Example 1:

```
List<String> myList =
   Arrays.asList("a1", "a2", "b1", "c2", "c1");


myList
   .stream()
   .filter(s -> s.startsWith("c"))
   .map(String::toUpperCase)
   .sorted()
   .forEach(System.out::println);  ///terminal stream
```

Stream operations are either intermediate or terminal. Intermediate operations return a stream so we can chain multiple intermediate operations without using semicolons. Terminal operations are either void or return a non-stream result. In the above example `filter`, `map` and `sorted` are intermediate operations whereas `forEach` is a terminal operation. For a full list of all available stream operations see the Stream Javadoc. Such a chain of stream operations as seen in the example above is also known as *operation pipeline*.

Example 2:

```
Arrays.asList("a1", "a2", "a3")
   .stream()
   .findFirst()
   .ifPresent(System.out::println);  // a1
```

Example 3:

```
Stream.of("a1", "a2", "a3")
   .findFirst()
   .ifPresent(System.out::println);  // a1
```

Note : use Stream.of() to create a stream from a bunch of object references.

Example 4:

special kinds of streams for working with the primitive data types int, long and double. As you might have guessed it's IntStream, LongStream and DoubleStream.

IntStreams can replace the regular for-loop utilizing IntStream.range():

```
IntStream.range(1, 4)
    .forEach(System.out::println);


// 1
// 2
// 3
```

All those primitive streams work just like regular object streams with the following differences: Primitive streams use specialized lambda expressions, e.g. IntFunction instead of Function or IntPredicate instead of Predicate. And primitive streams support the additional terminal aggregate operations sum() and average():

```
Arrays.stream(new int[] {1, 2, 3})
    .map(n -> 2 * n + 1)
    .average().ifPresent(System.out::println);  // 5.0
```

Example 5:

Primitive streams can be transformed to object streams via mapToObj():

```
IntStream.range(1, 4)
    .mapToObj(i -> "a" + i)
    .forEach(System.out::println);


// a1
// a2
// a3
```

the stream of doubles is first mapped to an int stream and than mapped to an object stream of strings:

```
Stream.of(10.0, 20.0, 30.0,40.0)
    .mapToInt(Double::intValue)
    .mapToObj(i -> "str" + i)
```

```
    .forEach(System.out::println);


// str1

// str2

// str3
```

Example 6

An important characteristic of intermediate operations is laziness. In following example terminal operation is missing:

```
Stream.of("x2", "y2", "z1", "Hello", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return true;
    });
```

When executing this code snippet, nothing is printed to the console. That is because intermediate operations will only be executed when a terminal operation is present.

Add terminal operation forEach:

```
Stream.of("Ashu", "Deepa", "Rajan", "Revati", "Anil")
    .filter(s -> {
        System.out.println("filter: " + s);
        return true;
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

Check the order of the execution. It would be to execute the operations horizontally one after another on all elements of the stream. But instead each element moves along the chain vertically. The first string "Ashu" passes filter then forEach, only then the second string "Deepa" is processed.

This behavior can reduce the actual number of operations performed on each element,

```
Stream.of("Ashu", "Deepa", "Rajan", "Revati", "Anil")
    .map(s -> {
        System.out.println("map: " + s);
```

```
        return s.toUpperCase();
    })
    .anyMatch(s -> {
        System.out.println("anyMatch: " + s);
        return s.startsWith("A");
    });

// map:     Ashu
// anyMatch: Ashu
// map:     Anil
// anyMatch: Anil
```

The operation anyMatch will stop when it will find first match. AllMatch will find all matches. And returns true as soon as the predicate applies to the given input element.

In above example This is true for the first element passed "AShu". Due to the vertical execution of the stream chain, map has only to be executed Once in this case. So instead of mapping all elements of the stream, map will be called as few as possible.

Example 7

To reduce the number of operations write proper sequence of functions

```
Stream.of("Ashu", "Deepa", "Rajan", "Revati", "Anil")
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("A");
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

Instead following will be faster

```
Stream.of("Ashu", "Deepa", "Rajan", "Revati", "Anil")
```

```
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("A");
    })
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

Question 1. Can we improve the performance.

Note sorted work horizontally. Filter and map works vertically for each element.

```
Stream.of("Ashu", "Deepa", "Rajan", "Revati", "Anil")
    .sorted((s1, s2) -> {
        System.out.printf("sort: %s; %s\n", s1, s2);
        return s1.compareTo(s2);
    })
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("A");
    })
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

Example 8 : check sorted will not called if size of collection is 1

```
Stream.of("Ashu", "Deepa", "Rajan", "Revati", "Anil")
    .filter(s -> {
        System.out.println("filter: " + s);
```

```
    return s.startsWith("D");
  })
  .sorted((s1, s2) -> {
    System.out.printf("sort: %s; %s\n", s1, s2);
    return s1.compareTo(s2);
  })
  .map(s -> {
    System.out.println("map: " + s);
    return s.toUpperCase();
  })
  .forEach(s -> System.out.println("forEach: " + s));
```

Example 9:

Java 8 streams cannot be reused. As soon as you call any terminal operation the stream is closed:

```
Stream<String> stream =
Stream.of("Ashu", "Deepa", "Rajan", "Revati", "Anil")
      .filter(s -> s.startsWith("A"));

stream.anyMatch(s -> true);    // ok
stream.noneMatch(s -> true);   // exception
```

Note Calling noneMatch after anyMatch on the same stream results in the following exception:

java.lang.IllegalStateException: stream has already been operated upon or closed

Example 9 : use stream in the following way to avoid exception

```
Supplier<Stream<String>> streamSupplier =
  () -> Stream.of("Ashu", "Deepa", "Rajan", "Revati", "Anil")
        .filter(s -> s.startsWith("A"));

streamSupplier.get().anyMatch(s -> s.startsWith(“A”));   // ok
streamSupplier.get().noneMatch(s ->s.startsWith(“D”));  // ok
```

Each call to get() constructs a new stream

Streams support plenty of different operations. We've already learned about the most important operations like filter or map. I leave it up to you to discover all other available operations (see Stream Javadoc). Instead let's dive deeper into the more complex operations collect, flatMap and reduce.

Most code samples from this section use the following list of persons for demonstration purposes:

```
class Person {
   String name;
   int age;

   Person(String name, int age) {
      this.name = name;
      this.age = age;
   }

   @Override
   public String toString() {
      return name;
   }
}

List<Person> persons =
   Arrays.asList(
      new Person("Meena", 18),
      new Person("Prasad", 23),
      new Person("Deepa", 23),
      new Person("David", 12));
```

Collect#

Collect is an extremely useful terminal operation to transform the elements of the stream into a different kind of result, e.g. a List, Set or Map. Collect accepts a Collector which consists of four different operations: a supplier, an accumulator, a combiner and a finisher. This sounds super complicated at first, but the good part is Java 8 supports various built-in collectors via the Collectors class. So for the most common operations you don't have to implement a collector yourself.

```
List<Person> filtered =
    persons
        .stream()
        .filter(p -> p.name.startsWith("P"))
        .collect(Collectors.toList());

System.out.println(filtered);    // [Prasad, Deepa]
```

To convert to set instead of list - just use Collectors.toSet().

The next example groups all persons by age:

```
Map<Integer, List<Person>> personsByAge = persons
    .stream()
    .collect(Collectors.groupingBy(p -> p.age));

personsByAge
    .forEach((age, p) -> System.out.format("age %s: %s\n", age, p));

// age 18: [Meena]
// age 23: [Prasad, Deepa]
// age 12: [David]
```

Collectors are extremely versatile. You can also create aggregations on the elements of the stream, e.g. determining the average age of all persons:

```
Double averageAge = persons
    .stream()
    .collect(Collectors.averagingInt(p -> p.age));

System.out.println(averageAge);     // 19.0
```

To get statistical values, the summarizing collectors return a special built-in summary statistics object. So we can simply determine min, max and arithmetic average age of the persons as well as the sum and count.

```
IntSummaryStatistics ageSummary =
    persons
        .stream()
```

```
    .collect(Collectors.summarizingInt(p -> p.age));


System.out.println(ageSummary);
// IntSummaryStatistics{count=4, sum=76, min=12, average=19.000000, max=23}
```

The next example joins all persons into a single string:

```
String phrase = persons
   .stream()
   .filter(p -> p.age >= 18)
   .map(p -> p.name)
   .collect(Collectors.joining(", ", "It selected ", " are of legal age.")); /// joining(delimiter,prefix,suffix)


System.out.println(phrase);
// It selected Meena, Prasad , Deepa are of legal age.
```

The join collector accepts a delimiter as well as an optional prefix and suffix.

In order to transform the stream elements into a map, we have to specify how both the keys and the values should be mapped. Keep in mind that the mapped keys must be unique, otherwise an IllegalStateException is thrown. You can optionally pass a merge function as an additional parameter to bypass the exception:

```
Map<Integer, String> map = persons
   .stream()
   .collect(Collectors.toMap(
      p -> p.age,
      p -> p.name,
      (name1, name2) -> name1 + ";" + name2));


System.out.println(map);
// {18=Meena, 23=Prasad;Deepa, 12=David}
```

Now that we know some of the most powerful built-in collectors, let's try to build our own special collector. We want to transform all persons of the stream into a single string consisting of all names in upper letters separated by the | pipe character. In order to achieve this we create a new collector via Collector.of(). We have to pass the four ingredients of a collector: a supplier, an accumulator, a combiner and a finisher.

```
Collector<Person, StringJoiner, String> personNameCollector =
```

```
    Collector.of(
        () -> new StringJoiner(" | "),         // supplier
        (j, p) -> j.add(p.name.toUpperCase()),  // accumulator
        (j1, j2) -> j1.merge(j2),               // combiner
        StringJoiner::toString);                // finisher

String names = persons
    .stream()
    .collect(personNameCollector);

System.out.println(names);  // MEENA | PRASAD | DEEPA | DAVID
```

Since strings in Java are immutable, we need a helper class like StringJoiner to let the collector construct our string. The supplier initially constructs such a StringJoiner with the appropriate delimiter. The accumulator is used to add each persons upper-cased name to the StringJoiner. The combiner knows how to merge two StringJoiners into one. In the last step the finisher constructs the desired String from the StringJoiner.

FlatMap

We've already learned how to transform the objects of a stream into another type of objects by utilizing the map operation. Map is kind of limited because every object can only be mapped to exactly one other object. But what if we want to transform one object into multiple others or none at all? This is where flatMap is used.

FlatMap transforms each element of the stream into a stream of other objects. So each object will be transformed into zero, one or multiple other objects backed by streams. The contents of those streams will then be placed into the returned stream of the flatMap operation.

Before we see flatMap in action we need an appropriate type hierarchy:

```
class Foo {
    String name;
    List<Bar> bars = new ArrayList<>();

    Foo(String name) {
        this.name = name;
    }
}
```

```
class Bar {
  String name;

  Bar(String name) {
    this.name = name;
  }
}
```

Create streams to instantiate a couple of objects:

```
List<Foo> foos = new ArrayList<>();

// create foos
IntStream
  .range(1, 4)
  .forEach(i -> foos.add(new Foo("Foo" + i)));

// create bars
foos.forEach(f ->
  IntStream
    .range(1, 4)
    .forEach(i -> f.bars.add(new Bar("Bar" + i + " <- " + f.name))));
```

Now we have a list of three foos each consisting of three bars.

FlatMap accepts a function which has to return a stream of objects. So in order to resolve the bar objects of each foo, we just pass the appropriate function:

```
foos.stream()
  .flatMap(f -> f.bars.stream())
  .forEach(b -> System.out.println(b.name));

// Bar1 <- Foo1
// Bar2 <- Foo1
// Bar3 <- Foo1
// Bar1 <- Foo2
```

```
// Bar2 <- Foo2

// Bar3 <- Foo2

// Bar1 <- Foo3

// Bar2 <- Foo3

// Bar3 <- Foo3
```

As you can see, we've successfully transformed the stream of three foo objects into a stream of nine bar objects.

Finally, the above code example can be simplified into a single pipeline of stream operations:

```
IntStream.range(1, 4)
   .mapToObj(i -> new Foo("Foo" + i))
   .peek(f -> IntStream.range(1, 4)
      .mapToObj(i -> new Bar("Bar" + i + " <- " f.name))
      .forEach(f.bars::add))
   .flatMap(f -> f.bars.stream())
   .forEach(b -> System.out.println(b.name));
```

FlatMap is also available for the Optional class introduced in Java 8. Optionals flatMap operation returns an optional object of another type. So it can be utilized to prevent nasty null checks.

Think of a highly hierarchical structure like this:

```
class Outer {
   Nested nested;
}

class Nested {
   Inner inner;
}

class Inner {
   String foo;
}
```

In order to resolve the inner string foo of an outer instance you have to add multiple null checks to prevent possible NullPointerExceptions:

```
Outer outer = new Outer();
```

```
if (outer != null && outer.nested != null && outer.nested.inner != null) {
   System.out.println(outer.nested.inner.foo);
}
```

The same behavior can be obtained by utilizing optionals flatMap operation:

```
Optional.of(new Outer())
   .flatMap(o -> Optional.ofNullable(o.nested))
   .flatMap(n -> Optional.ofNullable(n.inner))
   .flatMap(i -> Optional.ofNullable(i.foo))
   .ifPresent(System.out::println);
```

Each call to flatMap returns an Optional wrapping the desired object if present or null if absent.

Reduce#

The reduction operation combines all elements of the stream into a single result. Java 8 supports three different kind of reduce methods. The first one reduces a stream of elements to exactly one element of the stream. Let's see how we can use this method to determine the oldest person:

```
persons
   .stream()
   .reduce((p1, p2) -> p1.age > p2.age ? p1 : p2)
   .ifPresent(System.out::println);    // Deepa
```

The reduce method accepts a BinaryOperator accumulator function. That's actually a BiFunction where both operands share the same type, in that case Person. BiFunctions are like Function but accept two arguments. The example function compares both persons ages in order to return the person with the maximum age.

The second reduce method accepts both an identity value and a BinaryOperator accumulator. This method can be utilized to construct a new Person with the aggregated names and ages from all other persons in the stream:

```
Person result =
   persons
      .stream()
      .reduce(new Person("", 0), (p1, p2) -> {
         p1.age += p2.age;
         p1.name += p2.name;
         return p1;
      });
```

```
System.out.format("name=%s; age=%s", result.name, result.age);

// name=MeenaPrasadDeepaDavid; age=76
```

The third reduce method accepts three parameters: an identity value, a BiFunction accumulator and a combiner function of type BinaryOperator. Since the identity values type is not restricted to the Person type, we can utilize this reduction to determine the sum of ages from all persons:

```
Integer ageSum = persons
   .stream()
   .reduce(0, (sum, p) -> sum += p.age, (sum1, sum2) -> sum1 + sum2);


System.out.println(ageSum);  // 76
```

As you can see the result is 76, but what's happening exactly under the hood? Let's extend the above code by some debug output:

```
Integer ageSum = persons
   .stream()
   .reduce(0,
      (sum, p) -> {
         System.out.format("accumulator: sum=%s; person=%s\n", sum, p);
         return sum += p.age;
      },
      (sum1, sum2) -> {
         System.out.format("combiner: sum1=%s; sum2=%s\n", sum1, sum2);
         return sum1 + sum2;
      });


// accumulator: sum=0; person=Meena
// accumulator: sum=18; person=Prasad
// accumulator: sum=41; person=Deepa
// accumulator: sum=64; person=David
```

As you can see the accumulator function does all the work. It first get called with the initial identity value 0 and the first person Meena. In the next three steps sum continually increases by the age of the last steps person up to a total age of 76.

Wait wat? The combiner never gets called? Executing the same stream in parallel will lift the secret:

```
Integer ageSum = persons
    .parallelStream()
    .reduce(0,
        (sum, p) -> {
            System.out.format("accumulator: sum=%s; person=%s\n", sum, p);
            return sum += p.age;
        },
        (sum1, sum2) -> {
            System.out.format("combiner: sum1=%s; sum2=%s\n", sum1, sum2);
            return sum1 + sum2;
        });


// accumulator: sum=0; person=Deepa
// accumulator: sum=0; person=David
// accumulator: sum=0; person=Meena
// accumulator: sum=0; person=Prasad
// combiner: sum1=18; sum2=23
// combiner: sum1=23; sum2=12
// combiner: sum1=41; sum2=35
```

Executing this stream in parallel results in an entirely different execution behavior. Now the combiner is actually called. Since the accumulator is called in parallel, the combiner is needed to sum up the separate accumulated values.

Let's dive deeper into parallel streams in the next chapter.

Parallel Streams#

Streams can be executed in parallel to increase runtime performance on large amount of input elements. Parallel streams use a common ForkJoinPool available via the static ForkJoinPool.commonPool() method. The size of the underlying thread-pool uses up to five threads - depending on the amount of available physical CPU cores:

```
ForkJoinPool commonPool = ForkJoinPool.commonPool();

System.out.println(commonPool.getParallelism());    // 3
```

On my machine the common pool is initialized with a parallelism of 3 per default. This value can be decreased or increased by setting the following JVM parameter:

```
-Djava.util.concurrent.ForkJoinPool.common.parallelism=5
```

Collections support the method parallelStream() to create a parallel stream of elements. Alternatively you can call the intermediate method parallel() on a given stream to convert a sequential stream to a parallel counterpart.

In order to understate the parallel execution behavior of a parallel stream the next example prints information about the current thread to sout:

```
Arrays.asList("a1", "a2", "b1", "c2", "c1")
    .parallelStream()
    .filter(s -> {
        System.out.format("filter: %s [%s]\n",
            s, Thread.currentThread().getName());
        return true;
    })
    .map(s -> {
        System.out.format("map: %s [%s]\n",
            s, Thread.currentThread().getName());
        return s.toUpperCase();
    })
    .forEach(s -> System.out.format("forEach: %s [%s]\n",
        s, Thread.currentThread().getName()));
```

By investigating the debug output we should get a better understanding which threads are actually used to execute the stream operations:

```
filter:  b1 [main]
filter:  a2 [ForkJoinPool.commonPool-worker-1]
map:     a2 [ForkJoinPool.commonPool-worker-1]
filter:  c2 [ForkJoinPool.commonPool-worker-3]
map:     c2 [ForkJoinPool.commonPool-worker-3]
filter:  c1 [ForkJoinPool.commonPool-worker-2]
map:     c1 [ForkJoinPool.commonPool-worker-2]
forEach: C2 [ForkJoinPool.commonPool-worker-3]
forEach: A2 [ForkJoinPool.commonPool-worker-1]
map:     b1 [main]
forEach: B1 [main]
```

```
filter:  a1 [ForkJoinPool.commonPool-worker-3]
map:     a1 [ForkJoinPool.commonPool-worker-3]
forEach: A1 [ForkJoinPool.commonPool-worker-3]
forEach: C1 [ForkJoinPool.commonPool-worker-2]
```

As you can see the parallel stream utilizes all available threads from the common ForkJoinPool for executing the stream operations. The output may differ in consecutive runs because the behavior which particular thread is actually used is non-deterministic.

Let's extend the example by an additional stream operation, sort:

```
Arrays.asList("a1", "a2", "b1", "c2", "c1")
    .parallelStream()
    .filter(s -> {
        System.out.format("filter: %s [%s]\n",
            s, Thread.currentThread().getName());
        return true;
    })
    .map(s -> {
        System.out.format("map: %s [%s]\n",
            s, Thread.currentThread().getName());
        return s.toUpperCase();
    })
    .sorted((s1, s2) -> {
        System.out.format("sort: %s <> %s [%s]\n",
            s1, s2, Thread.currentThread().getName());
        return s1.compareTo(s2);
    })
    .forEach(s -> System.out.format("forEach: %s [%s]\n",
        s, Thread.currentThread().getName()));
```

The result may look strange at first:

```
filter:  c2 [ForkJoinPool.commonPool-worker-3]
filter:  c1 [ForkJoinPool.commonPool-worker-2]
map:     c1 [ForkJoinPool.commonPool-worker-2]
filter:  a2 [ForkJoinPool.commonPool-worker-1]
```

```
map:    a2 [ForkJoinPool.commonPool-worker-1]
filter:  b1 [main]
map:    b1 [main]
filter:  a1 [ForkJoinPool.commonPool-worker-2]
map:    a1 [ForkJoinPool.commonPool-worker-2]
map:    c2 [ForkJoinPool.commonPool-worker-3]
sort:   A2 <> A1 [main]
sort:   B1 <> A2 [main]
sort:   C2 <> B1 [main]
sort:   C1 <> C2 [main]
sort:   C1 <> B1 [main]
sort:   C1 <> C2 [main]
forEach: A1 [ForkJoinPool.commonPool-worker-1]
forEach: C2 [ForkJoinPool.commonPool-worker-3]
forEach: B1 [main]
forEach: A2 [ForkJoinPool.commonPool-worker-2]
forEach: C1 [ForkJoinPool.commonPool-worker-1]
```

It seems that sort is executed sequentially on the main thread only. Actually, sort on a parallel stream uses the new Java 8 method Arrays.parallelSort() under the hood. As stated in Javadoc this method decides on the length of the array if sorting will be performed sequentially or in parallel:

If the length of the specified array is less than the minimum granularity, then it is sorted using the appropriate Arrays.sort method.

Coming back to the reduce example from the last section. We already found out that the combiner function is only called in parallel but not in sequential streams. Let's see which threads are actually involved:

```
List<Person> persons = Arrays.asList(
    new Person("Meena", 18),
    new Person("Prasad", 23),
    new Person("Deepa", 23),
    new Person("David", 12));


persons
    .parallelStream()
```

```
    .reduce(0,
        (sum, p) -> {
            System.out.format("accumulator: sum=%s; person=%s [%s]\n",
                sum, p, Thread.currentThread().getName());
            return sum += p.age;
        },
        (sum1, sum2) -> {
            System.out.format("combiner: sum1=%s; sum2=%s [%s]\n",
                sum1, sum2, Thread.currentThread().getName());
            return sum1 + sum2;
        });
```

The console output reveals that both the accumulator and the combiner functions are executed in parallel on all available threads:

```
accumulator: sum=0; person=Deepa; [main]
accumulator: sum=0; person=Meena;    [ForkJoinPool.commonPool-worker-3]
accumulator: sum=0; person=David;  [ForkJoinPool.commonPool-worker-2]
accumulator: sum=0; person=Prasad;  [ForkJoinPool.commonPool-worker-1]
combiner:    sum1=18; sum2=23;     [ForkJoinPool.commonPool-worker-1]
combiner:    sum1=23; sum2=12;     [ForkJoinPool.commonPool-worker-2]
combiner:    sum1=41; sum2=35;     [ForkJoinPool.commonPool-worker-2]
```

In summary, it can be stated that parallel streams can bring be a nice performance boost to streams with a large amount of input elements. But keep in mind that some parallel stream operations like reduce and collect need additional computations (combine operations) which isn't needed when executed sequentially.

Furthermore we've learned that all parallel stream operations share the same JVM-wide common ForkJoinPool. So you probably want to avoid implementing slow blocking stream operations since that could potentially slow down other parts of your application which rely heavily on parallel streams.