Streams and Parallel Streams

Of the many new features added by JDK 8, the two that are, arguably, the most important are lambda expressions and the stream API.

The key aspect of the stream API is its ability to perform very sophisticated operations that search, filter, map, or otherwise manipulate data.

For example, using the stream API, you can construct sequences of actions that resemble, in concept, the type of database queries for which you might use SQL.

Furthermore, in many cases, such actions can be performed in parallel, thus providing a high level of efficiency, especially when large data sets are involved.

Put simply, the stream API provides a powerful means of handling data in an efficient, yet easy to use way.

The features of Java stream are

- A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.
- Streams don't change the original data structure; they only provide the result as per the pipelined methods.
- Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

Stream Basics

Let's begin by defining the term stream as it applies to the stream API: a stream is a conduit for data. Thus, a stream represents a sequence of objects.

A stream operates on a data source, such as an array or a collection.

A stream, itself, never provides storage for the data.

It simply moves data, possibly filtering, sorting, or otherwise operating on that data in the process.

As a rule, however, a stream operation by itself does not modify the data source.

For example, sorting a stream does not change the order of the source. Rather, sorting a stream result in the creation of a new stream that produces the sorted result.

Stream Interfaces

The stream API defines several stream interfaces, which are packaged in **java.util.stream** . At the foundation is **BaseStream**, which defines the basic functionality available in all streams.

BaseStream is a generic interface declared like this:

```
interface BaseStream<T, S extends BaseStream<T, S>>
```

Method	Description
void close( )	Closes the invoking stream, calling any registered close handlers. (As explained in the text, few streams need to be closed.)
boolean isParallel( )	Returns **true** if the invoking stream is parallel. Returns **false** if the stream is sequential.
Iterator<T> iterator( )	Obtains an iterator to the stream and returns a reference to it. (Terminal operation.)
S onClose(Runnable *handler*)	Returns a new stream with the close handler specified by *handler*. This handler will be called when the stream is closed. (Intermediate operation.)
S parallel( )	Returns a parallel stream based on the invoking stream. If the invoking stream is already parallel, then that stream is returned. (Intermediate operation.)
S sequential( )	Returns a sequential stream based on the invoking stream. If the invoking stream is already sequential, then that stream is returned. (Intermediate operation.)
Spliterator<T> spliterator( )	Obtains a spliterator to the stream and returns a reference to it. (Terminal operation.)
S unordered( )	Returns an unordered stream based on the invoking stream. If the invoking stream is already unordered, then that stream is returned. (Intermediate operation.)

Table 29-1 The Methods Declared by **BaseStream**

From BaseStream are derived several types of stream interfaces. The most general of these is Stream. It is declared as shown here:

```
interface Stream<T>
```

In addition to the methods that it inherits from BaseStream, the Stream interface adds several of its own, a sampling of which is shown in Table 29-2.

Method	Description
<R, A> R collect(Collector<? super T, A, R> *collectorFunc*)	Collects elements into a container, which is changeable, and returns the container. This is called a mutable reduction operation. Here, **R** specifies the type of the resulting container and **T** specifies the element type of the invoking stream. **A** specifies the internal accumulated type. The *collectorFunc* specifies how the collection process works. (Terminal operation.)
long count( )	Counts the number of elements in the stream and returns the result. (Terminal operation.)
Stream<T> filter(Predicate<? super T> *pred*)	Produces a stream that contains those elements from the invoking stream that satisfy the predicate specified by *pred*. (Intermediate operation.)
void forEach(Consumer<? super T> *action*)	For each element in the invoking stream, the code specified by *action* is executed. (Terminal operation.)
<R> Stream<R> map(Function<? super T, ? extends R> *mapFunc*)	Applies *mapFunc* to the elements from the invoking stream, yielding a new stream that contains those elements. (Intermediate operation.)
DoubleStream mapToDouble(ToDoubleFunction<? super T> *mapFunc*)	Applies *mapFunc* to the elements from the invoking stream, yielding a new **DoubleStream** that contains those elements. (Intermediate operation.)
IntStream mapToInt(ToIntFunction<? super T> *mapFunc*)	Applies *mapFunc* to the elements from the invoking stream, yielding a new **IntStream** that contains those elements. (Intermediate operation.)
LongStream mapToLong(ToLongFunction<? super T> *mapFunc*)	Applies *mapFunc* to the elements from the invoking stream, yielding a new **LongStream** that contains those elements. (Intermediate operation.)
Optional<T> max(Comparator<? super T> *comp*)	Using the ordering specified by *comp*, finds and returns the maximum element in the invoking stream. (Terminal operation.)
Optional<T> min(Comparator<? super T> *comp*)	Using the ordering specified by *comp*, finds and returns the minimum element in the invoking stream. (Terminal operation.)
T reduce(T *identityVal*, BinaryOperator<T> *accumulator*)	Returns a result based on the elements in the invoking stream. This is called a reduction operation. (Terminal operation.)
Stream<T> sorted( )	Produces a new stream that contains the elements of the invoking stream sorted in natural order. (Intermediate operation.)
Object[] toArray( )	Creates an array from the elements in the invoking stream. (Terminal operation.)

Table 29-2 A Sampling of Methods Declared by **Stream** *(continued)*

In both tables, notice that many of the methods are notated as being either **terminal** or **intermediate**. The difference between the two is very important.

A **terminal** operation consumes the stream. It is used to produce a result, such as finding the minimum value in the stream, or to execute some action, as is the case with the forEach( ) method. Once a stream has been consumed, it cannot be reused.

Intermediate operations produce another stream. Thus, intermediate operations can be used to create a pipeline that performs a sequence of actions.

One other point: intermediate operations do not take place immediately. Instead, the specified action is performed when a terminal operation is executed on the new stream created by an intermediate operation.

This mechanism is referred to as **lazy behavior**, and the intermediate operations are referred to as **lazy**. The use of lazy behavior enables the stream API to perform more efficiently.

Another key aspect of streams is that 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.

For example, sorting is a stateful operation because an element's order depends on the values of the other elements. Thus, the **sorted( ) method is stateful**.

However, filtering elements based on a stateless predicate is stateless because each element is handled individually. Thus, **filter( ) can (and should be) stateless**.

The difference between stateless and stateful operations is especially important when parallel processing of a stream is desired because a stateful operation may require more than one pass to complete.

Different operations on streams

Intermediate Operations

1. Map - The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.

   ```
   List number = Arrays.asList(2,3,4,5);
   List square = number.stream().map(x ->
   x*x).collect(Collectors.toList());
   ```

2. Filter - The filter method is used to select elements as per the Predicate passed as argument.

   ```
   List names =
   Arrays.asList("Reflection","Collection","Stream");
   ```

```
List result = names.stream().filter(s ->
s.startsWith("S")).collect(Collectors.toList());
```

3. Sorted - The sorted method is used to sort the stream.

```
List names =
Arrays.asList("Reflection","Collection","Stream");
List result =
names.stream().sorted().collect(Collectors.toList());
```

Terminal Operations

1. Collect - The collect method is used to return the result of the intermediate operations performed on the stream.

```
List number = Arrays.asList(2,3,4,5,3);
Set square = number.stream().map(x ->
x*x).collect(Collectors.toSet());
```

2. ForEach - The forEach method is used to iterate through every element of the stream.

```
List number = Arrays.asList(2,3,4,5);
number.stream().map(x->x*x).forEach(y ->
System.out.println(y));
```

3. Reduce - The reduce method is used to reduce the elements of a stream to a single value.

```
List number = Arrays.asList(2,3,4,5);
int even = number.stream().filter(x ->
x%2==0).reduce(0,(ans,i)-> ans+i);
```

How to Obtain a Stream

You can obtain a stream in a number of ways. Perhaps the most common is when a stream is obtained for a collection. Beginning with JDK 8, the Collection interface has been expanded to include two methods that obtain a stream from a collection. The first is stream( ), shown here:

```
default Stream<E> stream( )
```

Its **default** implementation returns a **sequential** stream. The second method is parallelStream( ), shown next:

```
default Stream<E> parallelStream( )
```

Its default implementation returns a parallel stream, if possible. (If a parallel stream cannot be obtained, a sequential stream may be returned instead.) Parallel streams support parallel execution of stream operations.

A stream can also be obtained from an array by use of the static stream( ) method, which was added to the Arrays class by JDK 8. One of its forms is shown here:

static <T> Stream<T> stream(T[ ] array)

This method returns a sequential stream to the elements in array. For example, given an array called addresses of type Address, the following obtains a stream to it:

```
Stream<Address> addrStrm = Arrays.stream(addresses);
```

Several overloads of the stream( ) method are also defined, such as those that handle arrays of the primitive types. They return a stream of type IntStream, DoubleStream, or LongStream.

A Simple Stream Example

The following program creates an ArrayList called myList that holds a collection of integers (which are automatically boxed into the Integer reference type). Next, it obtains a stream that uses myList as a source. It then demonstrates various stream operations.

```
package com.example.stream;

import java.util.ArrayList;
import java.util.Optional;
import java.util.stream.Stream;

//Demonstrate several stream operations.

public class StreamDemo {

	public static void main(String[] args) {
		// Create a list of Integer values.
		ArrayList<Integer> myList = new ArrayList<>();
		myList.add(7);
		myList.add(18);
		myList.add(10);
		myList.add(24);
		myList.add(17);
		myList.add(5);
		System.out.println("Original list: " + myList);

		// Obtain a Stream to the array list.
		Stream<Integer> myStream = myList.stream();

		// Obtain the minimum and maximum value by use of min(),
		// max(), isPresent(), and get().
		Optional<Integer> minVal = myStream.min(Integer::compare);
		if (minVal.isPresent())
			System.out.println("Minimum value: " + minVal.get());

		// Must obtain a new stream because previous call to min()
```

```
		// is a terminal operation that consumed the stream.
		myStream = myList.stream();
		Optional<Integer> maxVal = myStream.max(Integer::compare);
		if (maxVal.isPresent())
			System.out.println("Maximum value: " + maxVal.get());

		// Sort the stream by use of sorted().
		Stream<Integer> sortedStream = myList.stream().sorted();

		// Display the sorted stream by use of forEach().
		System.out.print("Sorted stream: ");
		sortedStream.forEach((n) -> System.out.print(n + " "));
		System.out.println();

		// Display only the odd values by use of filter().
		Stream<Integer> oddVals = myList.stream().sorted().filter((n) -> (n
% 2) == 1);
		System.out.print("Odd values: ");
		oddVals.forEach((n) -> System.out.print(n + " "));
		System.out.println();

		// Display only the odd values that are greater than 5. Notice that
		// two filter operations are pipelined.
		oddVals = myList.stream().filter((n) -> (n % 2) == 1).filter((n) ->
n > 5);
		System.out.print("Odd values greater than 5: ");
		oddVals.forEach((n) -> System.out.print(n + " "));
		System.out.println();
	}

}
```

Reduction Operations

Consider the min( ) and max( ) methods in the preceding example program. Both are terminal operations that return a result based on the elements in the stream. In the language of the stream API, they represent reduction operations because each reduces a stream to a single value—in this case, the minimum and maximum.

The stream API refers to these as special case reductions because they perform a specific function. In addition to min( ) and max( ), other special case reductions are also available, such as count( ), which counts the number of elements in a stream.

However, the stream API generalizes this concept by providing the reduce( ) method. By using reduce( ), you can return a value from a stream based on any arbitrary criteria. By definition, all reduction operations are terminal operations.

The following program demonstrates the versions of reduce( ) just described:

```
package com.example.stream;

import java.util.ArrayList;
import java.util.Optional;
```

```
//Demonstrate the reduce() method.

public class ReduceDemo {

	public static void main(String[] args) {
		// Create a list of Integer values.
		ArrayList<Integer> myList = new ArrayList<>();

		myList.add(7);
		myList.add(18);
		myList.add(10);
		myList.add(24);
		myList.add(17);
		myList.add(5);

		// Two ways to obtain the integer product of the elements
		// in myList by use of reduce().
		Optional<Integer> productObj = myList.stream().reduce((a, b) -> a *
b);
		if (productObj.isPresent())
			System.out.println("Product as Optional: " +
productObj.get());

		int product = myList.stream().reduce(1, (a, b) -> a * b);
		System.out.println("Product as int: " + product);

	}

}
```

Using Parallel Streams

Before exploring any more of the stream API, it will be helpful to discuss parallel streams. The parallel execution of code via multicore processors can result in a substantial increase in performance. Because of this, parallel programming has become an important part of the modern programmer's job.

However, parallel programming can be complex and error prone. One of the benefits that the stream library offers is the ability to easily—and reliably—parallel process certain operations.

Parallel processing of a stream is quite simple to request: just use a parallel stream.

As mentioned earlier, one way to obtain a parallel stream is to use the parallelStream( ) method defined by Collection. Another way to obtain a parallel stream is to call the parallel( ) method on a sequential stream. The parallel( ) method is defined by BaseStream, as shown here:

```
S parallel()
```

It returns a parallel stream based on the sequential stream that invokes it. (If it is called on a stream that is already parallel, then the invoking stream is returned.) Understand, of course, that even with a parallel stream, parallelism will be achieved only if the environment supports it.

As a general rule, any operation applied to a parallel stream must be stateless. It should also be non-interfering and associative. This ensures that the results obtained by executing operations on a parallel stream are the same as those obtained from executing the same operations on a sequential stream.

When using parallel streams, you might find the following version of reduce( ) especially helpful. It gives you a way to specify how partial results are combined:

```
<U> U reduce(U identityVal, BiFunction<U, ? super T, U> accumulator
BinaryOperator<U> combiner)
```

In this version, combiner defines the function that combines two values that have been produced by the accumulator function. Assuming the preceding program, the following statement computes the product of the elements in myList by use of a parallel stream:

```
int parallelProduct = myList.parallelStream().reduce(1, (a,b) ->
a*b, (a,b) -> a*b);
```

As you can see, in this example, both the accumulator and combiner perform the same function.

However, there are cases in which the actions of the accumulator must differ from those of the combiner.

For example, consider the following program. Here, myList contains a list of double values. It then uses the combiner version of reduce( ) to compute the product of the square roots of each element in the list.

```
package com.example.stream;

import java.util.ArrayList;

//Demonstrate the use of a combiner with reduce()

public class CombinerDemo {

	public static void main(String[] args) {
		// This is now a list of double values.
		ArrayList<Double> myList = new ArrayList<>();

		myList.add(7.0);
		myList.add(18.0);
		myList.add(10.0);
		myList.add(24.0);
		myList.add(17.0);
		myList.add(5.0);

		double productOfSqrRoots = myList.parallelStream().reduce(1.0, (a,
b) -> a * Math.sqrt(b), (a, b) -> a * b);

		System.out.println("Product of square roots: " + productOfSqrRoots);
	}

}
```

Mapping

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

For example, a stream that contains a database of name, telephone, and e-mail address information might map only the name and e-mail address portions to another stream.

As another example, you might want to apply some transformation to the elements in a stream. To do this, you could map the transformed elements to a new stream.

Because mapping operations are quite common, the stream API provides built-in support for them. The most general mapping method is map( ). It is shown here:

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

Here, R specifies the type of elements of the new stream; T is the type of elements of the invoking stream; and mapFunc is an instance of Function, which does the mapping.

The map function must be stateless and non-interfering. Since a new stream is returned, map( ) is an intermediate method.

The following is a simple example of map( ). It provides a variation on the previous example program. As before, the program computes the product of the square roots of the values in an ArrayList. In this version, however, the square roots of the elements are first mapped to a new stream. Then, reduce( ) is employed to compute the product.

```
package com.example.stream;

import java.util.ArrayList;
import java.util.stream.Stream;

//Map one stream to another.

public class MapDemo {

	public static void main(String[] args) {
		// A list of double values.
		ArrayList<Double> myList = new ArrayList<>();

		myList.add(7.0);
		myList.add(18.0);
		myList.add(10.0);
		myList.add(24.0);
		myList.add(17.0);
		myList.add(5.0);

		// Map the square root of the elements in myList to a new stream.
		Stream<Double> sqrtRootStrm = myList.stream().map((a) ->
Math.sqrt(a));

		// Find the product of the square roots.
		double productOfSqrRoots = sqrtRootStrm.reduce(1.0, (a, b) -> a *
b);

		System.out.println("Product of square roots is " +
productOfSqrRoots);
	}
```

```
}
```

The output is the same as before. The difference between this version and the previous is simply that the transformation (i.e., the computation of the square roots) occurs during mapping, rather than during the reduction. Because of this, it is possible to use the two-parameter form of reduce( ) to compute the product because it is no longer necessary to provide a separate combiner function.

Here is an example that uses map( ) to create a new stream that contains only selected fields from the original stream. In this case, the original stream contains objects of type NamePhoneEmail, which contains names, phone numbers, and e-mail addresses. The program then maps only the names and phone numbers to a new stream of NamePhoneobjects. The e-mail addresses are discarded.

```
package com.example.stream;

import java.util.ArrayList;
import java.util.stream.Stream;

//Use map() to create a new stream that contains only
//selected aspects of the original stream.

class NamePhoneEmail {
	String name;
	String phonenum;
	String email;

	NamePhoneEmail(String n, String p, String e) {
		name = n;
		phonenum = p;
		email = e;
	}
}

class NamePhone {
	String name;
	String phonenum;

	NamePhone(String n, String p) {
		name = n;
		phonenum = p;
	}
}

public class SelectMapDemo {

	public static void main(String[] args) {
		// A list of names, phone numbers, and e-mail addresses.
		ArrayList<NamePhoneEmail> myList = new ArrayList<>();

		myList.add(new NamePhoneEmail("Larry", "555-5555",
"Larry@HerbSchildt.com"));
```

```
		myList.add(new NamePhoneEmail("James", "555-4444",
"James@HerbSchildt.com"));
		myList.add(new NamePhoneEmail("Mary", "555-3333",
"Mary@HerbSchildt.com"));

		System.out.println("Original values in myList: ");
		myList.stream().forEach((a) -> {
			System.out.println(a.name + " " + a.phonenum + " " + a.email);
		});
		System.out.println();

		// Map just the names and phone numbers to a new stream.
		Stream<NamePhone> nameAndPhone = myList.stream().map((a) -> new
NamePhone(a.name, a.phonenum));

		System.out.println("List of names and phone numbers: ");
		nameAndPhone.forEach((a) -> {
			System.out.println(a.name + " " + a.phonenum);
		});
	}

}
```

Here is an example that uses a primitive stream. It first creates an ArrayList of Double values. It then uses stream( ) followed by mapToInt( ) to create an IntStream that contains the ceiling of each value.

```
package com.example.stream;

import java.util.ArrayList;
import java.util.stream.IntStream;

//Map a Stream to an IntStream.

public class PrimitiveDemo {

	public static void main(String[] args) {
		// A list of double values.
		ArrayList<Double> myList = new ArrayList<>();

		myList.add(1.1);
		myList.add(3.6);
		myList.add(9.2);
		myList.add(4.7);
		myList.add(12.1);
		myList.add(5.0);

		System.out.print("Original values in myList: ");
		myList.stream().forEach((a) -> {
			System.out.print(a + " ");
		});
		System.out.println();

		// Map the ceiling of the elements in myList to an IntStream.
		IntStream cStrm = myList.stream().mapToInt((a) -> (int)
Math.ceil(a));
```

```
		System.out.print("The ceilings of the values in myList: ");
		cStrm.forEach((a) -> {
			System.out.print(a + " ");
		});
	}

}
```

Collecting

As the preceding examples have shown, it is possible (indeed, common) to obtain a stream from a collection. Sometimes it is desirable to obtain the opposite: to obtain a collection from a stream. To perform such an action, the stream API provides the collect( ) method. It has two forms. The one we will use first is shown here:

```
<R, A> R collect(Collector<? super T, A, R> collectorFunc)
```

Here, R specifies the type of the result, and T specifies the element type of the invoking stream. The internal accumulated type is specified by A. The collectorFunc specifies how the collection process works. The collect( ) method is a terminal operation.

The following program puts the preceding discussion into action. It reworks the example in the previous section so that it collects the names and phone numbers into a List and a Set.

```
package com.example.stream;

import java.util.ArrayList;
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;
import java.util.stream.Stream;

//Use collect() to create a List and a Set from a stream.

class NamePhoneEmail {
	String name;
	String phonenum;
	String email;

	NamePhoneEmail(String n, String p, String e) {
		name = n;
		phonenum = p;
		email = e;
	}
}

class NamePhone {
	String name;
	String phonenum;

	NamePhone(String n, String p) {
		name = n;
		phonenum = p;
	}
}
```

```
public class CollectDemo {

	public static void main(String[] args) {
		// A list of names, phone numbers, and e-mail addresses.
		ArrayList<NamePhoneEmail> myList = new ArrayList<>();

		myList.add(new NamePhoneEmail("Larry", "555-5555",
"Larry@HerbSchildt.com"));
		myList.add(new NamePhoneEmail("James", "555-4444",
"James@HerbSchildt.com"));
		myList.add(new NamePhoneEmail("Mary", "555-3333",
"Mary@HerbSchildt.com"));

		// Map just the names and phone numbers to a new stream.
		Stream<NamePhone> nameAndPhone = myList.stream().map((a) -> new
NamePhone(a.name, a.phonenum));

		// Use collect to create a List of the names and phone numbers.
		List<NamePhone> npList = nameAndPhone.collect(Collectors.toList());

		System.out.println("Names and phone numbers in a List:");
		for (NamePhone e : npList)
			System.out.println(e.name + ": " + e.phonenum);

		// Obtain another mapping of the names and phone numbers.
		nameAndPhone = myList.stream().map((a) -> new NamePhone(a.name,
a.phonenum));

		// Now, create a Set by use of collect().
		Set<NamePhone> npSet = nameAndPhone.collect(Collectors.toSet());

		System.out.println("\nNames and phone numbers in a Set:");
		for (NamePhone e : npSet)
			System.out.println(e.name + ": " + e.phonenum);
	}

}
```