JAVA STREAMS

Streams: Special type of sequences in Java on which some processing and operations can be applied.

- Streams take input from Collections, Arrays or I/O.
- Streams don’t change the original data.

Types of operations on streams:

1. Intermediate operations
2. Terminal operations

Example creation of a stream:

```
import java.util.List;
import java.util.stream.Stream;

public class Streams {
    public static void main(String[] args) {
        List<String> langs = List.of("Java", "Python", "C++", "Ruby");

        Stream<String> streamLangs = langs.stream();

        System.out.println(streamLangs.getClass());
    }
}
```

Output: class java.util.stream.ReferencePipeline$Head

Parallel Processing:
Auto creation and handling of threads and thus enabling parallel processing on large amounts of data.

```
Stream<String> streamLangs = langs.parallelStream();
```

Advantages if Streams over Collections:

- Modifying the stream data won’t affect the original collection data.
- Once a stream is consumed (i.e., a terminal operation has been performed), it can’t be reused.

```
import java.util.List;
import java.util.stream.Stream;

public class Streams {
    public static void main(String[] args) {
        List<String> langs = List.of("Java", "Python", "C++", "Ruby");

        Stream<String> streamLangs = langs.stream();

        Long count = streamLangs.count();   // Terminal operation
        System.out.println(count);

        streamLangs.forEach(cnsmr -> System.out.println(cnsmr));
    }
}
```

Output:

```
4
Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed
        at java.base/java.util.stream.AbstractPipeline.sourceStageSpliterator(AbstractPipeline.java:279)
        at java.base/java.util.stream.ReferencePipeline$Head.forEach(ReferencePipeline.java:762)
        at Streams.main(Streams.java:13)
```

Intermediate Operations:
These are lazy operations and they won't be executed unless a terminal operation exists.

1. **map( ):** Transforms each element
2. **filter( ):** Filters elements based on a condition
3. **flatMap( ):** Flattens nested structures
4. **distinct( ):** Removes duplicates
5. **sorted( ):** Sorts elements
6. **peek( ):** Performs an action without modifying the stream
7. **limit(n):** Limit to the first n elements
8. **skip(n):** Skip the first n elements

Terminal Operations:
A stream is said to be consumed if a terminal operation is executed on it, and it can't be reused.

1. **collect( ):** Collects the stream elements into a collection
2. **forEach( ):** Performs an action on each element
3. **reduce( ):** Combines elements into a single result
4. **count( ):** Counts the no. of elements in a stream
5. **anyMatch( ):** Returns a true if any element passes a condition
6. **allMatch( ):** Returns a true if all elements pass a condition
7. **noneMatch( ):** Checks if no elements match a condition
8. **findFirst( ):** Returns the first element
9. **findAny( ):** Returns any element (useful in parallel streams)
10. **toArray( ):** Convert a stream into an Array