

JAVA 8

Features of Java 8

- Lambda expressions,
- Method references,
- Functional interfaces,
- Stream API,
- Default methods,
- Base64 Encode Decode,
- Static methods in interface,
- Optional class,
- Collectors class,
- ForEach() method,
- Nashorn JavaScript Engine,
- Parallel Array Sorting,
- Type and Repating Annotations,
- IO Enhancements,
- Concurrency Enhancements,
- JDBC Enhancements etc

STREAMS

Stream is A sequence of elements supporting sequential and parallel aggregate operations.

To perform a computation, stream [operations](#) are composed into a *stream pipeline*. A stream pipeline consists of a source (which might be an array, a collection, a generator function, an I/O channel, etc), zero or more *intermediate operations* (which transform a stream into another stream, such as `filter(Predicate)`), and a *terminal operation* (which produces a result or side-effect, such as `count()` or `forEach(Consumer)`).

Streams are lazy; computation on the source data is only performed when the terminal operation is initiated, and source elements are consumed only as needed.

Collections and streams, while bearing some superficial similarities, have different goals. Collections are primarily concerned with the efficient management of, and access to, their elements. By contrast, streams do not provide a means to directly access or manipulate their elements, and are instead concerned with declaratively describing their source and the computational operations

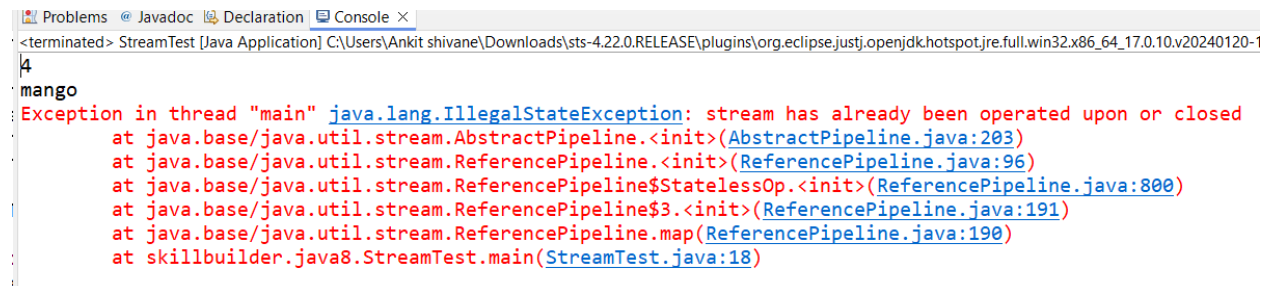
A stream implementation may throw `IllegalStateException` if it detects that the stream is being reused.

For Example:

```
public static void main(String[] args) {
    List<String> fruits = Arrays.asList("mango", "orange",
    "apple", "banana", "kiwi");

    Stream<String> str1 = fruits.stream().filter(i ->
i.contains("a")).filter(i -> i.length() > 3);
    String orElse = str1.map(i ->
i.toLowerCase()).findFirst().orElse("did not found anything");
    System.out.println(orElse);
    String any = str1.map(i ->
i.toUpperCase()).findAny().orElse("did ssnot found anything");
    System.out.println(any);
}
```

OUTPUT:



```
<terminated> StreamTest [Java Application] C:\Users\Ankit shivane\Downloads\sts-4.22.0.RELEASE\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.10.v20240120-1
4
mango
Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed
    at java.base/java.util.stream.AbstractPipeline.<init>(AbstractPipeline.java:203)
    at java.base/java.util.stream.ReferencePipeline.<init>(ReferencePipeline.java:96)
    at java.base/java.util.stream.ReferencePipeline$StatelessOp.<init>(ReferencePipeline.java:800)
    at java.base/java.util.stream.ReferencePipeline$3.<init>(ReferencePipeline.java:191)
    at java.base/java.util.stream.ReferencePipeline.map(ReferencePipeline.java:190)
    at skillbuilder.java8.StreamTest.main(StreamTest.java:18)
```

Stream pipelines may execute either sequentially or in parallel. `Collection.stream()` creates a sequential stream, and `Collection.parallelStream()` creates a parallel one.)

Here's a list of methods in Java 8 Streams API:

1. Creating Streams

- of
- generate

- iterate
- empty
- concat
- builder

2. Intermediate Operations

- filter
- map
- mapToInt
- mapToLong
- mapToDouble
- flatMap
- flatMapToInt
- flatMapToLong
- flatMapToDouble
- distinct
- sorted
- peek
- limit
- skip

3. Terminal Operations

- forEach
- forEachOrdered
- toArray
- reduce
- collect
- min
- max
- count
- anyMatch

- allMatch
- noneMatch
- findFirst
- findAny

4. Short-circuiting Operations

- anyMatch
- allMatch
- noneMatch
- findFirst
- findAny
- limit

5. Specialized Streams

- IntStream
- LongStream
- DoubleStream

6. Stream Collectors

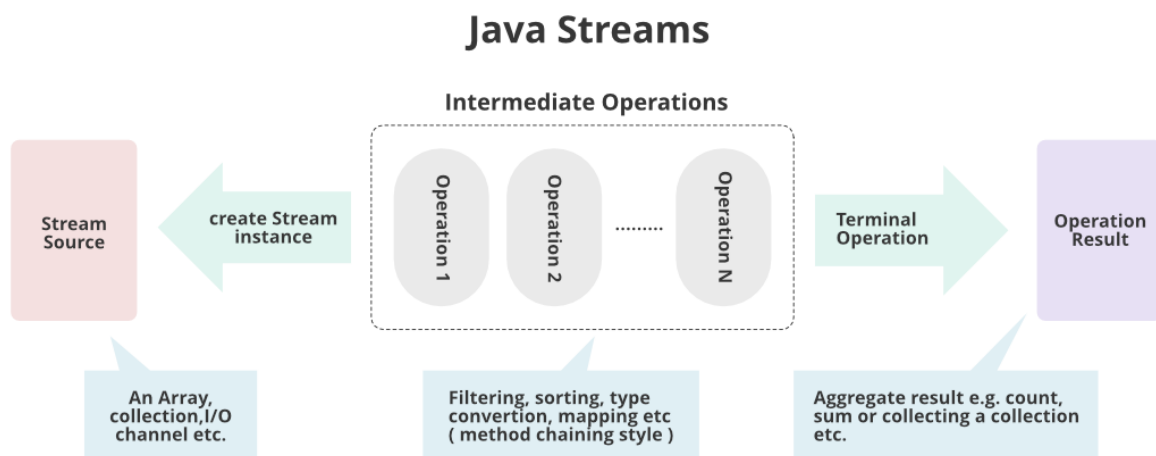
- Collectors.toList
- Collectors.toSet
- Collectors.toMap
- Collectors.toCollection
- Collectors.joining
- Collectors.counting
- Collectors.summarizingInt
- Collectors.summarizingDouble
- Collectors.summarizingLong
- Collectors.averagingInt
- Collectors.averagingDouble
- Collectors.averagingLong

- Collectors.reducing
- Collectors.collectingAndThen
- Collectors.partitioningBy
- Collectors.groupingBy
- Collectors.mapping
- Collectors.minBy
- Collectors.maxBy

7 Additional Methods

- parallel
- sequential
- unordered
- onClose
- close

This list covers the primary methods provided by the Java 8 Streams API, including creation, intermediate, terminal, short-circuiting operations, specialized streams, and collectors.



1. Creating Streams:

- of
- generate

- iterate
- empty
- concat
- builder

A. Empty Stream creation:

We should use the `empty()` method in case of the creation of an empty stream:

```
Stream<String> streamEmpty = Stream.empty();
```

B. Stream of collection:

We can also create a stream of any type of *Collection* (*Collection*, *List*, *Set*):

```
Collection<String> collection = Arrays.asList("a", "b", "c");
Stream<String> streamOfCollection = collection.stream();
```

C. Stream of Array:

An array can also be the source of a stream:

```
String[] arr = new String[]{"a", "b", "c"};
Stream<String> streamOfArrayFull = Arrays.stream(arr);
Stream<String> streamOfArrayPart = Arrays.stream(arr, 1, 3);
```

D. Of Method:

Returns a sequential ordered stream whose elements are the specified values like `of(T... values)`. And also it can Returns a sequential Stream containing a single element like `of(T t)`

```
Stream<String> stream = Stream.of("a", "b", "c");
```

E. Generate method:

Returns an infinite sequential unordered stream where each element is generated by the provided `Supplier`. This is suitable for generating constant streams, streams of random elements, etc.

```
Stream<Double> randomNumbers = Stream.generate(Math::random);
randomNumbers.limit(50).forEach(i -> System.out.println(i));
```

here it generates 50 random elements if we do not limit the elements till 50 then it will generate a infinite stream.

F. Iterate method:

The `Stream.iterate` method creates an infinite stream where each element is generated by applying a function to the previous element.

Syntax:

```
static <T> Stream<T>
```

```
iterate(T seed, UnaryOperator<T> f)
```

@param <T> the type of stream elements

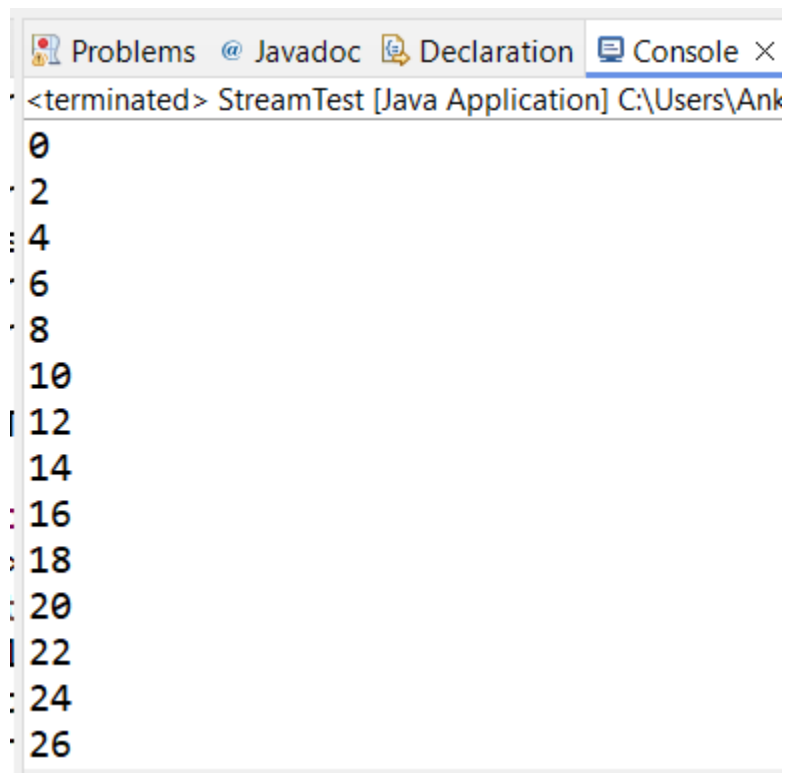
@param seed the initial element

@param f a function to be applied to the previous element to produce
a new element

@return a new sequential {@code Stream}

Example:

```
Stream<Integer> evenNumbers = Stream.iterate(0, n -> n + 2);
```



The screenshot shows an IDE window with tabs for Problems, Javadoc, Declaration, and Console. The Console tab is active, displaying the output of a Java application named 'StreamTest'. The output consists of a sequence of even numbers from 0 to 26, with each number on a new line. The numbers are: 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, and 26. The console window title is '<terminated> StreamTest [Java Application] C:\Users\Ank'.

G. Concat method:

The **Stream.concat** method concatenates two streams into a single stream.

```
Stream<String> stream1 = Stream.of("a", "b", "c");
```

```
Stream<String> stream2 = Stream.of("d", "e", "f");
```

```
Stream<String> combinedStream = Stream.concat(stream1, stream2);
```

```
combinedStream.forEach(i -> System.out.print(" " + i));
```

OUTPUT: a b c d e f

H. Builder method

The `Stream.builder` method allows you to create a stream by adding elements one by one.

```
Stream.Builder<String> builder = Stream.builder();
builder.add("a");
builder.add("b");
builder.add("c");
Stream<String> stream = builder.build();
stream.forEach(i->System.out.print(" "+i));
OUTPUT: a b c
```

. Intermediate Operations

- Filter
 - `filter(Predicate<? super T> predicate)`
 - Returns a stream consisting of the elements of this stream that match the given predicate.
- Map
 - `map(Function<? super T,? extends R> mapper)`
 - Returns a stream consisting of the results of applying the given function to the elements of this stream.
- `mapToInt`
 - Transforms each element of the stream into an `int`.
 - `List<String> words = Arrays.asList("a", "ab", "abc");`
`IntStream lengths = words.stream().mapToInt(String::length);`
- `mapToLong`
 - Transforms each element of the stream into a `long`.
 - `List<String> words = Arrays.asList("a", "ab", "abc");`
`LongStream lengths = words.stream().mapToLong(String::length);`
- `mapToDouble`
 - Transforms each element of the stream into a `double`.
 - `List<String> words = Arrays.asList("a", "ab", "abc");`
`DoubleStream lengths = words.stream().mapToDouble(String::length);`
- `flatMap`
 - Transforms each element of the stream into another stream, then flattens these streams into a single stream.

```
List<List<String>> nestedList = Arrays.asList(Arrays.asList("a", "b"),
Arrays.asList("c", "d"));
System.out.println(nestedList.size()); //OUTPUT: 2
List<String> collect2 = nestedList.stream().flatMap(i ->
i.stream()).peek(i -> System.out.println(i))
```



```

        .collect(Collectors.toList());
        System.out.println(collect2.size()); //OUTPUT: 4 here it is
        flatten list of list to a single stream to a,b,c,d

```

- Distinct
 - Returns a stream with duplicate elements removed.

```

List<String> items = Arrays.asList("apple", "apple", "banana");
Stream<String> distinctItems = items.stream().distinct();
distinctItems.forEach(i -> System.out.println(i));

```

/*

OUTPUT:

```

apple
banana
*/

```

- Sorted
 - Returns a stream with elements sorted in natural order or by a specified comparator.

```

List<Integer> numbers = Arrays.asList(5, 1, 3);
Stream<Integer> sortedNumbers = numbers.stream().sorted();
sortedNumbers.forEach(i->System.out.println("Natural
order:"+i));

```

```

        numbers.stream().sorted(Comparator.reverseOrder()).forEach(i-
>System.out.println("reversed order:"+i));
/*

```

Output:

```

Natural order:1
Natural order:3
Natural order:5
reversed order:5
reversed order:3
reversed order:1

```

*/

- Peek
 - Performs an action for each element of the stream, primarily for debugging purposes.
 - Syntax: Stream<T> peek(Consumer<? super T> action)
 - List<String> items = Arrays.asList("apple", "banana");
Stream<String> result = items.stream().peek(System.out::println);

- Limit
 - Returns a stream consisting of the first `n` elements.
 - `List<String> items = Arrays.asList("apple", "banana", "cherry");`
`Stream<String> limitedItems = items.stream().limit(2);`
`limitedItems.forEach(i -> System.out.println(i));` //output: apple
//banana
- Skip
 - Returns a stream with the first `n` elements discarded
 - `List<String> items = Arrays.asList("apple", "banana", "cherry");`
`Stream<String> remainingItems = items.stream().skip(1);`
`remainingItems.forEach(i -> System.out.println(i));`
`/*`

output
banana
cherry

`*/`

Terminal Operations

- `forEach`
 - Description: Performs an action for each element of the stream.
 - Example:
 - `List<String> names = Arrays.asList("Alice", "Bob", "Charlie");`
`names.stream().forEach(System.out::println);`
- `forEachOrdered`
 - Description: Performs an action for each element of the stream in the encounter order.
 - Example:
 - `List<String> names = Arrays.asList("Alice", "Bob", "Charlie");`
`names.stream().forEachOrdered(System.out::println);`
- `toArray`
 - Description: Returns an array containing the elements of the stream.
 - `List<String> names = Arrays.asList("Alice", "Bob", "Charlie");`
`String[] namesArray = names.stream().toArray(String[]::new);`
- `reduce`
 - Description: Performs a reduction on the elements of the stream, using an associative accumulation function, and returns an `Optional``.
 - `List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);`
`int sum = numbers.stream().reduce(0, Integer::sum);`
- `collect`
 - Description: Performs a mutable reduction operation on the elements of the stream using a `Collector``.

- `List<String> names = Arrays.asList("Alice", "Bob", "Charlie");`
`List<String> upperNames =`
`names.stream().map(String::toUpperCase).collect(Collectors.toList());`
- **min**
 - Description: Returns the minimum element of the stream according to a provided comparator.
 - `List<Integer> numbers = Arrays.asList(3, 1, 4, 1, 5);`
`int min =`
`numbers.stream().min(Integer::compare).orElseThrow(NoSuchElementException::new);`
- **max**
 - Description: Returns the maximum element of the stream according to a provided comparator.
 - `List<Integer> numbers = Arrays.asList(3, 1, 4, 1, 5);`
`int max =`
`numbers.stream().max(Integer::compare).orElseThrow(NoSuchElementException::new);`
- **count**
 - Description: Returns the count of elements in the stream.
 - `List<String> names = Arrays.asList("Alice", "Bob", "Charlie");`
`long count = names.stream().count();`
- **anyMatch**
 - Description: Returns `true` if any elements of the stream match the provided predicate.
 - `List<String> names = Arrays.asList("Alice", "Bob", "Charlie");`
`boolean hasAlice = names.stream().anyMatch(name -> name.equals("Alice"));`
`//OUTPUT: true`
- **allMatch**
 - Description: Returns `true` if all elements of the stream match the provided predicate.
 - `List<String> names = Arrays.asList("Alice", "Bob", "Charlie");`
`boolean allHaveA = names.stream().allMatch(name -> name.contains("a"));`
`//output:false`
- **noneMatch**
 - Description: Returns `true` if no elements of the stream match the provided predicate.
 - `List<String> names = Arrays.asList("Alice", "Bob", "Charlie");`
`boolean noneStartWithZ = names.stream().noneMatch(name -> name.startsWith("Z"));`
- **findFirst**
 - Description: Returns an `Optional` describing the first element of the stream.
 - `List<String> names = Arrays.asList("Alice", "Bob", "Charlie");`
`Optional<String> first = names.stream().findFirst();`
- **findAny**

- Description: Returns an `Optional` describing some element of the stream, or an empty `Optional` if the stream is empty.
 - `List<String> names = Arrays.asList("Alice", "Bob", "Charlie");`
`Optional<String> any = names.stream().findAny();`
-

IntStream Methods in Java 8

1. range

Generates a sequential IntStream from start (inclusive) to end (exclusive).

```
IntStream.range(1, 5).forEach(System.out::println);
```

```
// Output: 1, 2, 3, 4
```

2. rangeClosed

Generates a sequential IntStream from start (inclusive) to end (inclusive).

```
IntStream.rangeClosed(1, 5).forEach(System.out::println);
```

```
// Output: 1, 2, 3, 4, 5
```

3. sum

Calculates the sum of elements in the IntStream.

```
int sum = IntStream.of(1, 2, 3, 4).sum();
```

```
System.out.println(sum);
```

```
// Output: 10
```

4. average

Calculates the average of elements in the IntStream.

```
OptionalDouble average = IntStream.of(1, 2, 3, 4).average();
```

```
average.ifPresent(System.out::println);
```

```
// Output: 2.5
```

5. summaryStatistics

Returns an IntSummaryStatistics describing various summary data.

```
IntSummaryStatistics stats = IntStream.of(1, 2, 3, 4).summaryStatistics();
```

```
System.out.println(stats);
```

```
// Output: IntSummaryStatistics{count=4, sum=10, min=1, average=2.500000, max=4}
```

6. asDoubleStream

Converts the IntStream to a DoubleStream.

```
IntStream.of(1, 2, 3, 4).asDoubleStream().forEach(System.out::println);
```

```
// Output: 1.0, 2.0, 3.0, 4.0
```

7. asLongStream

Converts the IntStream to a LongStream.

```
IntStream.of(1, 2, 3, 4).asLongStream().forEach(System.out::println);
```

```
// Output: 1, 2, 3, 4
```

8. mapToDouble

Maps each int to a double in the resulting DoubleStream.

```
IntStream.of(1, 2, 3, 4).mapToDouble(i -> i * 2.0).forEach(System.out::println);
```

```
// Output: 2.0, 4.0, 6.0, 8.0
```

9. mapToLong

Maps each int to a long in the resulting LongStream.

```
IntStream.of(1, 2, 3, 4).mapToLong(i -> i * 2L).forEach(System.out::println);
```

```
// Output: 2, 4, 6, 8
```

10. mapToObj

Maps each int to an object in the resulting Stream.

```
IntStream.of(1, 2, 3, 4).mapToObj(Integer::toString).forEach(System.out::println);
```

```
// Output: "1", "2", "3", "4"
```

11. boxed

Converts the IntStream to a Stream<Integer>.

```
java
```

```
IntStream.of(1, 2, 3, 4).boxed().forEach(System.out::println);
```

```
// Output: 1, 2, 3, 4
```

All functional interfaces:

Interface Summary

BiConsumer<T,U>

Represents an operation that accepts two input arguments and returns no result.

BiFunction<T,U,R>

Represents a function that accepts two arguments and produces a result.

BinaryOperator<T>

Represents an operation upon two operands of the same type, producing a result of the same type as the operands.

BiPredicate<T,U>

Represents a predicate (boolean-valued function) of two arguments.

BooleanSupplier

Represents a supplier of boolean-valued results.

Consumer<T>

Represents an operation that accepts a single input argument and returns no result.

DoubleBinaryOperator

Represents an operation upon two double-valued operands and producing a double-valued result.

DoubleConsumer

Represents an operation that accepts a single double-valued argument and returns no result.

DoubleFunction<R>

Represents a function that accepts a double-valued argument and produces a result.

DoublePredicate

Represents a predicate (boolean-valued function) of one double-valued argument.

DoubleSupplier

Represents a supplier of double-valued results.

DoubleToIntFunction

Represents a function that accepts a double-valued argument and produces an int-valued result.

DoubleToLongFunction

Represents a function that accepts a double-valued argument and produces a long-valued result.

DoubleUnaryOperator

Represents an operation on a single double-valued operand that produces a double-valued result.

Function<T,R>

Represents a function that accepts one argument and produces a result.

IntBinaryOperator

Represents an operation upon two int-valued operands and producing an int-valued result.

IntConsumer

Represents an operation that accepts a single int-valued argument and returns no result.

IntFunction<R>

Represents a function that accepts an int-valued argument and produces a result.

IntPredicate

Represents a predicate (boolean-valued function) of one int-valued argument.

IntSupplier

Represents a supplier of int-valued results.

IntToDoubleFunction

Represents a function that accepts an int-valued argument and produces a double-valued result.

IntToLongFunction

Represents a function that accepts an int-valued argument and produces a long-valued result.

IntUnaryOperator

Represents an operation on a single int-valued operand that produces an int-valued result.

LongBinaryOperator

Represents an operation upon two long-valued operands and producing a long-valued result.

LongConsumer

Represents an operation that accepts a single long-valued argument and returns no result.

LongFunction<R>

Represents a function that accepts a long-valued argument and produces a result.

LongPredicate

Represents a predicate (boolean-valued function) of one long-valued argument.

LongSupplier

Represents a supplier of long-valued results.

LongToDoubleFunction

Represents a function that accepts a long-valued argument and produces a double-valued result.

LongToIntFunction

Represents a function that accepts a long-valued argument and produces an int-valued result.

LongUnaryOperator

Represents an operation on a single long-valued operand that produces a long-valued result.

ObjDoubleConsumer<T>

Represents an operation that accepts an object-valued and a double-valued argument, and returns no result.

ObjIntConsumer<T>

Represents an operation that accepts an object-valued and a int-valued argument, and returns no result.

ObjLongConsumer<T>

Represents an operation that accepts an object-valued and a long-valued argument, and returns no result.

Predicate<T>

Represents a predicate (boolean-valued function) of one argument.

Supplier<T>

Represents a supplier of results.

ToDoubleBiFunction<T,U>

Represents a function that accepts two arguments and produces a double-valued result.

ToDoubleFunction<T>

Represents a function that produces a double-valued result.

ToIntBiFunction<T,U>

Represents a function that accepts two arguments and produces an int-valued result.

ToIntFunction<T>

Represents a function that produces an int-valued result.

ToLongBiFunction<T,U>

Represents a function that accepts two arguments and produces a long-valued result.

ToLongFunction<T>

Represents a function that produces a long-valued result.

UnaryOperator<T>

Represents an operation on a single operand that produces a result of the same type as its operand.

Here is a detailed guide on each functional interface from the `java.util.function`` package, including their single abstract method (SAM) syntax and examples for easy reference:

1. BiConsumer<T, U>

- Abstract Method: ``void accept(T t, U u)``

- Example:

```
BiConsumer<Integer, Integer> add = (a, b) -> System.out.println(a + b);  
add.accept(2, 3); // Output: 5
```

2. BiFunction<T, U, R>

- Abstract Method: ``R apply(T t, U u)``
- Example:

```
BiFunction<Integer, Integer, Integer> multiply = (a, b) -> a * b;  
System.out.println(multiply.apply(2, 3)); // Output: 6
```

3. BinaryOperator<T>

- Abstract Method: ``T apply(T t1, T t2)``
- Example:

```
BinaryOperator<Integer> max = (a, b) -> a > b ? a : b;  
System.out.println(max.apply(2, 3)); // Output: 3
```

4. BiPredicate<T, U>

- Abstract Method: ``boolean test(T t, U u)``
- Example:

```
BiPredicate<String, String> equal = (a, b) -> a.equals(b);  
System.out.println(equal.test("test", "test")); // Output: true
```

5. BooleanSupplier

- Abstract Method: ``boolean getAsBoolean()``
- Example:

```
BooleanSupplier supplier = () -> true;  
System.out.println(supplier.getAsBoolean()); // Output: true
```

6. Consumer<T>

- Abstract Method: ``void accept(T t)``
- Example:

```
Consumer<String> print = System.out::println;  
print.accept("Hello, World!"); // Output: Hello, World!
```

7. DoubleBinaryOperator

- Abstract Method: ``double applyAsDouble(double left, double right)``
- Example:

```
DoubleBinaryOperator multiply = (a, b) -> a * b;  
System.out.println(multiply.applyAsDouble(2.0, 3.0)); // Output: 6.0
```

8. DoubleConsumer

- Abstract Method: ``void accept(double value)``

- Example:

```
DoubleConsumer printDouble = System.out::println;  
printDouble.accept(3.14); // Output: 3.14
```

9. DoubleFunction<R>

- Abstract Method: ``R apply(double value)``

- Example:

```
DoubleFunction<String> doubleToString = Double::toString;  
System.out.println(doubleToString.apply(3.14)); // Output: 3.14
```

10. DoublePredicate

- Abstract Method: ``boolean test(double value)``

- Example:

```
DoublePredicate isPositive = value -> value > 0;  
System.out.println(isPositive.test(3.14)); // Output: true
```

11. DoubleSupplier

- Abstract Method: ``double getAsDouble()``

- Example:

```
DoubleSupplier piSupplier = () -> 3.14;  
System.out.println(piSupplier.getAsDouble()); // Output: 3.14
```

12. DoubleToIntFunction

- Abstract Method: ``int applyAsInt(double value)``

- Example:

```
DoubleToIntFunction doubleToInt = value -> (int) value;  
System.out.println(doubleToInt.applyAsInt(3.14)); // Output: 3
```

13. DoubleToLongFunction

- Abstract Method: ``long applyAsLong(double value)``

- Example:

```
DoubleToLongFunction doubleToLong = value -> (long) value;  
System.out.println(doubleToLong.applyAsLong(3.14)); // Output: 3
```

14. DoubleUnaryOperator

- Abstract Method: ``double applyAsDouble(double operand)``

- Example:

```
DoubleUnaryOperator square = value -> value * value;  
System.out.println(square.applyAsDouble(3.14)); // Output: 9.8596
```

15. Function<T, R>

- Abstract Method: ``R apply(T t)``
- Example:

```
Function<String, Integer> length = String::length;  
System.out.println(length.apply("Hello")); // Output: 5
```

16. IntBinaryOperator

- Abstract Method: ``int applyAsInt(int left, int right)``
- Example:

```
IntBinaryOperator add = (a, b) -> a + b;  
System.out.println(add.applyAsInt(2, 3)); // Output: 5
```

17. IntConsumer

- Abstract Method: ``void accept(int value)``
- Example:

```
IntConsumer printInt = System.out::println;  
printInt.accept(5); // Output: 5
```

18. IntFunction<R>

- Abstract Method: ``R apply(int value)``
- Example:

```
IntFunction<String> intToString = Integer::toString;  
System.out.println(intToString.apply(5)); // Output: 5
```

19. IntPredicate

- Abstract Method: ``boolean test(int value)``
- Example:

```
IntPredicate isEven = value -> value % 2 == 0;  
System.out.println(isEven.test(4)); // Output: true
```

20. IntSupplier

- Abstract Method: ``int getAsInt()``
- Example:

```
IntSupplier randomInt = () -> (int) (Math.random() * 100);  
System.out.println(randomInt.getAsInt()); // Output: random int between 0 and 99
```

21. IntToDoubleFunction

- Abstract Method: ``double applyAsDouble(int value)``
- Example:

```
IntToDoubleFunction half = value -> value / 2.0;  
System.out.println(half.applyAsDouble(5)); // Output: 2.5
```

22. IntToLongFunction

- Abstract Method: ``long applyAsLong(int value)``
- Example:

```
IntToLongFunction intToLong = value -> (long) value;  
System.out.println(intToLong.applyAsLong(5)); // Output: 5
```

23. IntUnaryOperator

- Abstract Method: ``int applyAsInt(int operand)``
- Example:

```
IntUnaryOperator increment = value -> value + 1;  
System.out.println(increment.applyAsInt(5)); // Output: 6
```

24. LongBinaryOperator

- Abstract Method: ``long applyAsLong(long left, long right)``
- Example:

```
LongBinaryOperator add = (a, b) -> a + b;  
System.out.println(add.applyAsLong(2L, 3L)); // Output: 5L
```

25. LongConsumer

- Abstract Method: ``void accept(long value)``
- Example:

```
LongConsumer printLong = System.out::println;  
printLong.accept(10L); // Output: 10
```

26. LongFunction<R>

- Abstract Method: ``R apply(long value)``
- Example:

```
LongFunction<String> longToString = Long::toString;  
System.out.println(longToString.apply(10L)); // Output: 10
```

27. LongPredicate

- Abstract Method: `boolean test(long value)`
- Example:

```
LongPredicate isPositive = value -> value > 0;  
System.out.println(isPositive.test(10L)); // Output: true
```

28. LongSupplier

- Abstract Method: `long getAsLong()`
- Example:

```
LongSupplier randomLong = () -> (long) (Math.random() * 1000);  
System.out.println(randomLong.getAsLong()); // Output: random long between 0 and 999
```

Sure, here's a detailed document describing each functional interface in the `java.util.function` package, including syntax, a clear example, and details about their single abstract method.

1. `Predicate<T>`

Represents a boolean-valued function of one argument.

Single Abstract Method: `boolean test(T t);`

Example:

```
Predicate<Integer> isEven = (n) -> n % 2 == 0;  
System.out.println(isEven.test(4)); // Output: true  
System.out.println(isEven.test(5)); // Output: false
```

2. `Function<T, R>`

Represents a function that accepts one argument and produces a result.

Single Abstract Method: `R apply(T t);`

Example:

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

3. `Consumer<T>`

Represents an operation that accepts a single input argument and returns no result.

Single Abstract Method: `void accept(T t);`

Example:

```
Consumer<String> printConsumer = (s) -> System.out.println(s);  
printConsumer.accept("Hello World!"); // Output: Hello World!
```

4. `Supplier<T>`

Represents a supplier of results.

Single Abstract Method: `T get();`

Example:

```
Supplier<Double> randomSupplier = () -> Math.random();  
System.out.println(randomSupplier.get());
```

5. `UnaryOperator<T>`

Represents an operation on a single operand that produces a result of the same type as its operand.

Single Abstract Method: `T apply(T t);`

Example:

```
UnaryOperator<Integer> square = (n) -> n * n;  
System.out.println(square.apply(5)); // Output: 25
```

6. `BinaryOperator<T>`

Represents an operation upon two operands of the same type, producing a result of the same type as the operands.

Single Abstract Method: `T apply(T t1, T t2);`

Example:

```
BinaryOperator<Integer> add = (a, b) -> a + b;  
System.out.println(add.apply(5, 3)); // Output: 8
```

7. `BiPredicate<T, U>`

Represents a predicate (boolean-valued function) of two arguments.

Single Abstract Method: `boolean test(T t, U u);`

Example:

```
BiPredicate<String, Integer> checkLength = (s, i) -> s.length() == i;
System.out.println(checkLength.test("Hello", 5)); // Output: true
System.out.println(checkLength.test("Hello", 4)); // Output: false
```

8. `BiFunction<T, U, R>`

Represents a function that accepts two arguments and produces a result.

Single Abstract Method: `R apply(T t, U u);`

Example:

```
BiFunction<Integer, Integer, String> sumAndConvert = (a, b) -> String.valueOf(a + b);
System.out.println(sumAndConvert.apply(3, 4)); // Output: "7"
```

9. `BiConsumer<T, U>`

Represents an operation that accepts two input arguments and returns no result.

Single Abstract Method: `void accept(T t, U u);`

Example:

```
BiConsumer<String, Integer> printDetails = (name, age) -> System.out.println(name + " is " + age +
" years old.");
printDetails.accept("John", 25); // Output: John is 25 years old.
```