Stream API

Two Core and new features of Java: Lambda expressions and Stream API.
The stream API is designed on the basis of lambda expressions.

The key aspect of stream API is its ability to perform operations that search, filter, map, and manipulate data in many other ways.
Say, we can construct a sequence of operations on the data and such actions can also be done in parallel, thereby increasing the efficiency of processing in case of large datasets.

Stream API provides a powerful way of handling data easily.

Stream Basics

1. A stream is a channel for data.
2. A stream represents a sequence of objects.
3. A stream operates on a data source [array/collection].
4. A stream, by itself, never provides storage for the data.
5. A stream just moves data, possibly filtering, sorting, and doing other operations on the data in the process.
6. A stream operation, as a rule, by itself, does not modify the data source. Say, sorting a stream results in the creation of a new stream that produces the sorted result, rather than changing the original source.

Stream Interfaces

The stream API defines several stream interfaces packaged in java.util.stream.

The top interface is BaseStream which defines the basic functionality available in all streams
$ interface BaseStream<T, S extends BaseSteam<T, S>>
T specifies the type of elements in the stream
S specifies the type of stream that extends the BaseStream

Methods in BaseStream Interface

1 void close() Closes the invoking stream
2 S parallel() Returns a parallel stream based on the invoking stream [Intermediate operation]
3 S sequential() Returns a sequential stream based on the invoking stream [Intermediate operation]
4 S unordered() Returns an unordered stream based on the invoking stream [Intermediate operation]
5 Iterator<T> iterator
Obtains an iterator to the stream and returns a reference to the iterator. [Terminal operation]
6 boolean isParallel()
Returns true if invoking stream is parallel, else false if the stream is sequential

Methods in Stream Interface

There are several types of stream interfaces that are derived from BaseStream. The most general of these subinterfaces is Stream: interface Stream<T>
T specifies the type of elements in the stream.

1 long count()
Counts and returns the number of elements in the stream [Terminal operation]

2 Stream <T> filter(Predicate<? super T> p)
Produces a stream that contains those elements from the invoking stream that satisfy the specified predicate p [Intermediate operation]

3 void forEach(Consumer<? super T> action)
For each element in the invoking stream, the code specified by action is executed [Terminal operation]

4 <R> Stream<R> map(Function<? super T, ? extends R> mapfunction)
Applies mapfunction to the elements from the invoking stream, yielding a new stream that contains those elements [Intermediate operation]

5 Optional<T> max(Comparator<? Super T> c)
Using the ordering specified by c, find, and return maximum element in the invoking stream [Terminal operation]

6 Optional<T> min(Comparator<? Super T> c)
Using the ordering specified by c, find, and return minimum element in the invoking stream [Terminal operation]

7 Stream<T> sorted()
Produces a new stream that contains the elements of the invoking stream sorted in natural order [Intermediate operation]

8 T reduce(T val, BinaryOperator<T> acc)
Returns a result based on the elements in the invoking stream, called a reduction operation [Terminal operation]

9 Object[] toArray()
Creates an array from the elements in the invoking stream [Terminal operation]

10 default List<T> toList()
Creates an unmodifiable List from the elements in invoking stream
[Terminal operation]

Few Key Things

- Many methods of stream API are either terminal or intermediate operations.
- A terminal operation consumes the stream and is used to produce a result or to execute some action on each object of the stream.
- Once a stream is consumed it cannot be reused.
- An intermediate operation produces another stream and can be used to create a pipeline that performs a sequence of operations.

Intermediate operations do not execute immediately – The specified action is performed when a terminal operation is ececuted on the new stream created by an intermediate operation – This is known as lazy behavior and the intermediate operations are known as lazy operations.
(The lazy behavior enables the stream API to perform more efficiently)

- Some intermediate operations are stateless, and some are stateful.
- In a stateless operation, each element is processed independently of the others.
- In a stateful operation, the processing of an element may depend on aspects of the other elements.
- Say, Sorting is a stateful operation because an element's order depends on the values of the other elements. [sorted() method is stateful]
- Say, filtering elements based on a stateless predicate is stateless because each element is handled individually. [filter() method is stateless]

Because Stream operates on object references, it cannot operate directly on primitive types:
To handle primitive type streams, the stream API defines the following interfaces: DoubleStream, IntStream, LongStream
These streams extend BaseStream and have similar capabilities to Stream, but they operate on primitive types rather than reference types.

Obtain a Stream

A stream can be obtained in several ways.
The most common way is when a stream is obtained for a collection.

The Collection interface has two methods to obtain a stream from a collection.

1 default Stream<E> stream()
  Its default implementation returns a sequential stream
2 default Stream<E> parallelStream()
  Its default implementation returns a parallel stream, if possible, otherwise a sequential stream. A parallel stream supports parallel execution of stream operations.

As the Collection interface is implemented by every collection, these 2 methods can be used to obtain a stream from any collection class, say, ArrayList.

A stream can also be obtained from an array using the static stream() method of the Arrays class

1 static <T> Stream<T> stream(T[] array)
  This method returns a sequential stream to the elements in given array.

Example: Stream<Employee> st = Arrays.stream(employees)
         where employees is an array of type Employee,
         Employee[] employees = { new Employee(..), new Employee(..), .. };

Several overloaded stream() methods are defined in Arrays class which handle arrays of primitive types and return appropriate streams.

Streams can also be obtained in other ways – say, many stream operations return a new stream – A stream to an I/O source can be obtained by using lines() method on a BufferedReader.

Irrespective of the way a stream is obtained, all streams are used in same manner.

Reduction Operations

The min(), max(), count() methods are terminal operations that return a result based on the elements in the stream.
Such operations represent reduction operations because each reduces a stream to a single value.
Stream API refers them as special case reductions because they perform a specific function.

The stream API generalizes this concept of reduction by providing the reduce() method, using which we can return a value from a stream based on any arbitrary criteria.

By definition, all reduction operations are terminal operations.

1 Optional<T> reduce(BinaryOperator<T> accumulator)
2 T reduce(T identityVal, BinaryOperator<T> acuumulator)

- The first form returns an object of type Optional which contains the result.
- The second form returns an object of type T which is element type of stream, and identityVal is a value such that an accumulator operation involving identityVal and any element of the stream yields that element unchanged.
- Say, if operation is addition, then identityVal will be 0 because 0+x is x.
- Say, for multiplication, the value will be 1, because 1*x is x.
- In both forms, accumulator is a function that operates on two values and produces a result.

- BinaryOperator is a functional interface declared in java.util.function that extends the BiFunction functional interface.
- BiFunction defines the abstract method: R apply(T val1, U val2)
- R specifies the result type; T is type of first operand; U type of second one;
- The apply() applies a function to its operands and returns the result.

- When BinaryOperator extends BiFunction it specifies the same type for all type parameters: T apply(T val, T val2)
- Also, val will contain the previous result and val2 will contain the next element.
- In the first invocation, val will contain either the idenity value or the first element, depending on which version of reduce() is used.

The accumulator operation must satisfy three constraints:
Stateless, Non-interfering, Associative

Stateless means that the operation does not depend on any state information and each element is processed independently.
Non-interfering means that the data source is not modified by the operation.
Associative means that given an associative operator used in a sequence of operations, it does not matter which pair of operands are processed first.

Mapping Operations

Often it is useful to map the elements of one stream to another.

Say, we want to apply some transformation to the elements in a stream, and for this we can map the transformed elements to a new stream.

As mapping operations are common, the stream API provides built-in support for them. The most general mapping method is map()

<R> Stream<R> map(Function<? super T, ? extends R> mapfunction)

R specifies the type of elements of new stream; T is the type of elements of invoking stream; mapfunction is an instance of Function which does the mapping.

The mapfunction must be stateless and non-interfering.
As a new stream is returned, map() is an intermediate operation.

Function is a functional interface declared in java.util.function and is declared as: Function<T, R>
Function has the abstract method: R apply(T value)
Here, value is the reference to the object being mapped and the mapped result is returned.

Collection Operations

It is always possible to obtain a stream from a collection.
Sometimes it is needed to obtain a collection from a stream.

To do such a thing, use the collect() method
<R, A> R collect(Collector<? Super T, A, R> collfunction)

The collect() method is a terminal operation.

The Collector interface is declared in java.util.stream:
interface Collector<T, A, R>

Collector interface specifies several methods.

Two predefined collectors are provided by the Collectors class in java.util.stream

The Collectors class defines a number of static collector methods – Two of these are:
1 static <T> Collector<T, ?, List<T>> toList()
2 static <T> Collector<T, ?, Set<T>>   toSet()

StreamDemo.java

```
import java.util.*;
import java.util.stream.*;

public class StreamDemo
{
      public static void main(String[] args)
      {
            ArrayList<Integer> al = new ArrayList<>();
            al.add(7); al.add(18); al.add(10);
            al.add(24); al.add(17); al.add(5);

            System.out.println("Actual List: " + al);

            Stream<Integer> stm = al.stream();  // get a stream; factory method

            Optional<Integer> least = stm.min(Integer::compare);
            // Integer class implements Comparator [compare(obj1, obj2)
            // static method]

            if(least.isPresent())
                  System.out.println("Minimum integer: " + least.get());

            // min is a terminal operation, get the stream again

            /* Optional<T> is a generic class packaged in java.util package.
               An Optional class instance can either contains a value of
               type T or is empty.
               Use method isPresent() to check if a value is present.
```

```
            Obtain the value by calling get()
         */

         stm = al.stream();
         System.out.println("Available values: " + stm.count());

         stm = al.stream();
         Optional<Integer> higher = stm.max(Integer::compare);
         if(higher.isPresent())
              System.out.println("Maximum integer: " + higher.get());

         // max is a terminal operation, get stream again

         Stream<Integer> sortedStm = al.stream().sorted();
         // sorted() is intermediate operation
         // here, we obtain a sorted stream
         System.out.print("Sorted stream: ");
         sortedStm.forEach(n -> System.out.print(n + " "));
         // lambda expression
         System.out.println();

         /* Consumer<T> is a  generic functional interface
            declared in java.util.function
            Its abstract method is: void accept(T obj)
            The lambda expression in the call to forEach()
            provides the implementation of accept() method.
         */

         Stream<Integer> odds = al.stream().sorted().filter(n -> (n%2)==1);
         System.out.print("Odd values only: ");
         odds.forEach(n -> System.out.print(n + " "));
         System.out.println();

         /* Predicate<T> is a generic functional interface
            defines in java.util.function
            and its abstract method is test(): boolean test(T obj)
            Retruns true if object for test() statisfies the predicate
            and false otherwise.
            The lambda expression passed to the filter() implements
            this method.
      */

         odds = al.stream().sorted().filter(n -> (n%2)==1).filter(n -> n>5);
         // possible to chain the filters
         System.out.print("Odd values bigger than 5 only: ");
         odds.forEach(n -> System.out.print(n + " "));
         System.out.println();
      }
}
```

ReductionDemo.java

```
import java.util.*;
import java.util.stream.*;

public class ReductionDemo
{
	public static void main(String[] args)
	{
		ArrayList<Integer> al = new ArrayList<>();
		al.add(10); al.add(20); al.add(30); al.add(40); al.add(50);

		System.out.println("Contents of the collection: " + al);
		System.out.println();

		Optional<Integer> obj1 = al.stream().reduce((x, y) -> (x+y));
		if(obj1.isPresent())
			System.out.println("Total is: " + obj1.get());

		System.out.println();

		long product = al.stream().reduce(1, (x, y) -> (x*y));
		System.out.println("Product is: " + product);
	}
}
```

MappingDemo.java

```
import java.util.*;
import java.util.stream.*;

public class MappingDemo
{
	public static void main(String[] args)
	{
		ArrayList<Integer> al = new ArrayList<>();
		al.add(5); al.add(16); al.add(25); al.add(30);
		al.add(49); al.add(100);

		System.out.println("Actual List: " + al);

		Stream<Double> sqr = al.stream().map(n -> Math.sqrt(n));
		System.out.print("Square roots: ");

		sqr.forEach(n -> System.out.print("["+n+"] "));
		System.out.println();
	}
}
```

CollectingDemo.java

```
import java.util.*;
import java.util.stream.*;

public class CollectingDemo
{
      public static void main(String[] args)
      {
            ArrayList<Integer> al = new ArrayList<>();
            al.add(5); al.add(7); al.add(7);
            al.add(30); al.add(49); al.add(100);
            System.out.println("Actual List: " + al);

            Stream<Integer> odds = al.stream().filter(n -> n%2==1);
            System.out.print("Odd Numbers: ");
            odds.forEach(n -> System.out.print(n+" "));
            System.out.println();

            List<Integer> oddList =
                  al.stream().filter(n -> n%2==1).collect(Collectors.toList());

            System.out.println("The list: " + oddList);

            Set<Integer> oddSet =
                  al.stream().filter(n -> n%2==1).collect(Collectors.toSet());

            System.out.println("The set: " + oddSet);
      }
}
```