Java 8 Streams - Mutable Reduction

[Updated: Apr 29, 2017, Created: Nov 26, 2016]

Mutable reductions collect the desired results into a mutable container object such as a java.util.Collection or an array.

Mutable reduction in Java stream API are implemented as **collect()** methods.

reduce() methods which we discussed in the last tutorial are immutable reduction, as they reduce the result into a single valued immutable variable.

reduce() vs collect()

Both terminal operations, reduce() and collect(), are categorized as reduction operations.

In **collect()** operations, elements are incorporated by updating the state of a mutable container object.

In **reduce()** operations result is updated by replacing the previous result.

Stream#collect() methods

```
(1) <R> R collect(Supplier<R> supplier,
                  BiConsumer<R,? super T> accumulator,
                  BiConsumer<R,R> combiner)
```

This method reduces stream element of type T to a mutable result container of type R.

supplier: this function creates a a new result container. In sequential execution it's called only once, whereas, for parallel execution, it may be called multiple times to get a new instance for different parallel threads.

accumulator: an associative function to incorporate the current element to the result object (the result object is created in supplier function)

combiner: in parallel execution this function combines the results received from different threads. This must be associative function.

Following example uses StringBuilder as the mutable container to concatenate the strings:

```
List<String> list = Arrays.asList("Mike", "Nicki", "John");
String s = list.stream().collect(StringBuilder::new,
                          (sb, s1) -> sb.append(" ").append(s1),
                          (sb1, sb2) -> sb1.append(sb2.toString())).toString();
System.out.println(s);
```

Output

```
 Mike Nicki John
```

Here's the reduce() version of same concatenation operation:

```
List<String> list = Arrays.asList("Mike", "Nicki", "John");
String s = list.stream().reduce("", (s1, s2) -> s1 + " " + s2);
System.out.println(s);
```

Output

```
 Mike Nicki John
```

The reduce() operation in an example like above will be less efficient in performance than collect() operation, specially for large number of stream elements. That's because collect() creates only one instance of the container object rather than creating a new one for each iteration.

Other variants:

Class	Method

IntStream	`<R> R collect(Supplier<R> supplier,`
	`ObjIntConsumer<R> accumulator,`
	`BiConsumer<R,R> combiner)`

LongStream	`<R> R collect(Supplier<R> supplier,`
	`ObjLongConsumer<R> accumulator,`
	`BiConsumer<R,R> combiner)`

DoubleStream	`<R> R collect(Supplier<R> supplier,`
	`ObjDoubleConsumer<R> accumulator,`
	`BiConsumer<R,R> combiner)`

Following is an IntStream example to collect multiples of 10 into ArrayList:

```
public class MutableReductionExample2 {
    public static void main (String[] args) {
       IntStream stream = IntStream.range(1, 100);
       List<Integer> list = stream.parallel()
                                  .filter(i -> i % 10 == 0)
                                  .collect(ArrayList::new, ArrayList::add
                                                    , ArrayList::addAll);
       System.out.println(list);
    }
}
```

Output

```
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

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

Collector interface: encapsulates the same arguments used by the method `collect(Supplier, BiConsumer, BiConsumer)` which we discussed above, plus an optional `finisher()` method for final type conversion form A to R and a method `characteristics()` for indicating the collector properties.

```
package java.util.stream;

 //imports
```

```
public interface Collector<T, A, R> {

    Supplier<A> supplier();
    BiConsumer<A, T> accumulator();
    BinaryOperator<A> combiner();
    Function<A, R> finisher();
    Set<Characteristics> characteristics();

    //some static methods

    enum Characteristics {
      CONCURRENT,
      UNORDERED,
      IDENTITY_FINISH
    }
}
```

Note that 'A' type of this interface is actually 'R' type of the method `collect(Supplier, BiConsumer, BiConsumer)` which we discussed in (1). So to be clear:

```
T => the underlying stream type,
A => the accumulator mutable container returned from the supplier,
R => the final type which returns from the finisher() call so does from Stream#collect() call.
```

Using this interface also allows to reuse various collect operations, which includes factory creation of the collectors by the class Collectors .

Implementing Collector

To understand Collector interface clearly, let's implement a collector for our first example of StringBuilder above where we did this:

```
List<String> list = Arrays.asList("Mike", "Nicki", "John");
String s = list.stream().collect(StringBuilder::new,
                            (sb, s1) -> sb.append(" ").append(s1),
                            (sb1, sb2) -> sb1.append(sb2.toString())).toString();
```

The equivalent Collector implementation:

```
public class CollectorExample {
    public static void main (String[] args) {
        String s = Stream.of("Mike", "Nicki", "John").collect(new
                            MyCollector());
        System.out.println(s);
    }

    private static class MyCollector implements
                        Collector<String, StringBuilder, String> {

        @Override
        public Supplier<StringBuilder> supplier () {
            return StringBuilder::new;
        }

        @Override
        public BiConsumer<StringBuilder, String> accumulator () {
            return (sb, s) -> sb.append(" ").append(s);
        }

        @Override
        public BinaryOperator<StringBuilder> combiner () {
            return StringBuilder::append;
        }

        @Override
        public Function<StringBuilder, String> finisher () {
            return stringBuilder -> stringBuilder.toString();
        }

        @Override
        public Set<Characteristics> characteristics () {
            return Collections.emptySet();
        }
    }
}
```

Output

```
Mike Nicki John
```

What is Collector#characteristics() method:

The set of characteristics specified by the collector implementation is to provide followings one or more optimization hints to the processing stream pipeline.

1. **Characteristics.CONCURRENT**

   If a collector has this characteristic and we have parallel() pipeline, then followings points are important to understand:

   1. The supplier function is called only once and a single shared instance for 'A' (the mutable result container) will be used in the **accumulator()** function by the multiple threads. In our last example that would be `StringBuilder` instance.
   2. We have to provide a **thread-safe** instance for 'A' because multiple threads will be updating it in a parallel stream. In our last example, we probably have to replace `StringBuilder` with `StringBuffer`.
   3. The **combiner function** will not be used to combine multiple result containers, because we don't have multiple of them, a single shared instance of result container has been updated in **accumulator()** directly by multiple threads.
   4. This hint may be ignored. The current stream implementation has only one such scenario:
      In the case of ordered stream (having Spliterator#ORDERED characteristic) if multiple threads are updating the shared accumulator container, the order in which results are updated, will be non-deterministic. To avoid that the pipeline will ignore CONCURRENT characteristic for the ordered source unless it has **UNORDERED characteristic** (next discussion) as well.
      See also: Encounter order tutorial

   Let's rewrite our last example and use CONCURRENT characteristic. We are going replace StringBuilder with StringBuffer and going to specify CONCURRENT. We will do that for unordered and then with ordered stream source one by one:

```
public class CollectorExample2 {
  public static void main (String[] args) {
      String s = Stream.of("Mike", "Nicki", "John")
                       .parallel()
                       .unordered()
                       .collect(new MyCollector());
      System.out.println(s);
  }

  private static class MyCollector implements
                      Collector<String, StringBuffer, String> {

      @Override
      public Supplier<StringBuffer> supplier () {
```

Modern J
Lambdas
$44.05

OCP: Or
Professio
$37.53

Java 1
Java 1
Java 1
Java 1
Java 9
Java 9
Java 9
Java 9
Recent
Spring
Local
Spring
Listen
Spring
Defaul
Spring
reques
Spring
config
Spring
Config
Spring
failure
Spring
TypeS
Suppo
JavaS
Promis
TypeS
JPA Cr
Object
JPA Cr
JPA Cr
JPA Cr
Criteri
JPA Cr
Criteri
Java 1
files w
getCh

```
          return () -> {
              System.out.println("supplier call");
              return new StringBuffer();
          };
      }

      @Override
      public BiConsumer<StringBuffer, String> accumulator () {
          return (sb, s) -> {
              System.out.println("accumulator function call,"
                                  + "accumulator container: "
                                  + System.identityHashCode(sb)
                                  + " thread: "
                                  + Thread.currentThread().getName()
                                  + ", processing: " + s);
              sb.append(" ").append(s);
          };
      }

      @Override
      public BinaryOperator<StringBuffer> combiner () {
          return (stringBuilder, s) -> {
              System.out.println("combiner function call");
              return stringBuilder.append(s);
          };
      }

      @Override
      public Function<StringBuffer, String> finisher () {
          return stringBuilder -> stringBuilder.toString();
      }

      @Override
      public Set<Characteristics> characteristics () {
          // return Collections.emptySet();
          return EnumSet.of(Collector.Characteristics.CONCURRENT);
      }
  }
}
```

Output

```
supplier call
accumulator function call, accumulator container: 1791741888 thread: main, processing: Nicki
accumulator function call, accumulator container: 1791741888 thread: ForkJoinPool.commonPool-worker-2, processing: John
accumulator function call, accumulator container: 1791741888 thread: ForkJoinPool.commonPool-worker-1, processing: Mike
 Nicki John Mike
```

It's clear that supplier function was called only once even for parallel stream. Let's modify the example a little and return empty set from `characteristics()` method, the rest is unchanged. This time we will have this output:

```
supplier call
supplier call
supplier call
accumulator function call, accumulator container: 668386784 thread: main, processing: Nicki
accumulator function call, accumulator container: 1697408497 thread: ForkJoinPool.commonPool-worker-1, processing: Mike
accumulator function call, accumulator container: 1014427870 thread: ForkJoinPool.commonPool-worker-2, processing: John
combiner function call
combiner function call
 Mike Nicki John
```

Now let's remove unordered() from the pipeline and put back the CONCURRENT characteristic

Java 1
except
Java 1
Java 1
Collect
Java 1
Java 1
Java 1
Git - U
Git - C
'branc
TypeSc
TypeSc
TypeSc
Jackso
Install
startec
Mongo
JPA Cr
Operat
JPA Cr
index(
JPA Cr
Operat
JPA Cr
Manipu
TypeSc
Obtain
Object
Git - H
over S
Groov
Groov
Groov
private
Groov
Groov
Groov
Groov
Gettin
Aparap
Extrac
Mount
Jackso
Annota
Jackso
JSON i
Linux
Java C
Compl
Java -
and vi
JPA Cr
metho
Git - R
Spring
Backe
Groov
Groov
Groov

```
package com.logicbig.example;

import java.util.Collections;
import java.util.EnumSet;
import java.util.Set;
import java.util.function.BiConsumer;
import java.util.function.BinaryOperator;
import java.util.function.Function;
import java.util.function.Supplier;
import java.util.stream.Collector;
import java.util.stream.Stream;

public class CollectorExample2 {
    public static void main (String[] args) {
        String s = Stream.of("Mike", "Nicki", "John")
                         .parallel()
                     //  .unordered()
                         .collect(new MyCollector());
        System.out.println(s);
    }

    private static class MyCollector implements
                        Collector<String, StringBuffer, String> {

        ....
        //the rest is the same

        @Override
        public Set<Characteristics> characteristics () {
            // return Collections.emptySet();
            return EnumSet.of(Collector.Characteristics.CONCURRENT);
        }
    }
}
```

Output

```
supplier call
supplier call
supplier call
accumulator function call, accumulator container: 1897705563 thread: ForkJoinPool.commonPool-worker-2, processing: Mike
accumulator function call, accumulator container: 1329552164 thread: main, processing: Nicki
accumulator function call, accumulator container: 1674443221 thread: ForkJoinPool.commonPool-worker-1, processing: John
combiner function call
combiner function call
 Mike Nicki John
```

It proves that Characteristics.CONCURRENT is ignored for ordered stream source.

When/why we should use CONCURRENT characteristic?

It is a common mistake to think that for a mutable container like ArrayList we should use CONCURRENT characteristic. No, that's not a deciding factor because even though ArrayList is not thread-safe, one instance of this mutable container will **normally** be used by only one thread in parallel execution.
So when we should use CONCURRENT? One scenario (as stated by the API docs) is:
It can be expensive to merge multiple results containers from multiple threads into one final result (normally done in **combiner()** function) e.g. merging multiple HashMap instances by keys. The performance can be improved by specifying CONCURRENT characteristic and replacing HashMap with ConcurrentHashMap. Also remember this performance improvement is only possible for unordered stream sources.

Uploac
withou
Java C
Java I
JPA Cr
Metho
How t
Outpu
Git - U
Install
single
Java C
To Jav
Java S
positic
Java S
TypeS
TypeS
TypeS
Studio
How t
Code I
Groov
Groov
Groov
Groov
Git - U
JPA Cr
metho
JPA Cr
JPA Cr
of Crit
JPA Cr
Spring
@Last
an ent
JPA Cr
Criteri
TypeS
brows
TypeS
tsconfi
JavaS
Export
Java C
Two C
Compl
Groov
Groov
Java S
curren
Groov
Groov
Groov
Groov
Groov
Git - I
Git - S
Tree I
Git - C

2. **Characteristics.UNORDERED**

   This characteristic declares that this Collector instance doesn't want to respect encountered order and the caller (which of course is stream pipeline) is enforced to ignore any order it has. This is particularly important if we have also specified Characteristics.CONCURRENT and don't want stream source's encounter order to be deciding factor to ignore CONCURRENT characteristic.

   Let's modify our last example once more. We are going to remove **unordered()** from the stream pipeline and going to include Characteristics.UNORDERED along with Characteristics.CONCURRENT in our collector implementation

```
public class CollectorExample3 {
    public static void main (String[] args) {
        String s = Stream.of("Mike", "Nicki", "John")
                         .parallel()
                         .collect(new MyCollector());
        System.out.println(s);
    }

    private static class MyCollector implements
                        Collector<String, StringBuffer, String> {

        ....
        // rest is same

        @Override
        public Set<Characteristics> characteristics () {
            //  return Collections.emptySet();
            return EnumSet.of(Collector.Characteristics.CONCURRENT
                             , Characteristics.UNORDERED);
        }
    }
}
```

   Output

```
supplier call
accumulator function call, accumulator container: 1791741888 thread: main, processing: Nicki
accumulator function call, accumulator container: 1791741888 thread: ForkJoinPool.commonPool-worker-2, processing: John
accumulator function call, accumulator container: 1791741888 thread: ForkJoinPool.commonPool-worker-1, processing: Mike
 Nicki John Mike
```

   This time only one instance of the mutable container is used even we have ordered stream.

3. **Characteristics.IDENTITY_FINISH**

   Indicates that the finisher function is the identity function and can be skipped by the pipeline. Specifying this characteristic, we tell the pipeline that we are not performing a final transformation and finisher function should not even be invoked. With this, our Collector implementation becomes equivalent to using three parameters version of collect method i.e. `collect(Supplier, BiConsumer, BiConsumer)`

```
package com.logicbig.example;

import java.util.ArrayList;
import java.util.EnumSet;
import java.util.List;
import java.util.Set;
import java.util.function.BiConsumer;
import java.util.function.BinaryOperator;
import java.util.function.Function;
import java.util.function.Supplier;
import java.util.stream.Collector;
```

JavaSc
Modul
JavaSc
JavaSc
global
JavaSc
With N
Spring
using
JavaSc
JavaSc
Java S
JPA Cr
JPA Cr
Predic
Jackso
deseri
Git - U
Spring
with @
Spring
Declar
Spring
JavaSc
JavaSc
ES6 It
JavaSc
JavaSc
JavaSc
TypeSc
Spring
Classe
Using
new m
progra
Spring
Scope
Groov
Groov
Groov
Groov
Groov
Groov
Groov
Gettin
JPA Cr
Predic
Gradle
Java C
Stages
Spring
to retr
Spring
Object
Spring
via Ob
Spring
spring
Spring
Progra

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

public class CollectorExample4 {
    public static void main (String[] args) {
        List<String> s = Stream.of("Mike", "Nicki", "John")
                               .parallel()
                               .collect(new MyCollector());
        System.out.println(s);
    }

    private static class MyCollector implements
                        Collector<String, List<String>, List<String>> {


        @Override
        public Supplier<List<String>> supplier () {
            return ArrayList::new;
        }

        @Override
        public BiConsumer<List<String>, String> accumulator () {
            return List::add;
        }

        @Override
        public BinaryOperator<List<String>> combiner () {
            return (list, list2) -> {
                list.addAll(list2);
                return list;
            };
        }

        @Override
        public Function<List<String>, List<String>> finisher () {
            return null;
        }

        @Override
        public Set<Characteristics> characteristics () {
            //   return Collections.emptySet();
            return EnumSet.of(Characteristics.IDENTITY_FINISH);
        }
    }
}
```

Output

```
[Mike, Nicki, John]
```

If characteristic() returns empty set, in the same example:

```
Exception in thread "main" java.lang.NullPointerException
	at java.util.stream.ReferencePipeline.collect(ReferencePipeline.java:503)
	at com.logicbig.example.CollectorExample4.main(CollectorExample4.java:15)
	at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
	at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
	at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
	at java.lang.reflect.Method.invoke(Method.java:498)
	at com.intellij.rt.execution.application.AppMain.main(AppMain.java:144)
```

Spring
Spring
Spring uncon proper
Git - C
Install
TypeS
TypeS
TypeS
TypeS
TypeS
TypeS
TypeS
TypeS
Spring custon
Spring
Jackso always
Jackso using
JPA - / @Colu
JPA - / @Tabl
JPA Cr isNotM
JPA Cr Empty
Java C Stages
Spring Proper
Spring Comm
Spring
Spring
Java 1
TypeS
TypeS
TypeS
TypeS
TypeS
Spring Specif
Java 9 jlink
JPA Cr
JPA Cr applyi

Collectors class

Collectors class defines various useful factory creation of Collector instances which includes converting to collections and maps, aggregation, grouping, partitioning etc.

Browse the Collectors examples here . They are also included under the package com.logicbig.example.collectors in the project browser below.

Example project

Dependencies and Technologies Used:

- JDK 10
- Maven 3.3.9

Stream Collect Examples

streams-collect-examples
    src
        main
            java
                com
                    logicbig
                        example
                            collectors
                                CollectorExample.java
                                CollectorExample2.java
                                CollectorExample3.java
                                CollectorExample4.java
                                MutableReductionExamp
                                MutableReductionExamp
    pom.xml

```
            public Supplier<StringBuilder> supplier () {
                return StringBuilder::new;
            }

            @Override
            public BiConsumer<StringBuilder, String> accumulator () {
                return (sb, s) -> sb.append(" ").append(s);
            }

            @Override
            public BinaryOperator<StringBuilder> combiner () {
                return (sb1, sb2) -> sb1.append(sb2);
            }

            @Override
            public Function<StringBuilder, String> finisher () {
                return stringBuilder -> stringBuilder.toString();
            }

            @Override
            public Set<Characteristics> characteristics () {
                return Collections.emptySet();
            }
        }
    }
```

Project Structure

streams-collect-examples
    src
        main
            java
                com
                    logicbig
                        example
                            collectors

CollectorExample.java
CollectorExample2.java
CollectorExample3.java
CollectorExample4.java
MutableReductionExample.java
MutableReductionExample2.java
pom.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.logicbig.example</groupId>
    <artifactId>streams-collect-examples</artifactId>
    <version>1.0-SNAPSHOT</version>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.5.1</version>
                <configuration>
                    <source>10</source>
                    <target>10</target>
                    <encoding>UTF-8</encoding>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

```java
package com.logicbig.example;

import java.util.Collections;
import java.util.Set;
import java.util.function.BiConsumer;
import java.util.function.BinaryOperator;
import java.util.function.Function;
import java.util.function.Supplier;
import java.util.stream.Collector;
import java.util.stream.Stream;

public class CollectorExample {
    public static void main (String[] args) {
        String s = Stream.of("Mike", "Nicki", "John").collect(new
                          MyCollector());
        System.out.println(s);
    }

    private static class MyCollector implements
                        Collector<String, StringBuilder, String> {

        @Override
        public Supplier<StringBuilder> supplier () {
            return StringBuilder::new;
        }

        @Override
        public BiConsumer<StringBuilder, String> accumulator () {
            return (sb, s) -> sb.append(" ").append(s);
        }
        @Override
        public BinaryOperator<StringBuilder> combiner () {
            return (sb1, sb2) -> sb1.append(sb2);
        }

        @Override
        public Function<StringBuilder, String> finisher () {
            return stringBuilder -> stringBuilder.toString();
        }

        @Override
        public Set<Characteristics> characteristics () {
            return Collections.emptySet();
        }
    }
}
```

```java
package com.logicbig.example;

import java.util.EnumSet;
import java.util.Set;
import java.util.function.BiConsumer;
import java.util.function.BinaryOperator;
import java.util.function.Function;
import java.util.function.Supplier;
import java.util.stream.Collector;
import java.util.stream.Stream;

public class CollectorExample2 {
    public static void main (String[] args) {
        String s = Stream.of("Mike", "Nicki", "John")
                         .parallel()
                         .unordered()
                         .collect(new MyCollector());
        System.out.println(s);
    }

    private static class MyCollector implements
                        Collector<String, StringBuffer, String> {

        @Override
        public Supplier<StringBuffer> supplier () {
            return () -> {
                System.out.println("supplier call");
                return new StringBuffer();
            };
        }

        @Override
        public BiConsumer<StringBuffer, String> accumulator () {
            return (sb, s) -> {
                System.out.println("accumulator function call,"
                                    + " accumulator container: "
                                    + System.identityHashCode(sb)
                                    + " thread: "
                                    + Thread.currentThread().getName()
                                    + ", processing: " + s);
                sb.append(" ").append(s);
            };
        }

        @Override
        public BinaryOperator<StringBuffer> combiner () {
            return (stringBuilder, s) -> {
                System.out.println("combiner function call");
                return stringBuilder.append(s);
            };
        }

        @Override
        public Function<StringBuffer, String> finisher () {
            return stringBuilder -> stringBuilder.toString();
        }

        @Override
        public Set<Characteristics> characteristics () {
            // return Collections.emptySet();
            return EnumSet.of(Characteristics.CONCURRENT);
        }
    }
}
```

```
package com.logicbig.example;

import java.util.EnumSet;
import java.util.Set;
import java.util.function.BiConsumer;
import java.util.function.BinaryOperator;
import java.util.function.Function;
import java.util.function.Supplier;
import java.util.stream.Collector;
import java.util.stream.Stream;

public class CollectorExample3 {
    public static void main (String[] args) {
        String s = Stream.of("Mike", "Nicki", "John")
                         .parallel()
                         .collect(new MyCollector());
        System.out.println(s);
    }

    private static class MyCollector implements
                        Collector<String, StringBuffer, String> {

        @Override
        public Supplier<StringBuffer> supplier () {
            return () -> {
                System.out.println("supplier call");
                return new StringBuffer();
            };
        }

        @Override
        public BiConsumer<StringBuffer, String> accumulator () {
            return (sb, s) -> {
                System.out.println("accumulator function call,"
                                    + " accumulator container: "
                                    + System.identityHashCode(sb)
                                    + " thread: "
                                    + Thread.currentThread().getName()
                                    + ", processing: " + s);
                sb.append(" ").append(s);
            };
        }

        @Override
        public BinaryOperator<StringBuffer> combiner () {
            return (stringBuilder, s) -> {
                System.out.println("combiner function call");
                return stringBuilder.append(s);
            };
        }

        @Override
        public Function<StringBuffer, String> finisher () {
            return stringBuilder -> stringBuilder.toString();
        }

        @Override
        public Set<Characteristics> characteristics () {
            //  return Collections.emptySet();
            return EnumSet.of(Characteristics.CONCURRENT
                                , Characteristics.UNORDERED);
        }
    }
}
```

```
package com.logicbig.example;

import java.util.ArrayList;
import java.util.EnumSet;
import java.util.List;
import java.util.Set;
import java.util.function.BiConsumer;
import java.util.function.BinaryOperator;
import java.util.function.Function;
import java.util.function.Supplier;
import java.util.stream.Collector;
import java.util.stream.Stream;

public class CollectorExample4 {
    public static void main (String[] args) {
        List<String> s = Stream.of("Mike", "Nicki", "John")
                               .parallel()
                               .collect(new MyCollector());
        System.out.println(s);
    }

    private static class MyCollector implements
                        Collector<String, List<String>, List<String>> {


        @Override
        public Supplier<List<String>> supplier () {
            return ArrayList::new;
        }

        @Override
        public BiConsumer<List<String>, String> accumulator () {
            return List::add;
        }

        @Override
        public BinaryOperator<List<String>> combiner () {
            return (list, list2) -> {
                list.addAll(list2);
                return list;
            };
        }

        @Override
        public Function<List<String>, List<String>> finisher () {
            return null;
        }
        @Override
        public Set<Characteristics> characteristics () {
            //  return Collections.emptySet();
            return EnumSet.of(Characteristics.IDENTITY_FINISH);
        }
    }
}
```

```
package com.logicbig.example.collectors;


import java.math.BigDecimal;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class AveragingDoubleExample {
    public static void main (String[] args) {
        Stream<BigDecimal> s = Stream.iterate(
                            BigDecimal.ONE, bigDecimal ->
                                                bigDecimal.add(BigDecimal.ONE))
                                     .limit(10).peek(System.out::println);

        Double d = s.collect(Collectors.averagingDouble(BigDecimal::doubleValue));
        System.out.println("average: " + d);
    }
}
```

```
package com.logicbig.example.collectors;


import java.math.BigDecimal;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class AveragingIntExample {
    public static void main (String[] args) {
        Stream<BigDecimal> s = Stream.iterate(
                            BigDecimal.ONE, bigDecimal ->
                                                bigDecimal.add(BigDecimal.ONE))
                                     .limit(10).peek(System.out::println);

        Double d = s.collect(Collectors.averagingInt(BigDecimal::intValue));
        System.out.println("average: " + d);
    }
}
```

```
package com.logicbig.example.collectors;

import java.math.BigDecimal;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class AveragingLongExample {
    public static void main (String[] args) {
        Stream<BigDecimal> s = Stream.iterate(
                            BigDecimal.ONE, bigDecimal ->
                                                bigDecimal.add(BigDecimal.ONE))
                                     .limit(10).peek(System.out::println);
        Double d = s.collect(Collectors.averagingLong(BigDecimal::longValue));
        System.out.println("average: " + d);
    }
}
```

```
package com.logicbig.example.collectors;

import java.util.Collections;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class CollectingAndThenExample {
    public static void main (String[] args) {
        Stream<String> s = Stream.of("apple", "banana", "orange");
        List<String> synchronizedList = s.collect(Collectors.collectingAndThen(
                                Collectors.toList(), Collections::synchronizedList));
        System.out.println(synchronizedList);
    }
}
```

```
package com.logicbig.example.collectors;

import java.util.stream.Collectors;
import java.util.stream.Stream;

public class CountingExample {
    public static void main (String[] args) {
        Stream<String> s = Stream.of("apple", "banana", "orange");
        Long c = s.collect(Collectors.counting());
        System.out.println(c);
    }
}
```

```
package com.logicbig.example.collectors;

import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

public class FilteringExample {

    public static void main(String... args) {
        List<Integer> list = IntStream.of(2, 4, 6, 8, 10, 12)
                                      .boxed()
                                      .collect(Collectors.filtering(i -> i % 4 == 0,
                                              Collectors.toList()));
        System.out.println(list);

    }
}
```

```
package com.logicbig.example.collectors;

import java.util.List;
import java.util.Map;
import java.util.Objects;
```

```java
import java.util.Set;
import java.util.stream.Collectors;

public class FilteringExample2 {

    public static void main(String... args) {

        Map<Department, Set<Employee>> wellPaidEmployeesByDepartment
                = getEmployees().stream().collect(
                Collectors.groupingBy((Employee employee) -> employee.dept, Collectors
                        .filtering(e -> e.salary >= 2000,
                                Collectors.toSet())));
        wellPaidEmployeesByDepartment.forEach((k, v) -> System.out.printf("%8s: %s%n", k.name, v));
    }

    private static List<Employee> getEmployees() {
        return List.of(
                new Employee("Sara", new Department("Admin"), 3000),
                new Employee("Joe", new Department("IT"), 1000),
                new Employee("Mike", new Department("Account"), 2000),
                new Employee("Tony", new Department("Account"), 1500),
                new Employee("Linda", new Department("IT"), 5000)
        );
    }

    private static class Employee {
        private String name;
        private Department dept;
        private int salary;

        public Employee(String name, Department dept, int salary) {
            this.name = name;
            this.dept = dept;
            this.salary = salary;
        }

        @Override
        public String toString() {
            return "Employee{" +
                    "name='" + name + '\'' +
                    ", dept=" + dept +
                    ", salary=" + salary +
                    '}';
        }
    }

    private static class Department {
        private String name;

        public Department(String name) {
            this.name = name;
        }

        @Override
        public String toString() {
            return "Department{" +
                    "name='" + name + '\'' +
                    '}';
        }

        @Override
        public boolean equals(Object o) {
            Department that = (Department) o;
            return Objects.equals(name, that.name);
        }

        @Override
        public int hashCode() {
            return Objects.hash(name);
        }
    }
}
```

```java
package com.logicbig.example.collectors;

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

public class FlatMappingExample {

    public static void main(String... args) {
        List<Integer> list = Stream.of(List.of(1, 2, 3, 4), List.of(5, 6, 7, 8))
                                   .collect(
                                           Collectors.flatMapping(
                                                   l -> l.stream()
                                                         .filter(i -> i % 2 == 0),
                                                   Collectors.toList()
                                           )
                                   );
        System.out.println(list);
    }
}
```

```java
package com.logicbig.example.collectors;

import java.util.Collection;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class FlatMappingExample2 {

    public static void main(String... args) {
        Map<Integer, List<Integer>> map =
                Stream.of(List.of(1, 2, 3, 4, 5, 6), List.of(7, 8, 9, 10))
                      .collect(
                              Collectors.groupingBy(
                                      Collection::size,
                                      Collectors.flatMapping(
                                              l -> l.stream()
                                                    .filter(i -> i % 2 == 0),
                                              Collectors.toList())
                              )
                      );
        System.out.println(map);
    }
}
```

```java
package com.logicbig.example.collectors;

import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class GroupingByConcurrent {
    public static void main (String[] args) {
        Stream<String> s = Stream.of("apple", "banana", "orange");
        Map<Integer, List<String>> map = s.collect(
                            Collectors.groupingByConcurrent(String::length));
        System.out.println(map);
    }
}
```

```java
package com.logicbig.example.collectors;

import java.util.Map;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class GroupingByConcurrentExample2 {
    public static void main (String[] args) {
        Stream<String> s = Stream.of("apple", "banana", "orange");
        //cascaded group by
        Map<Integer, Long> map = s.collect(
                            Collectors.groupingByConcurrent(String::length,
                                                Collectors.counting()));
        System.out.println(map);
    }
}
```

```java
package com.logicbig.example.collectors;

import java.util.concurrent.ConcurrentSkipListMap;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class GroupingByConcurrentExample3 {
    public static void main (String[] args) {
        Stream<String> s = Stream.of("apple", "banana", "orange");
        //cascaded group by
        ConcurrentSkipListMap<Integer, Long> map = s.collect(
                            Collectors.groupingByConcurrent(String::length,
                                                ConcurrentSkipListMap::new,
                                                Collectors.counting()));
        System.out.println(map);
    }
}
```

```java
package com.logicbig.example.collectors;

import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;
import java.util.stream.Stream;


public class GroupingByExample {
    public static void main (String[] args) {
        Stream<String> s = Stream.of("apple", "banana", "orange");
        Map<Integer, List<String>> map = s.collect(
                            Collectors.groupingBy(String::length));
        System.out.println(map);
    }
}
```

```java
package com.logicbig.example.collectors;

import java.util.Map;
import java.util.stream.Collectors;
import java.util.stream.Stream;

/**
 * Created by Joe on 11/27/2016.
 */
public class GroupingByExample2 {
    public static void main (String[] args) {
        Stream<String> s = Stream.of("apple", "banana", "orange");
        //cascaded group by
        Map<Integer, Long> map = s.collect(
                            Collectors.groupingBy(String::length, Collectors.counting()));
        System.out.println(map);
    }
}
```

```java
package com.logicbig.example.collectors;

import java.util.concurrent.ConcurrentHashMap;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class GroupingByExample3 {
    public static void main (String[] args) {
        Stream<String> s = Stream.of("apple", "banana", "orange");
        //cascaded group by
        ConcurrentHashMap<Integer, Long> map = s.collect(
                            Collectors.groupingBy(String::length,
                                                ConcurrentHashMap::new,
                                                Collectors.counting()));
        System.out.println(map);
    }
}
```

```java
package com.logicbig.example.collectors;

import java.util.stream.Collectors;
import java.util.stream.Stream;

public class JoiningExample {
    public static void main (String[] args) {
        Stream<String> s = Stream.of("what", "so", "ever");
        String str = s.collect(Collectors.joining());
        System.out.println(str);
    }
}
```

```java
package com.logicbig.example.collectors;

import java.util.stream.Collectors;
import java.util.stream.Stream;

public class JoiningExample2 {
    public static void main (String[] args) {
        Stream<String> s = Stream.of("what", "so", "ever");
        String str = s.collect(Collectors.joining("|"));
        System.out.println(str);
    }
}
```

```
package com.logicbig.example.collectors;

import java.util.stream.Collectors;
import java.util.stream.Stream;

public class JoiningExample3 {
    public static void main (String[] args) {
        Stream<String> s = Stream.of("what", "so", "ever");
        String str = s.collect(Collectors.joining("|", "-", "!"));
        System.out.println(str);
    }
}
```

```
package com.logicbig.example.collectors;

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

public class MappingExample {
    public static void main (String[] args) {
        Stream<String> s = Stream.of("apple", "banana", "orange");
        List<String> list = s.collect(Collectors.mapping(s1 -> s1.substring(0, 2),
                                    Collectors.toList()));
        System.out.println(list);
    }
}
```

```
package com.logicbig.example.collectors;

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

public class MaxByExample {
    public static void main (String[] args) {
        Stream<String> s = Stream.of("orange", "apple", "banana");
        Optional<String> o = s.collect(
                                    Collectors.maxBy(String::compareTo));
        System.out.println(o.get());
    }
}
```

```
package com.logicbig.example.collectors;

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


public class MinByExample {
    public static void main (String[] args) {
```

```java
        Stream<String> s = Stream.of("orange", "apple", "banana");
        Optional<String> o = s.collect(
                            Collectors.minBy(String::compareTo));
        System.out.println(o.get());
    }
}
```

```java
package com.logicbig.example.collectors;

import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;
import java.util.stream.IntStream;


public class PartitioningByExample {
    public static void main (String[] args) {
        Map<Boolean, List<Long>> m = IntStream.range(1, 10)
                                              .mapToObj(Long::new)
                                              .collect(Collectors.partitioningBy(
                                                       i -> i % 2 == 0));
        System.out.println(m);

    }
}
```

```java
package com.logicbig.example.collectors;

import java.util.Map;
import java.util.stream.Collectors;
import java.util.stream.Stream;


public class PartitioningByExample2 {
    public static void main (String[] args) {
        Stream<String> s = Stream.of("Ace", "heart", "Club", "diamond");
        Map<Boolean, Long> m = s.collect(Collectors.partitioningBy(
                            x -> Character.isUpperCase(x.charAt(0)),
                            Collectors.counting()));
        System.out.println(m);
    }
}
```

```java
package com.logicbig.example.collectors;

import java.util.stream.Collectors;
import java.util.stream.Stream;

public class ReducingExample {
    public static void main (String[] args) {
        Stream<Integer> s = Stream.of(5, 10, 20, 50);
        Integer i = s.collect(Collectors.reducing((integer, integer2)
                            -> integer2 - integer))
                     .orElse(-1);

        System.out.println(i);
    }
}
```

```java
package com.logicbig.example.collectors;

import java.util.stream.Collectors;
import java.util.stream.Stream;

public class ReducingExample2 {
    public static void main (String[] args) {
        Stream<Integer> s = Stream.of(5, 10, 20, 50);
        Integer i = s.collect(Collectors.reducing(1, (integer, integer2)
                            -> integer2 * integer));
        System.out.println(i);
    }
}
```

```java
package com.logicbig.example.collectors;

import java.util.stream.Collectors;
import java.util.stream.Stream;

public class ReducingExample3 {
    public static void main (String[] args) {
        Stream<Integer> s = Stream.of(5, 10, 20, 50).parallel();
        String str = s.collect(Collectors.reducing(
                            "",
                            x -> Integer.toString(x),
                            (s1, s2) -> s1 + s2));
        System.out.println(str);
    }
}
```

```java
package com.logicbig.example.collectors;

import java.math.BigDecimal;
import java.util.DoubleSummaryStatistics;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class SummarizingDoubleExample {
    public static void main (String[] args) {

        Stream<BigDecimal> s = Stream.iterate(
                            BigDecimal.ONE, bigDecimal ->
                                        bigDecimal.add(BigDecimal.ONE))
                                       .limit(10).peek(System.out::println);

        DoubleSummaryStatistics d = s.collect(Collectors.summarizingDouble
                                                (BigDecimal::doubleValue));
        System.out.println(d);
    }
}
```

```java
package com.logicbig.example.collectors;

import java.math.BigDecimal;
import java.util.IntSummaryStatistics;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;
```

```
public class SummarizingIntExample {
    public static void main (String[] args) {
        Stream<BigDecimal> s = Stream.iterate(
                            BigDecimal.ONE, bigDecimal ->
                                      bigDecimal.add(BigDecimal.ONE))
                                       .limit(10).peek(System.out::println);

        IntSummaryStatistics i = s.collect(Collectors.summarizingInt
                                                      (BigDecimal::intValue));
        System.out.println(i);
    }
}
```

```
package com.logicbig.example.collectors;

import java.math.BigDecimal;
import java.util.DoubleSummaryStatistics;
import java.util.LongSummaryStatistics;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class SummarizingLongExample {
    public static void main (String[] args) {

        Stream<BigDecimal> s = Stream.iterate(
                            BigDecimal.ONE, bigDecimal ->
                                                bigDecimal.add(BigDecimal.ONE))
                                                 .limit(10).peek(System.out::println);

        LongSummaryStatistics l = s.collect(Collectors.summarizingLong
                                                    (BigDecimal::longValue));
        System.out.println(l);
    }
}
```

```
package com.logicbig.example.collectors;

import java.math.BigDecimal;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class SummingDoubleExample {
    public static void main (String[] args) {

        Stream<BigDecimal> s = Stream.iterate(BigDecimal.ONE, bigDecimal ->
                                              bigDecimal.add(BigDecimal.ONE))
                                     .limit(10).peek(System.out::println);

        double d = s.collect(Collectors.summingDouble(
                                        (BigDecimal::doubleValue)));
        System.out.println("sum: " + d);
    }
}
```

```
package com.logicbig.example.collectors;

import java.math.BigDecimal;
import java.util.stream.Collectors;
import java.util.stream.Stream;
```

```
public class SummingIntExample {
    public static void main (String[] args) {

        Stream<BigDecimal> s = Stream.iterate(
                            BigDecimal.ONE, bigDecimal ->
                                                bigDecimal.add(BigDecimal.ONE))
                                     .limit(10).peek(System.out::println);

        int i = s.collect(Collectors.summingInt(
                            (BigDecimal::intValue)));
        System.out.println("sum: " + i);
    }
}
```

```
package com.logicbig.example.collectors;

import java.math.BigDecimal;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class SummingLongExample {
    public static void main (String[] args) {

        Stream<BigDecimal> s = Stream.iterate(
                                         BigDecimal.ONE, bigDecimal ->
                                                  bigDecimal.add(BigDecimal.ONE))
                                     .limit(10)
                                     .peek(System.out::println);

        long l = s.collect(Collectors.summingLong(
                            (BigDecimal::longValue)));
        System.out.println("sum: " + l);
    }
}
```

```
package com.logicbig.example.collectors;

import java.math.BigDecimal;
import java.util.LinkedList;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class ToCollectionExample {
    public static void main (String[] args) {
        Stream<BigDecimal> s = Stream.iterate(
                            BigDecimal.ONE, bigDecimal ->
                                                  bigDecimal.add(BigDecimal.ONE))
                                     .limit(10)
                                     .peek(System.out::println);

        LinkedList<BigDecimal> l = s.collect(
                            Collectors.toCollection(LinkedList::new));
        System.out.println(l);
    }
}
```

```
package com.logicbig.example.collectors;

import java.util.concurrent.ConcurrentMap;
import java.util.stream.Collectors;
```

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

public class ToConcurrentMapExample {
    public static void main (String[] args) {
        Stream<String> s = Stream.of("apple", "banana", "orange");
        ConcurrentMap<Character, String> m = s.collect(
                                Collectors.toConcurrentMap(s1 -> s1.charAt(0),
                                                    String::toUpperCase));
        System.out.println(m);
    }
}
```

```java
package com.logicbig.example.collectors;

import java.util.concurrent.ConcurrentMap;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class ToConcurrentMapExample1 {
    public static void main (String[] args) {
        Stream<String> s = Stream.of("apple", "banana", "apricot", "orange");
        ConcurrentMap<Character, String> m = s.collect(
                                Collectors.toConcurrentMap(s1 -> s1.charAt(0),
                                                    String::toUpperCase));
        System.out.println(m);
    }
}
```

```java
package com.logicbig.example.collectors;

import java.util.Map;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class ToConcurrentMapExample2 {
    public static void main (String[] args) {
        Stream<String> s = Stream.of("apple", "banana", "apricot", "orange");
        Map<Character, String> m = s.collect(
                                Collectors.toConcurrentMap(s1 -> s1.charAt(0),
                                                    String::toUpperCase,
                                                    (s1, s2) -> s1 + "|" + s2));
        System.out.println(m);
    }
}
```

```java
package com.logicbig.example.collectors;

import java.util.concurrent.ConcurrentHashMap;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class ToConcurrentMapExample3 {
    public static void main (String[] args) {
        Stream<String> s = Stream.of("apple", "banana", "apricot", "orange");
        ConcurrentHashMap<Character, String> m = s.collect(
                                Collectors.toConcurrentMap(s1 -> s1.charAt(0),
                                                    String::toUpperCase,
                                                    (s1, s2) -> s1 + "|" + s2,
                                                    ConcurrentHashMap::new));
        System.out.println(m);
    }
}
```

```
package com.logicbig.example.collectors;

import java.math.BigDecimal;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class ToListExample {
    public static void main (String[] args) {
        Stream<BigDecimal> s = Stream.iterate(
                            BigDecimal.ONE, bigDecimal ->
                                                bigDecimal.add(BigDecimal.ONE))
                                     .limit(10)
                                     .peek(System.out::println);

        List<BigDecimal> l = s.collect(Collectors.toList());
        System.out.println(l);
    }
}
```

```
package com.logicbig.example.collectors;

import java.util.Map;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class ToMapExample {
    public static void main (String[] args) {
        Stream<String> s = Stream.of("apple", "banana", "orange");
        Map<Character, String> m = s.collect(
                            Collectors.toMap(s1 -> s1.charAt(0),
                                                s1 -> s1));
        System.out.println(m);
    }
}
```

```
package com.logicbig.example.collectors;

import java.util.Map;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class ToMapExample1 {
    public static void main (String[] args) {
        Stream<String> s = Stream.of("apple", "banana", "apricot", "orange");
        Map<Character, String> m = s.collect(
                            Collectors.toMap(s1 -> s1.charAt(0),
                                                s1 -> s1));
        System.out.println(m);
    }
}
```

```
package com.logicbig.example.collectors;
```

```java
import java.util.Map;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class ToMapExample2 {
    public static void main (String[] args) {
        Stream<String> s = Stream.of("apple", "banana", "apricot", "orange",
                                    "apple");
        Map<Character, String> m = s.collect(
                                    Collectors.toMap(s1 -> s1.charAt(0),
                                                        s1 -> s1,
                                                        (s1, s2) -> s1 + "|" + s2));
        System.out.println(m);
    }
}
```

```java
package com.logicbig.example.collectors;

import java.util.LinkedHashMap;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class ToMapExample3 {
    public static void main (String[] args) {
        Stream<String> s = Stream.of("apple", "banana", "apricot", "orange",
                                    "apple");
        LinkedHashMap<Character, String> m = s.collect(
                                    Collectors.toMap(s1 -> s1.charAt(0),
                                                        s1 -> s1,
                                                        (s1, s2) -> s1 + "|" + s2,
                                                        LinkedHashMap::new));
        System.out.println(m);
    }
}
```

```java
package com.logicbig.example.collectors;

import java.math.BigDecimal;
import java.util.Set;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class ToSetExample {
    public static void main (String[] args) {
        Stream<BigDecimal> s = Stream.iterate(
                                    BigDecimal.ONE, bigDecimal ->
                                                        bigDecimal.add(BigDecimal.ONE))
                                            .limit(10)
                                            .peek(System.out::println);

        Set<BigDecimal> l = s.collect(Collectors.toSet());
        System.out.println(l);
    }
}
```

```java
package com.logicbig.example.collectors;

import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
```

```java
public class ToUnmodifiableListExample {

    public static void main(String... args) {
        List<Integer> list = IntStream.range(1, 5)
                                      .boxed()
                                      .collect(Collectors.toUnmodifiableList());
        System.out.println(list);
        System.out.println(list.getClass().getName());
    }
}
```

```java
package com.logicbig.example.collectors;

import java.util.Map;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

public class ToUnmodifiableMapExample {

    public static void main(String... args) {
        Map<Integer, Double> map =
                IntStream.range(1, 5)
                         .boxed()
                         .collect(Collectors.toUnmodifiableMap(
                                 i -> i,
                                 i -> Math.pow(i, 3))
                         );
        System.out.println(map);
        System.out.println(map.getClass().getTypeName());
    }
}
```

```java
package com.logicbig.example.collectors;

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

public class ToUnmodifiableMapExample2 {

    public static void main(String... args) {
        Map<Integer, List<String>> map =
                Stream.of("rover", "joyful", "depth", "hunter")
                      .collect(Collectors.toUnmodifiableMap(
                              String::length,
                              List::of,
                              ToUnmodifiableMapExample2::mergeFunction)
                      );
        System.out.println(map);
    }

    private static List<String> mergeFunction(List<String> l1, List<String> l2) {
        List<String> list = new ArrayList<>();
        list.addAll(l1);
        list.addAll(l2);
        return list;
    }
}
```

```java
package com.logicbig.example.collectors;

import java.util.Set;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

public class ToUnmodifiableSetExample {

    public static void main(String... args) {
        Set<Integer> set = IntStream.range(1, 5)
                                    .boxed()
                                    .collect(Collectors.toUnmodifiableSet());
        System.out.println(set);
        System.out.println(set.getClass().getTypeName());
    }
}
```

```java
package com.logicbig.example;

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

public class MutableReductionExample {
    public static void main (String[] args) {
        List<String> list = Arrays.asList("Mike", "Nicki", "John");
        runMutableCollect(list);
        runImmutableReduce(list);
    }

    private static void runMutableCollect (List<String> list) {
        String s = list.stream().collect(StringBuilder::new,
                            (sb, s1) -> sb.append(" ").append(s1),
                            (sb1, sb2) -> sb1.append(sb2.toString())).toString();
        System.out.println(s);
    }

    private static void runImmutableReduce (List<String> list) {
        String s = list.stream().reduce("", (s1, s2) -> s1 + " " + s2);
        System.out.println(s);
    }
}
```

```java
package com.logicbig.example;


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

public class MutableReductionExample2 {
    public static void main (String[] args) {
        IntStream stream = IntStream.range(1, 100);
        List<Integer> list = stream.parallel()
                                   .filter(i -> i % 10 == 0)
                                   .collect(ArrayList::new, ArrayList::add
                                                    , ArrayList::addAll);
        System.out.println(list);
    }
}
```