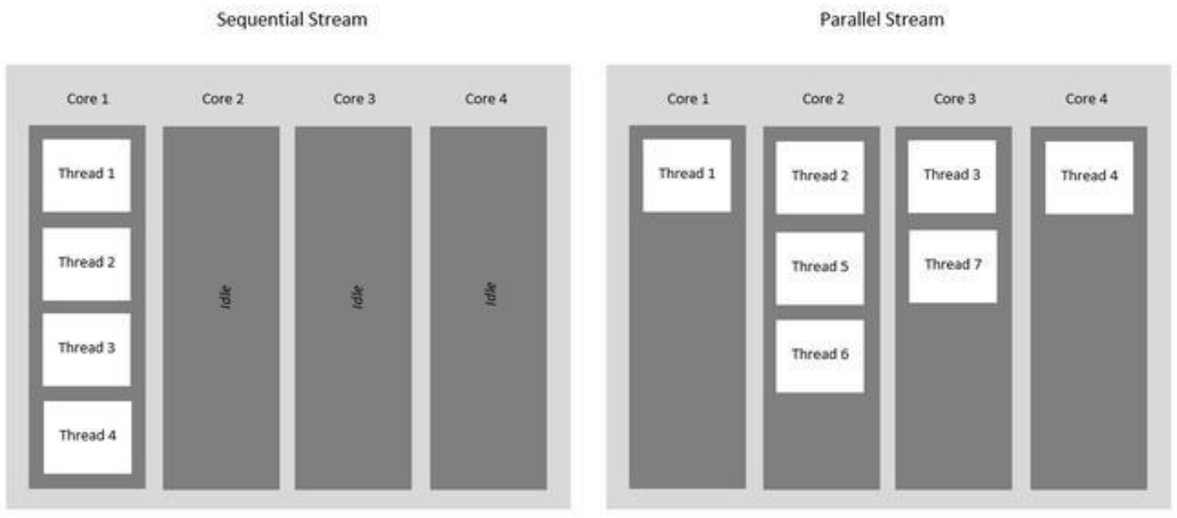
Parallel vs Sequential Stream in Java

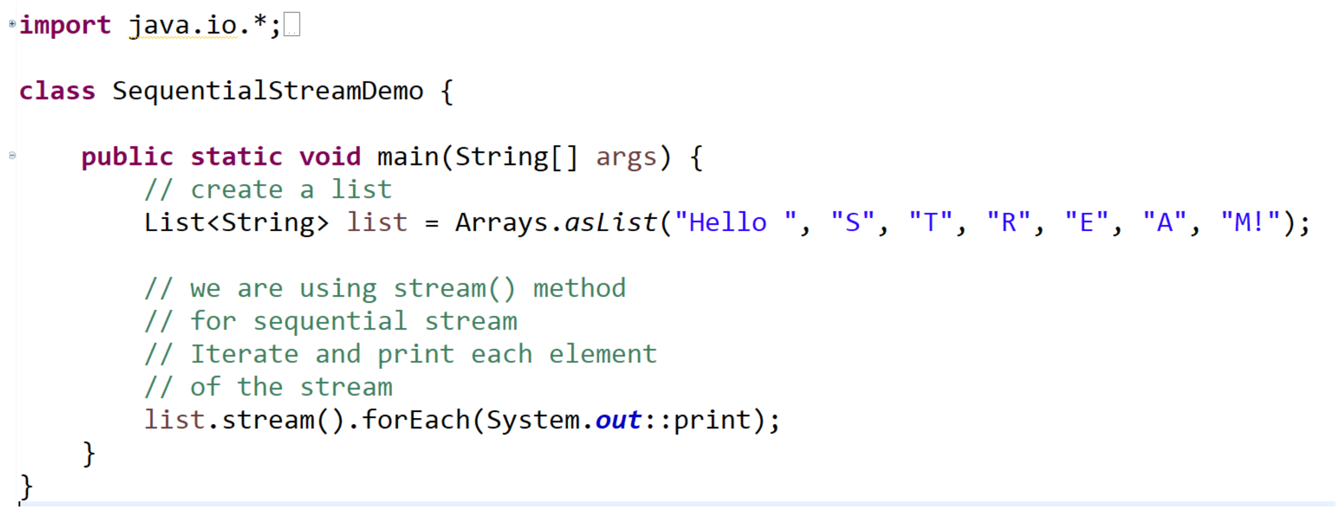
A stream in Java is a sequence of objects which operates on a data source such as an array or a collection and supports various methods. It was introduced in Java 8's java.util.stream package. Stream supports many aggregate operations like filter, map, limit, reduce, find, and match to customize the original data into a different form according to the need of the programmer. The operations performed on a stream do not modify its source hence a new stream is created according to the operation applied to it. The new data is a transformed copy of the original form.



Sequential Stream

Sequential Streams are non-parallel streams that use a single thread to process the pipelining. Any stream operation without explicitly specified as parallel is treated as a sequential stream. Sequential stream's objects are pipelined in a single stream on the same processing system hence it never takes the advantage of the multi-core system even though the underlying system supports parallel execution. Sequential stream performs operation one by one.

stream() method returns a sequential stream in Java.



```java
import java.io.*;

class SequentialStreamDemo {

    public static void main(String[] args) {
        // create a list
        List<String> list = Arrays.asList("Hello ", "S", "T", "R", "E", "A", "M!");

        // we are using stream() method
        // for sequential stream
        // Iterate and print each element
        // of the stream
        list.stream().forEach(System.out::print);
    }
}
```

In this example the list.stream() works in sequence on a single thread with the print() operation and in the output of the preceding program, the content of the list is printed in an ordered sequence as this is a sequential stream.

Parallel Stream

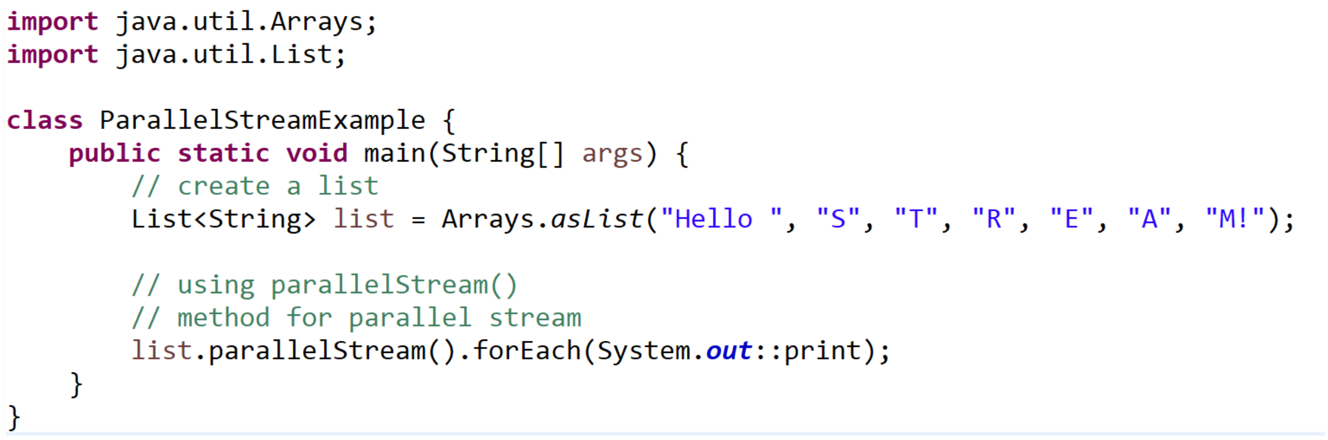
It is a very useful feature of Java to use parallel processing, even if the whole program may not be parallelized. Parallel stream leverage multi-core processors, which increases its performance. Using parallel streams, our code gets divide into multiple streams which can be executed parallelly on separate cores of the system and the final result is shown as the combination of all the individual core’s outcomes. It is always not necessary that the whole program be parallelized, but at least some parts should be parallelized which handles the stream. The order of execution is not under our control and can give us unpredictably unordered results and like any other parallel programming, they are complex and error-prone.

The Java stream library provides a couple of ways to do it. easily, and in a reliable manner.

One of the simple ways to obtain a parallel stream is by invoking the parallelStream() method of Collection interface.

Another way is to invoke the parallel() method of BaseStream interface on a sequential stream.

It is important to ensure that the result of the parallel stream is the same as is obtained through the sequential stream, so the parallel streams must be stateless, non-interfering, and associative.



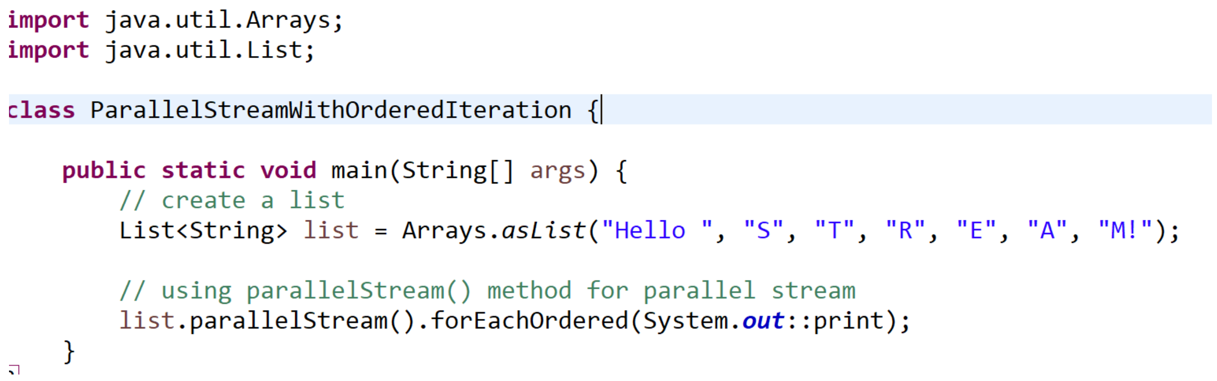
```java
import java.util.Arrays;
import java.util.List;

class ParallelStreamExample {
    public static void main(String[] args) {
        // create a list
        List<String> list = Arrays.asList("Hello ", "S", "T", "R", "E", "A", "M!");

        // using parallelStream()
        // method for parallel stream
        list.parallelStream().forEach(System.out::print);
    }
}
```

Output:

```
ERM!ASHello T
```

Here we can see the order is not maintained as the list.parallelStream() works parallelly on multiple threads. If we run this code multiple times then we can also see that each time we are getting a different order as output but this parallel stream boosts the performance so the situation where the order is not important is the best technique to use.

Note: If we want to make each element in the parallel stream to be ordered, we can use the forEachOrdered() method, instead of the forEach() method.



```java
import java.util.Arrays;
import java.util.List;

class ParallelStreamWithOrderedIteration {

    public static void main(String[] args) {
        // create a list
        List<String> list = Arrays.asList("Hello ", "S", "T", "R", "E", "A", "M!");

        // using parallelStream() method for parallel stream
        list.parallelStream().forEachOrdered(System.out::print);
    }
```

Output:

```
Hello STREAM!
```

We can always switch between parallel and sequential very easily according to our requirements. If we want to change the parallel stream as sequential, then we should use the sequential() method specified by BaseStream Interface.

Differences between Sequential Stream and Parallel Stream

Sequential Stream	Parallel Stream
Runs on a single-core of the computer	Utilize the multiple cores of the computer.
Performance is poor	The performance is high.
Order is maintained	Doesn't care about the order,
Only a single iteration at a time just like the for-loop.	Operates multiple iterations simultaneously in different available cores.
Each iteration waits for currently running one to finish,	Waits only if no cores are free or available at a given time,
More reliable and less error,	Less reliable and error-prone.
Platform independent,	Platform dependent

Conclusion

The stream APIs have been a part of Java for a long time for its intriguing feature. It is also very much popular for parallel processing capability and improved performance. In this era literally, every modern machine is multi-core so to this core efficiently we should use parallel streams however parallel programming design is complex. So it’s completely up to the programmer whether he wants to use parallel streams or sequential streams based on the requirements.