

System.out.println("Is 2 present: " + containsTwo);

System.out.println("Is 0 present: " + containsZero);

}

}

**Explanation**

* The first noneMatch(value -> value == 2) returns false because there is a 2 in the stream.
* The second noneMatch(value -> value == 0) returns true because there is no 0's in the stream.

**allMatch()**

When all the stream elements meet the specified predicate, the allMatch() method returns true, otherwise false. If the stream is empty, it returns true.

**Example**

public class Main {

public static void main(String[] args) {

final List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));

boolean isAllEven = list.stream().allMatch(value -> value == 2);

System.out.println("Is all even: " + isAllEven);

}

}

**Explanation**

The allMatch(value -> value == 2) returns false because there are other elements in the stream apart from 2.

**anyMatch()**

When any stream element matches the specified predicate, the anyMatch() method returns true, otherwise false. If the stream is empty, it returns false.

**Example**

public class Main {

public static void main(String[] args) {

final List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));

boolean isTwoPresent = list.stream().anyMatch(value -> value == 2);

System.out.println("Is 2 present: " + isTwoPresent);

}

}

**Explanation**

The anyMatch(value -> value == 2) returns true because there is a 2 in the stream.

**Stream Short-circuit Operations**

Stream short-circuit operations can be better understood with Java's logical && and || operators.

* expression1 && expression2 doesn't evaluate expression2 if expression1 is false because false && anything is always false.
* expression1 || expression2 doesn't evaluate expression2 if expression1 is true because true || anything is always true.

Stream short-circuit operations are those that can terminate without processing all the elements in a stream. Short-circuit operations can be classified into two types.

1. Intermediate short-circuit operations
2. Terminal short-circuit operations

**Intermediate Short-Circuit Operations**

Intermediate short-circuit operations produce a finite stream from an infinite stream. The intermediate short-circuit operation in the Java stream is limit().

**Example**

public class Main {

public static void main(String[] args) {

Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);

stream.limit(3).forEach(System.out::println);

}

}

**Explanation**

The method limit(3) reduced the stream size from 5 to 3.

**Terminal Short-Circuit Operations**

Terminal short-circuit operations are those that can produce the result before processing all the elements of the stream. The terminal short-circuit operations in stream are findFirst(), findAny(), allMatch(), anyMatch(), and noneMatch().

**Example**

public class Main {

public static void main(String[] args) {

Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);

boolean ans = stream.peek(x -> System.out.println("Checking " + x))

.anyMatch(x -> x == 3);

System.out.println("Is 3 present: " + ans);

}

}

**Explanation**

The terminal operation anyMatch(x -> x == 3) halts once 3 is found and doesn't process the remaining stream elements.

**Parallel Stream**

By default, all stream operations are sequential in Java unless explicitly specified as parallel. Parallel streams are created in Java in two ways.

1. Calling the parallel() method on a sequential stream.
2. Calling parallelStream() method on a collection.

Parallel streams are useful when we have many independent tasks that can be processed simultaneously to minimize the running time of the program.

**parallel()**

parallel() method is called on the existing sequential stream to make it parallel.

**Example**

public class Main {

public static void main(String[] args) {

Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);

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

}

}

**Explanation**

The stream instance is converted to parallel stream by calling stream.parallel(). Since the forEach() method is called on a parallel stream, the output order will not be same as the input order because the elements are processed parallel.

**parallelStream()**

parallelStream() is called on Java collections like **List**, **Set**, etc to make it a parallel stream.

**Example**

public class Main {

public static void main(String[] args) {

List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));

list.parallelStream().forEach(System.out::println);

}

}

**Explanation**

The list object is converted to a parallel stream by calling the parallelStream() method.

**Lazy Evaluation**

Lazy evaluation (aka) call-by-need evaluation is an evaluation strategy that delays evaluating an expression until its value is needed. All intermediate operations are performed on the stream only when a terminal operation is invoked on it. Lazy evaluation is one of the critical characteristics of Java streams that allows significant optimizations.

**Example**

Consider the below example where we created a stream that filters odd numbers.

public class Main {

public static void main(String[] args) {

Stream<Integer> stream = Stream.of(1, 2, 3)

.filter(x -> x % 2 == 1)

.peek(x -> System.out.println("Filtered " + x));

System.out.println("Result");

System.out.println(stream.collect(Collectors.toList()));

}

}

**Explanation**

* Each element of the above stream is passed to the filter() method where odd numbers are filtered and then the peek() method where the filtered value is printed.
* Finally, we print the text *Result* and the filtered values.
* If we look at the statements, the ideal order of printing should be

Filtered 1

Filtered 2

Result

[1, 3]

* But we got a different order because the intermediate operations filter() and peek() are not executed until the terminal operation collect() is invoked on the stream.

**Conclusion**

* Stream is a sequence of objects that provides various sequential and parallel aggregate operations.
* Stream provides many intermediate and terminal operations that can be applied to the stream instance.
* Intermediate operations return a stream as the output and are not executed until a terminal operation is invoked on the stream. filter, map, sorted are some intermediate operations.
* Terminal operations produce the results of the stream. forEach, collect, min, max are some of the terminal operations.
* Some stream operations are short-circuiting (i.e.); those operations terminate without processing all the elements in a stream.
* Stream follows lazy evaluation, and all intermediate operations are performed on the stream only when a terminal operation is invoked.