
# Good Old Stream API

Tagir Valeev

- ✓ JDK-8072727 Add variation of Stream.iterate() that's finite
- ✓ JDK-8136686 Collectors.counting can use Collectors.summingLong to reduce boxing
- ✓ JDK-8141630 Specification of Collections.synchronized* need to state traversal constraints
- ✓ JDK-8145007 Pattern splitAsStream is not late binding as required by the specification
- ✓ JDK-8146218 Add LocalDate.datesUntil method producing Stream<LocalDate>
- ✓ JDK-8147505 BaseStream.onClose() should not allow registering new handlers after stream is consumed
- ✓ JDK-8148115 Stream.findFirst for unordered source optimization
- ✓ JDK-8148250 Stream.limit() parallel tasks with ordered non-SUBSIZED source should short-circuit
- ✓ JDK-8148838 Stream.flatMap(...).spliterator() cannot properly split after tryAdvance()
- ✓ JDK-8148748 ArrayList.subList().spliterator() is not late-binding
- ✓ JDK-8151123 Collectors.summingDouble/averagingDouble unnecessarily call mapper twice
- ✓ JDK-8153293 Preserve SORTED and DISTINCT characteristics for boxed() and asLongStream() operations
- ✓ JDK-8154387 Parallel unordered Stream.limit() tries to collect 128 elements even if limit is less
- ✓ JDK-8164189 Collectors.toSet() parallel performance improvement
- ✓ JDK-8209685 Create Collector which merges results of two other collectors

amaembo / streamex

Unwatch 79 | Unstar 1.3k | Fork 150

Code | Issues 18 | Pull requests 4 | Actions | Projects | Wiki | Security | ...

master | Go to file | Add file | Clone

README.md

# StreamEx 0.7.2

Enhancing Java Stream API.


This library defines four classes: `StreamEx`, `IntStreamEx`, `LongStreamEx`, `DoubleStreamEx` which are fully compatible with Java 8 stream classes and provide many additional useful methods. Also `EntryStream` class is provided which represents the stream of map entries and provides additional functionality for this case. Finally there are some new useful collectors defined in `MoreCollectors` class as well as primitive collectors concept.

Full API documentation is available here.

Take a look at the Cheatsheet for brief introduction to the StreamEx!

Before updating StreamEx check the migration notes and full list of changes.

StreamEx library main points are following:

- Shorter and convenient ways to do the common tasks.
- Better interoperability with older code.
- 100% compatibility with original JDK streams.
- Friendliness for parallel processing: any new feature takes the advantage on parallel streams as much as possible.

## About

Enhancing Java Stream API

java | java8 | collections | streams-api

Readme

Apache-2.0 License

## Latest release

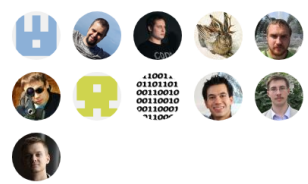
streamex-0.7.2 05c8f36
on 7 Nov 2019

+ 38 releases

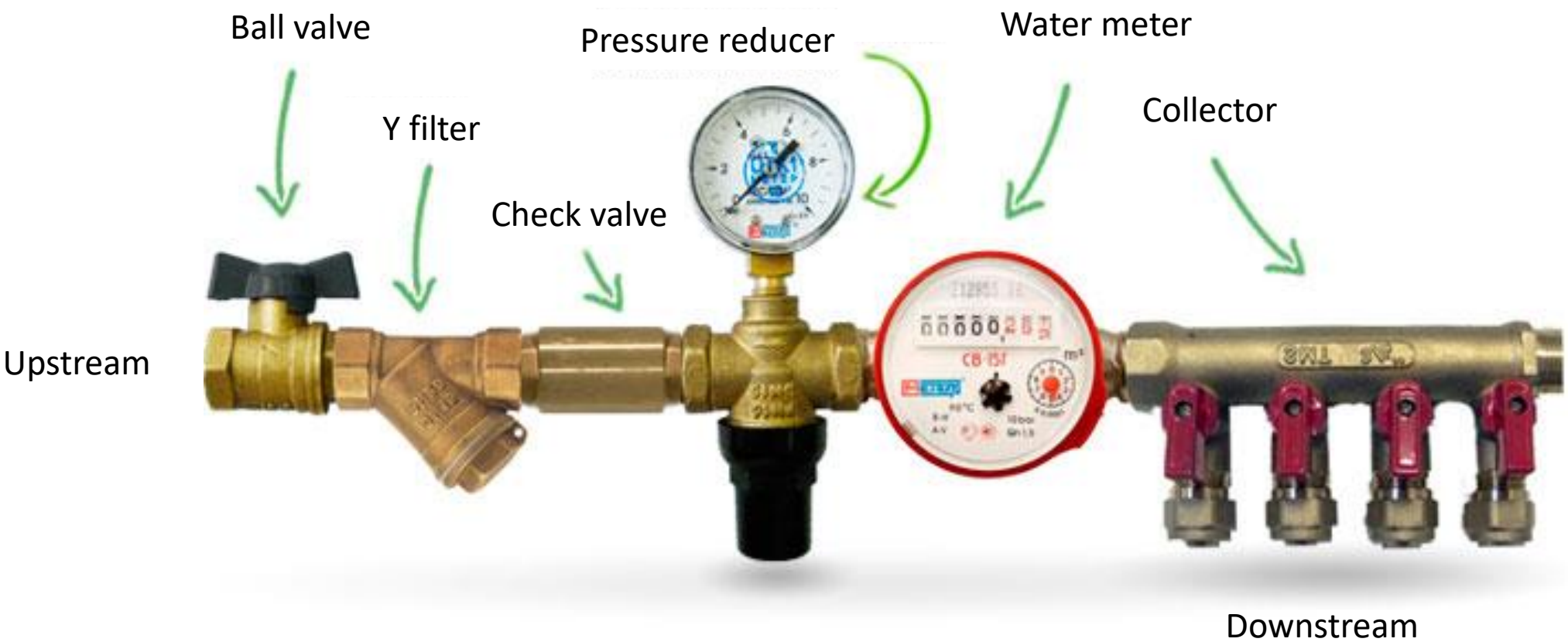
## Used by 419

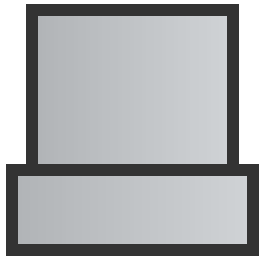
+ 411

## Contributors 15

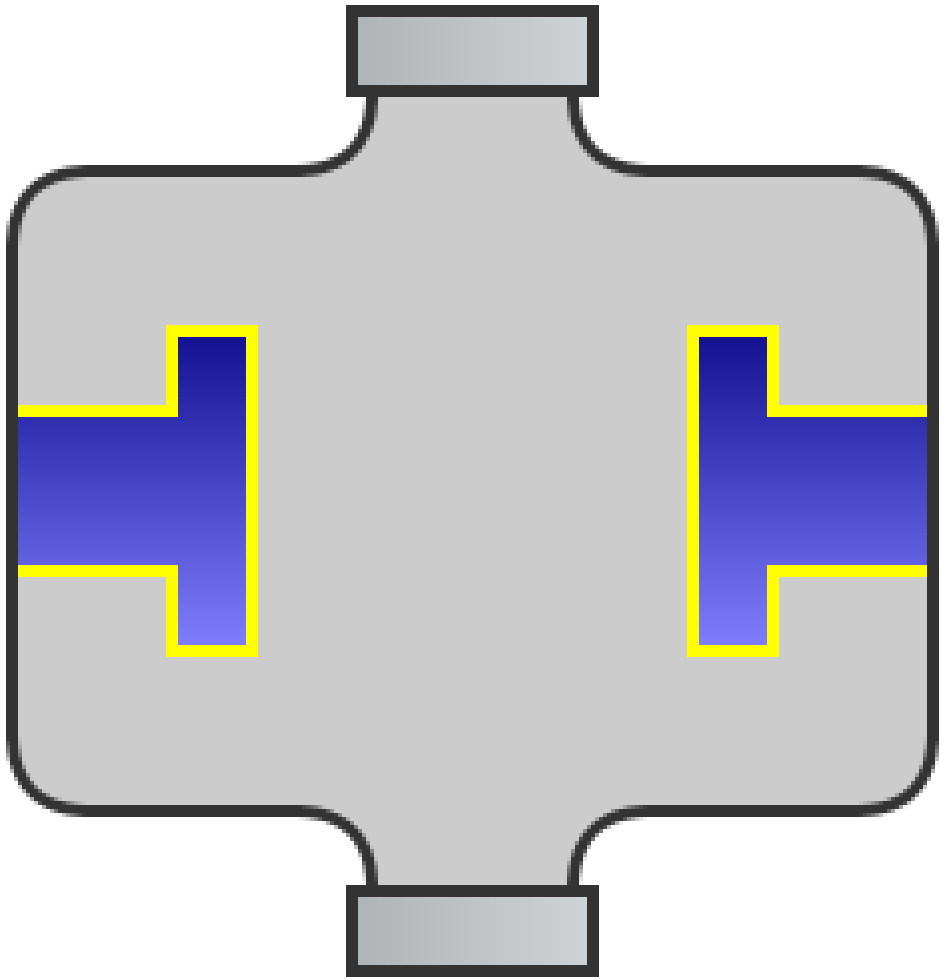


+ 4 contributors

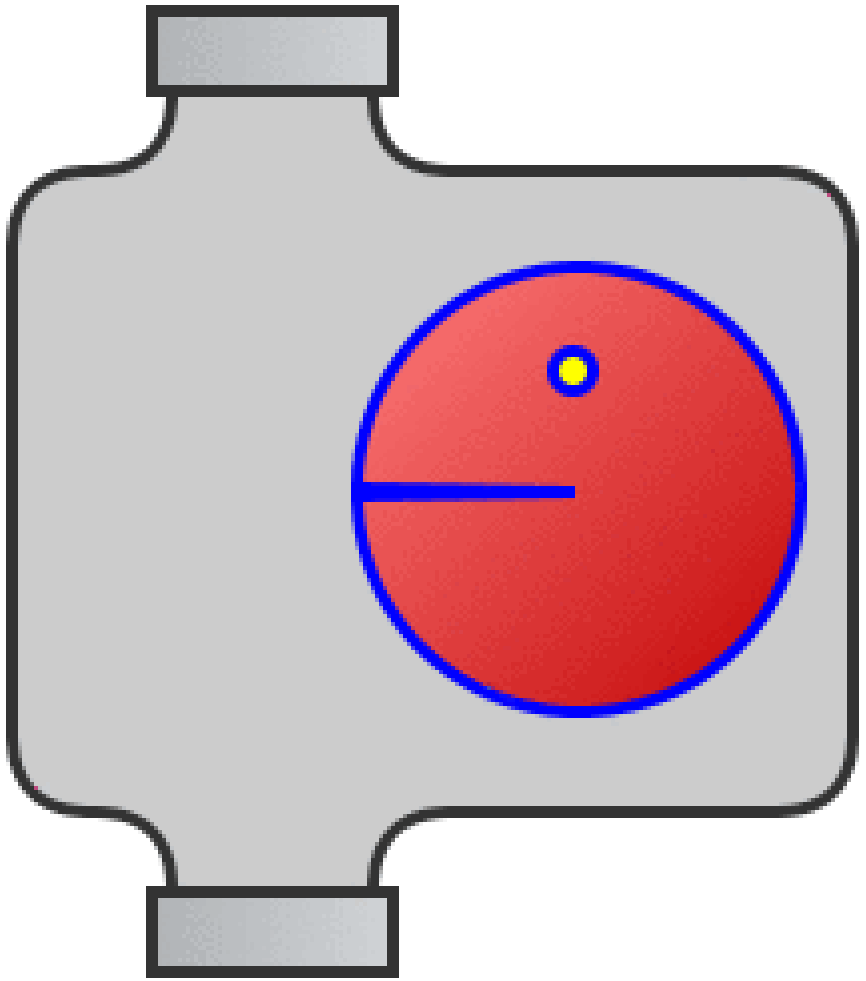

Ball valve
Y filter
Pressure reducer
Check valve
Water meter
Collector
Upstream
Downstream



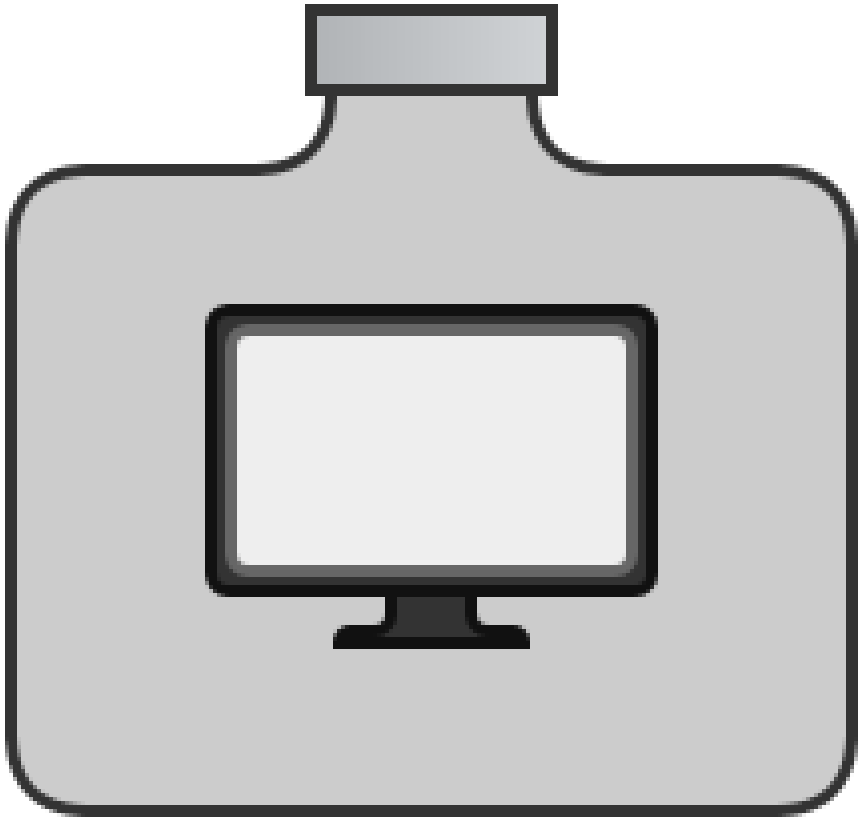
```
source.stream()
```



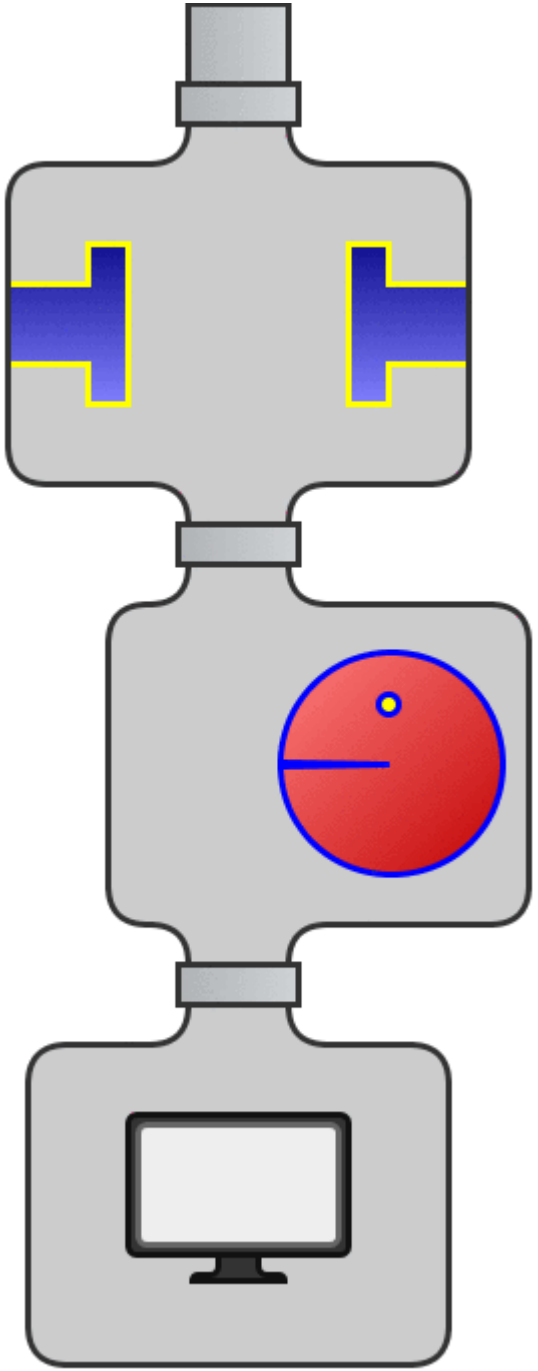
```
.map(x -> x.squash())
```



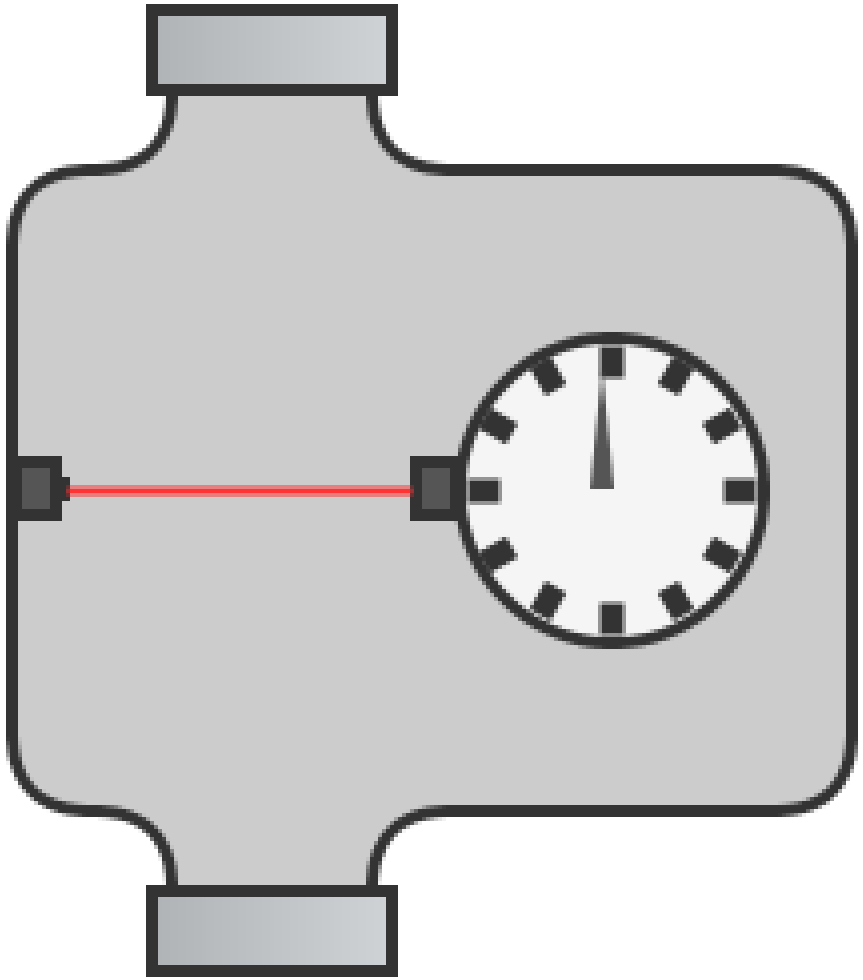
```
.filter(x -> x.getColor() != YELLOW)
```



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

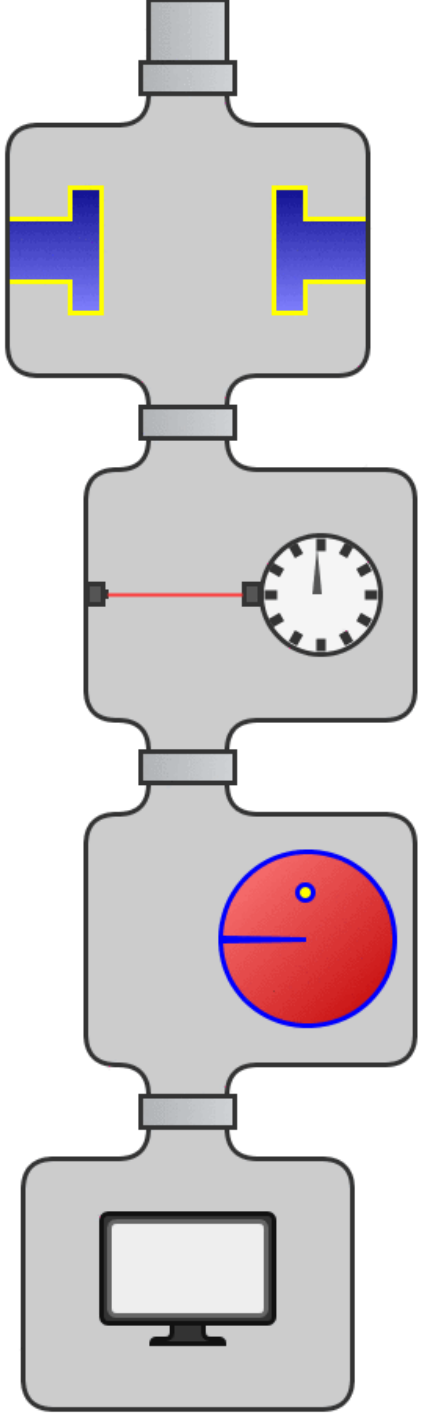


```
source.stream()
   .map(x -> x.squash())
   .filter(x -> x.getColor() != YELLOW)
   .forEach(System.out::println);
```



```
AtomicLong cnt = new AtomicLong();

    .peek(x -> cnt.incrementAndGet())
```



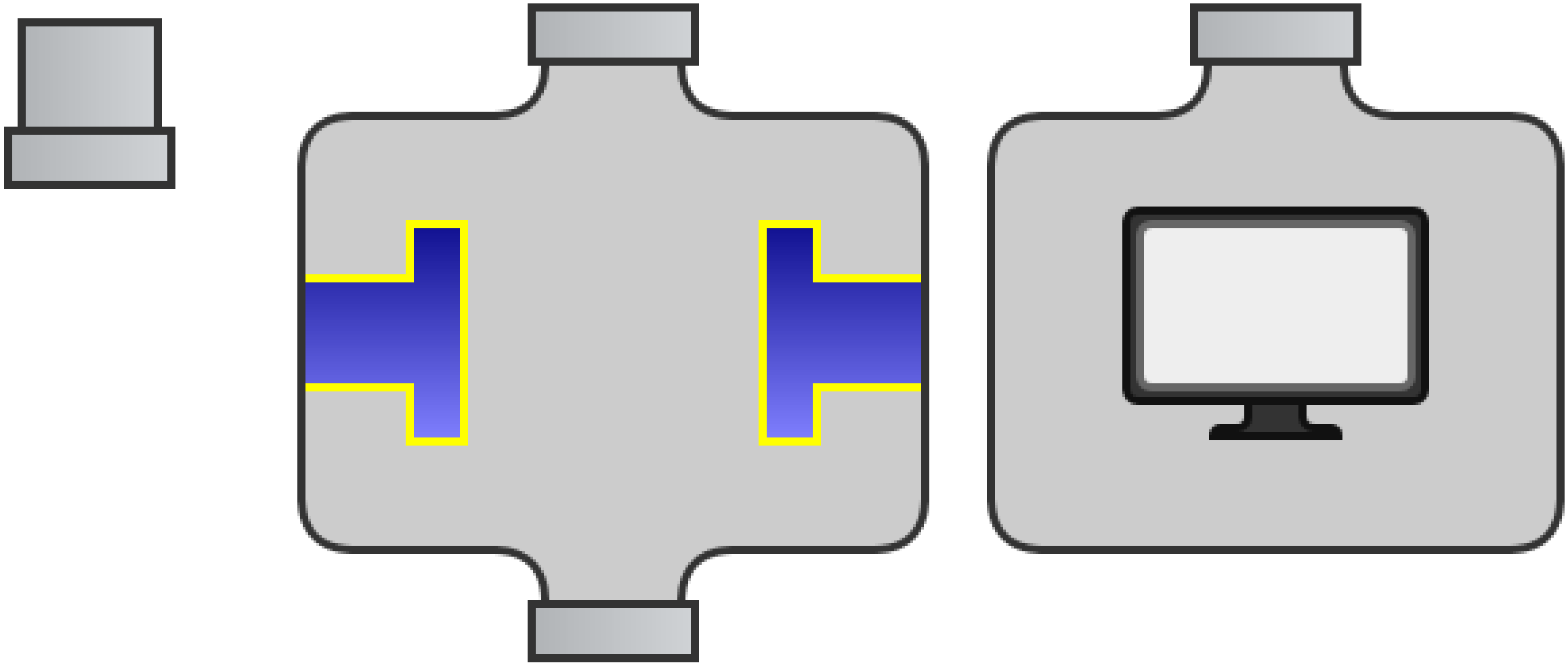
```
AtomicLong cnt = new AtomicLong();

source.stream()
   .map(x -> x.squash())
   .peek(x -> cnt.incrementAndGet())
   .filter(x -> x.getColor() != YELLOW)
   .forEach(System.out::println);
```

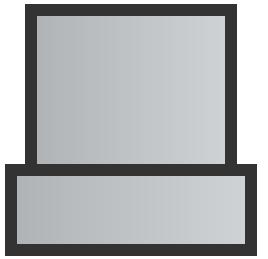
Sources

Intermediate operations

Terminal operations (sinks)



# Sources



## Problem #1:

Make the source of child nodes for a given XML DOM node.

## Problem #1:

Make the source of child nodes for a given XML DOM node.

```
public static Stream<Node> children(Node parent) {
  NodeList nodeList = parent.getChildNodes();
  return IntStream.range(0, nodeList.getLength())
      .mapToObj(nodeList::item);
}
```

## Problem #1:

Make the source of child nodes for a given XML DOM node.

```
public static Stream<Node> children(Node parent) {
  NodeList nodeList = parent.getChildNodes();
  return IntStream.range(0, nodeList.getLength())
      .mapToObj(nodeList::item);
}
```

Random.ints()
Random.longs()
String.chars()
Files.lines()
Files.list()

LocalDate.datesUntil()
Process.descendants()
ProcessHandle.allProcesses()
NetworkInterface.inetAddresses()
NetworkInterface.subInterfaces()

```java
package javax.swing;

public interface ListModel<E>
{
  /**
   * Returns the length of the list.
   * @return the length of the list
   */
  int getSize();

  /**
   * Returns the value at the specified index.
   * @param index the requested index
   * @return the value at <code>index</code>
   */
  E getElementAt(int index);

  …
}
```

## Problem #2:

Make the source of list elements, along with indices.

```
List.of("Good", "Old", "Stream", "API")
```

```
(0, "Good"), (1, "Old"), (2, "Stream"), (3, "API")
```

## Problem #2:

Make the source of list elements, along with indices.

```
public class IndexedValue<T> {
  public final int index;
  public final T value;

  public IndexedValue(int index, T value) {
    this.index = index;
    this.value = value;
  }
}
```

## Problem #2:

Make the source of list elements, along with indices.

```
public class IndexedValue<T> {
  public final int index;
  public final T value;

  public IndexedValue(int index, T value) {
    this.index = index;
    this.value = value;
  }
}
```

```
public record IndexedValue<T>(int index, T value) {}
```

## Problem #2:

Make the source of list elements, along with indices.

```
public record IndexedValue<T>(int index, T value) {}

public static <T> Stream<IndexedValue<T>> withIndices(List<T> list) {
  return IntStream.range(0, list.size())
      .mapToObj(idx -> new IndexedValue<>(idx, list.get(idx)));
}
```

## Problem #3:

Zip two lists together.

```
List.of("JDK 1.0", "J2SE 1.2", "J2SE 5.0", "Java SE 8", "Java SE 11", "Java SE 14");
List.of(1996, 1998, 2004, 2014, 2018, 2020);
```

```
"JDK 1.0 was released in 1996"
"J2SE 1.2 was released in 1998"
"J2SE 5.0 was released in 2004"
"Java SE 8 was released in 2014"
"Java SE 11 was released in 2018"
"Java SE 14 was released in 2020"
```

## Problem #3:

Zip two lists together.

```
List<String> list1 = List.of("JDK 1.0", "J2SE 1.2", "J2SE 5.0",
                             "Java SE 8", "Java SE 11", "Java SE 14");
List<Integer> list2 = List.of(1996, 1998, 2004, 2014, 2018, 2020);

zip(list1, list2, (jdk, year) -> jdk + " was released in " + year)
    .forEach(System.out::println);
```

## Problem #3:

Zip two lists together.

```
List<String> list1 = List.of("JDK 1.0", "J2SE 1.2", "J2SE 5.0",
                             "Java SE 8", "Java SE 11", "Java SE 14");
List<Integer> list2 = List.of(1996, 1998, 2004, 2014, 2018, 2020);

zip(list1, list2, Pair::new)
    .map(pair -> pair.first() + " was released in " + pair.second())
    .forEach(System.out::println);
```

## Problem #3:

Zip two lists together.

```
public static <T1, T2, R> Stream<R> zip(
    List<T1> list1, List<T2> list2,
    BiFunction<? super T1, ? super T2, ? extends R> mapper) {
  …
}
```

## Problem #3:

Zip two lists together.

```
public static <T1, T2, R> Stream<R> zip(
    List<T1> list1, List<T2> list2,
    BiFunction<? super T1, ? super T2, ? extends R> mapper) {
  int size = list1.size();
  if (list2.size() != size) {
    throw new IllegalArgumentException("Different list sizes");
  }
  …
}
```

## Problem #3:

Zip two lists together.

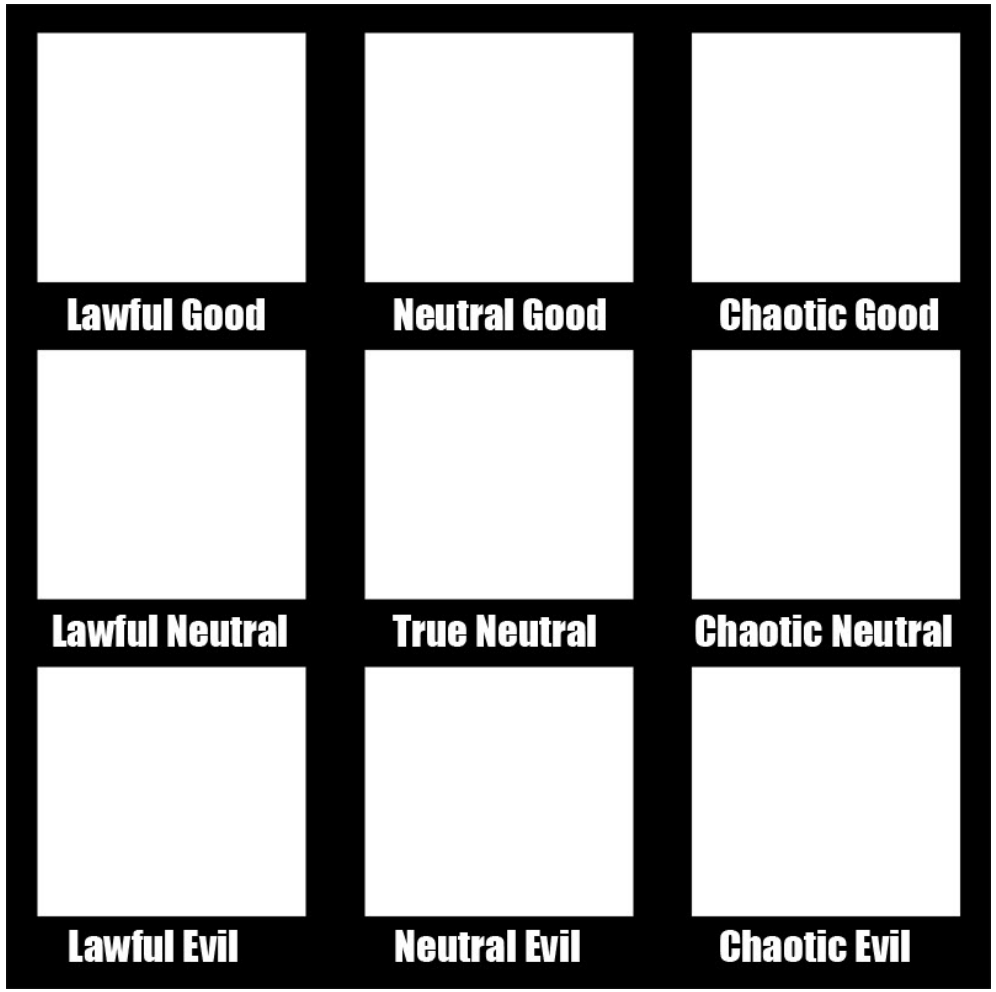
```
public static <T1, T2, R> Stream<R> zip(
    List<T1> list1, List<T2> list2,
    BiFunction<? super T1, ? super T2, ? extends R> mapper) {
  int size = list1.size();
  if (list2.size() != size) {
    throw new IllegalArgumentException("Different list sizes");
  }
  return IntStream.range(0, size)
          .mapToObj(idx -> mapper.apply(list1.get(idx), list2.get(idx)));
}
```

## Problem #4:

Make the source that generates Cartesian product of lists of strings.

## Problem #4:

Make the source that generates Cartesian product of lists of strings.

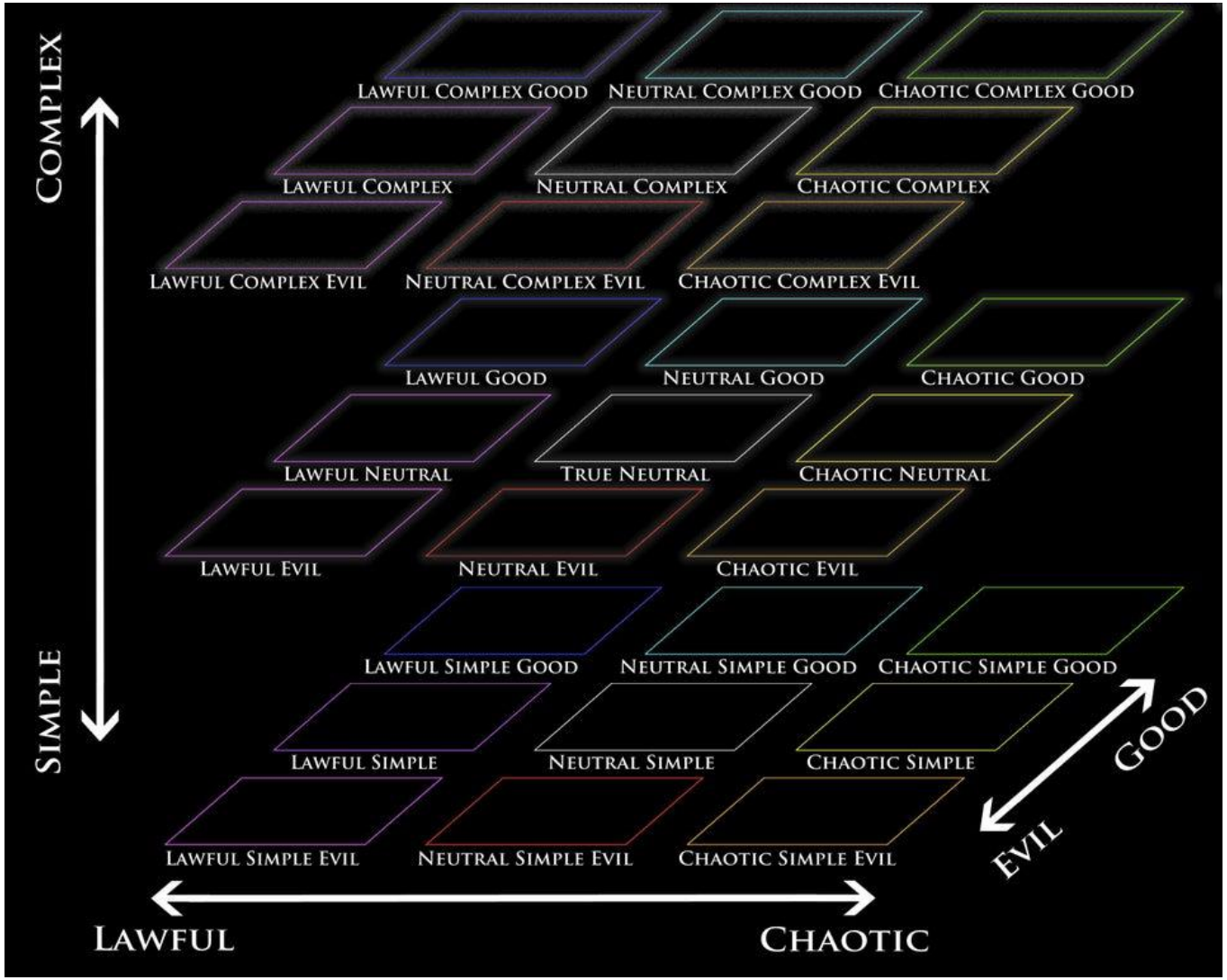



```
List<List<String>> input = List.of(
  List.of("Lawful ", "Neutral ", "Chaotic "),
  List.of("Good", "Neutral", "Evil")
);
```

```
[Lawful Good, Lawful Neutral, Lawful Evil,
Neutral Good, Neutral Neutral, Neutral Evil,
Chaotic Good, Chaotic Neutral, Chaotic Evil]
```

# Problem #4:

Make the source that generates Cartesian product of lists of strings.




Source:
https://www.reddit.com/r/AlignmentCharts/comments/ca2zvz/3x3x3_template_inspired_by_that_nationstates/

## Problem #4:

Make the source that generates Cartesian product of lists of strings.

```
List<List<String>> input = List.of(
    List.of("Lawful ", "Neutral ", "Chaotic "),
    List.of("Simple ", "", "Complex "),
    List.of("Good", "Neutral", "Evil")
);
```

```
[Lawful Simple Good, Lawful Simple Neutral, Lawful Simple Evil,
Lawful Good, Lawful Neutral, Lawful Evil,
Lawful Complex Good, Lawful Complex Neutral, Lawful Complex Evil,
Neutral Simple Good, Neutral Simple Neutral, Neutral Simple Evil,
Neutral Good, Neutral Neutral, Neutral Evil,
Neutral Complex Good, Neutral Complex Neutral, Neutral Complex Evil,
Chaotic Simple Good, Chaotic Simple Neutral, Chaotic Simple Evil,
Chaotic Good, Chaotic Neutral, Chaotic Evil,
Chaotic Complex Good, Chaotic Complex Neutral, Chaotic Complex Evil]
```

## Problem #4:

Make the source that generates Cartesian product of lists of strings.

```
List<List<String>> input = List.of(
    List.of("Lawful ", "Neutral ", "Chaotic "),
    List.of("Simple ", "", "Complex "),
    List.of("Good", "Neutral", "Evil")
);
Stream<String> stream =
    input.get(0).stream().flatMap(a ->
    input.get(1).stream().flatMap(b ->
    input.get(2).stream().map(c -> a + b + c)));
stream.forEach(System.out::println);
>> Lawful Simple Good
>> Lawful Simple Neutral
>> Lawful Simple Evil
>> Lawful Good
…
```

## Problem #4:

Make the source that generates Cartesian product of lists of strings.

```
Stream<String> s1 = Stream.of("Lawful ", "Neutral ", "Chaotic ");
Stream<String> s2 = Stream.of("Good", "Neutral", "Evil");
s1.flatMap(a -> s2.map(b -> a + b)).forEach(System.out::println);
```

## Problem #4:

Make the source that generates Cartesian product of lists of strings.

```
Stream<String> s1 = Stream.of("Lawful ", "Neutral ", "Chaotic ");
Stream<String> s2 = Stream.of("Good", "Neutral", "Evil");
s1.flatMap(a -> s2.map(b -> a + b)).forEach(System.out::println);
```

```
Lawful Good
Lawful Neutral
Lawful Evil
Exception in thread "main" java.lang.IllegalStateException: stream has
already been operated upon or closed
```

## Problem #4:

Make the source that generates Cartesian product of lists of strings.

```
Supplier<Stream<String>> s1 =
    () -> Stream.of("Lawful ", "Neutral ", "Chaotic ");
Supplier<Stream<String>> s2 =
    () -> Stream.of("Good", "Neutral", "Evil");

s1.get().flatMap(a -> s2.get().map(b -> a + b))
        .forEach(System.out::println);
```

## Problem #4:

Make the source that generates Cartesian product of lists of strings.

```
Supplier<Stream<String>> s1 =
    () -> Stream.of("Lawful ", "Neutral ", "Chaotic ");
Supplier<Stream<String>> s2 =
    () -> Stream.of("Good", "Neutral", "Evil");

Supplier<Stream<String>> reduced =
    () -> s1.get().flatMap(a -> s2.get().map(b -> a + b));
reduced.get().forEach(System.out::println);
```

## Problem #4:

Make the source that generates Cartesian product of lists of strings.

```
List<List<String>> input = List.of(
    List.of("Lawful ", "Neutral ", "Chaotic "),
    List.of("Simple ", "", "Complex "),
    List.of("Good", "Neutral", "Evil")
);

Stream<Supplier<Stream<String>>> streamOfStreamSuppliers = …


Supplier<Stream<String>> s = streamOfStreamSuppliers
    .reduce((s1, s2) ->
            () -> s1.get().flatMap(a -> s2.get().map(b -> a + b)))
    .orElse(() -> Stream.of(""));
```

## Problem #4:

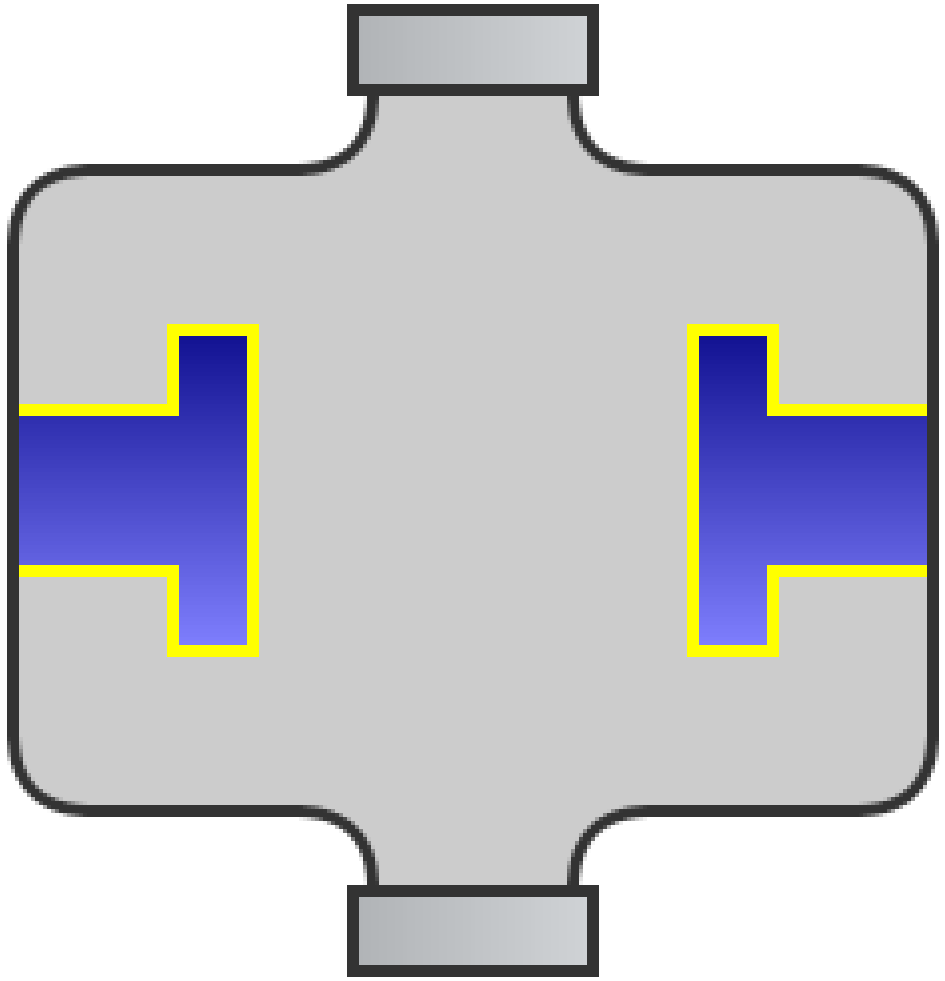
Make the source that generates Cartesian product of lists of strings.

```
List<List<String>> input = List.of(
    List.of("Lawful ", "Neutral ", "Chaotic "),
    List.of("Simple ", "", "Complex "),
    List.of("Good", "Neutral", "Evil")
);
```

```
Stream<Supplier<Stream<String>>> streamOfStreamSuppliers = input.stream()
    .map(list -> list::stream);
```

```
Supplier<Stream<String>> s = streamOfStreamSuppliers
    .reduce((s1, s2) ->
            () -> s1.get().flatMap(a -> s2.get().map(b -> a + b)))
    .orElse(() -> Stream.of(""));
```

# Intermediate operations



## Problem #5:

Leave only elements of given type in the stream

```
IntStream.range(0, nodeList.getLength())
    .mapToObj(nodeList::item)
    .???
```

## Problem #5:

Leave only elements of given type in the stream

```
IntStream.range(0, nodeList.getLength())
    .mapToObj(nodeList::item)
    .filter(node -> node instanceof Element);
```

```
IntStream.range(0, nodeList.getLength())
    .mapToObj(nodeList::item)
    .filter(Element.class::isInstance);
```

## Problem #5:

Leave only elements of given type in the stream

```
IntStream.range(0, nodeList.getLength())
    .mapToObj(nodeList::item)
    .filter(node -> node instanceof Element)
    .map(node -> (Element) node);

IntStream.range(0, nodeList.getLength())
    .mapToObj(nodeList::item)
    .filter(Element.class::isInstance)
    .map(Element.class::cast);
```

## Problem #5:

Leave only elements of given type in the stream

```
public static <T, R> Stream<R> select(Stream<T> stream, Class<R> clazz) {
  return stream.filter(clazz::isInstance).map(clazz::cast);
}
```

## Problem #5:

Leave only elements of given type in the stream

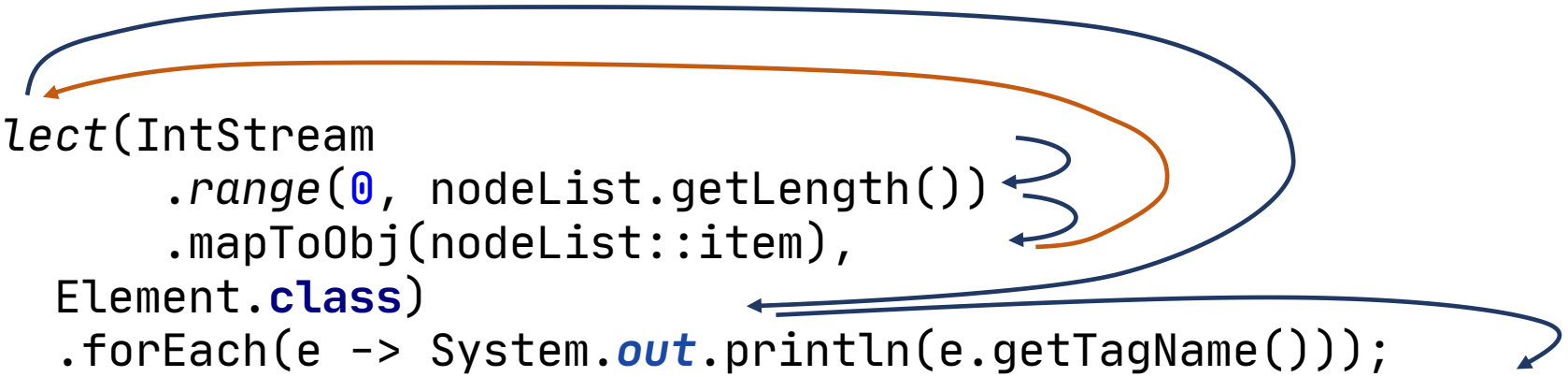
```
public static <T, R> Stream<R> select(Stream<T> stream, Class<R> clazz) {
  return stream.filter(clazz::isInstance).map(clazz::cast);
}
```

```
  select(IntStream
          .range(0, nodeList.getLength())
          .mapToObj(nodeList::item),
      Element.class)
      .forEach(e -> System.out.println(e.getTagName()));
```

# Problem #5:

## Leave only elements of given type in the stream

```
public static <T, R> Stream<R> select(Stream<T> stream, Class<R> clazz) {
  return stream.filter(clazz::isInstance).map(clazz::cast);
}
```



```
  select(IntStream
          .range(0, nodeList.getLength())
          .mapToObj(nodeList::item),
      Element.class)
      .forEach(e -> System.out.println(e.getTagName()));
```

# flatMap encapsulation

```
List<String> result = Stream.of("one", "two", "three")
    .flatMapToInt(word -> word.chars())
    .filter(ch -> ch != 'o')
    .peek(System.out::println)
    .mapToObj(ch -> "[" + ((char) ch) + "]")
    .collect(Collectors.toList());
```

# flatMap encapsulation

```
List<String> result = Stream.of("one", "two", "three")
    .flatMapToInt(word -> word.chars())
    .filter(ch -> ch != 'o')
    .peek(System.out::println)
    .mapToObj(ch -> "[" + ((char) ch) + "]")
    .collect(Collectors.toList());
```

```
List<String> result = Stream.of("one", "two", "three")
    .flatMap(word -> word.chars()
        .filter(ch -> ch != 'o')
        .peek(System.out::println)
        .mapToObj(ch -> "[" + ((char) ch) + "]"))
    .collect(Collectors.toList());
```

# flatMap encapsulation

```
static Function<String, Stream<String>> process() {
  return word -> word.chars()
      .filter(ch -> ch != 'o')
      .peek(System.out::println)
      .mapToObj(ch -> "[" + ((char) ch) + "]");
}

List<String> result = Stream.of("one", "two", "three")
      .flatMap(process())
      .collect(Collectors.toList());
```

## Problem #5:

Leave only elements of given type in the stream

```
public static <T, R> Function<T, Stream<R>> select(Class<R> clazz) {
  return e -> clazz.isInstance(e) ? Stream.of(clazz.cast(e)) : null;
}

IntStream.range(0, nodeList.getLength())
    .mapToObj(nodeList::item)
    .flatMap(select(Element.class));
```

## Problem #5:

Leave only elements of given type in the stream

```
// with one.util:streamex
IntStreamEx.range(nodeList.getLength())
    .mapToObj(nodeList::item)
    .select(Element.class)
    .forEach(e -> System.out.println(e.getTagName()));
```

## Problem #6:

Leave only those values that repeat at least N times.

```
List<String> langs = List.of("Java", "Scala", "Kotlin", "Kotlin", "Ceylon",
  "Groovy", "Groovy", "Kotlin", "Ceylon", "Clojure", "Scala", "Scala", "Groovy");
```

```
[Scala, Groovy, Kotlin]
```

## Problem #6:

Leave only those values that repeat at least N times.

```
List<String> langs = List.of("Java", "Scala", "Kotlin", "Kotlin", "Ceylon",
  "Groovy", "Groovy", "Kotlin", "Ceylon", "Clojure", "Scala", "Scala", "Groovy");

Map<String, Long> counts = langs.stream()
    .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()));
counts.values().removeIf(cnt -> cnt < 3);
counts.keySet().forEach(System.out::println);
```

## Problem #6:

Leave only those values that repeat at least N times.

```
public static <T> Predicate<T> distinct(long atLeast) {
  Map<T, Long> map = new ConcurrentHashMap<>();
  return t -> map.merge(t, 1L, Long::sum) == atLeast;
}

List<String> langs = List.of("Java", "Scala", "Kotlin", "Kotlin", "Ceylon",
  "Groovy", "Groovy", "Kotlin", "Ceylon", "Clojure", "Scala", "Scala", "Groovy");

langs.stream().filter(distinct(3)).forEach(System.out::println);
>> Kotlin
>> Scala
>> Groovy
```

**filter**

`Stream<T> filter(Predicate<? super T> predicate)`

Returns a stream consisting of the elements of this stream that match the given predicate.

This is an intermediate operation.

**Parameters:**

`predicate - a non-interfering, stateless predicate to apply to each element to determine if it should be included`

**Returns:**

`the new stream`

https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

## Problem #7:

Process adjacent overlapping pairs from the stream.

```
List<String> input = List.of("lion", "fox", "hare", "carrot");
```

```
+
```

```
(a, b) -> a + " eats " + b
```

```
lion eats fox
fox eats hare
hare eats carrot
```

## Problem #7:

Process adjacent overlapping pairs from the stream.

```
static <T, R> Function<T, Stream<R>> pairMap(BiFunction<T, T, R> mapper) {
  return new Function<>() {
    T previous = null;
    boolean hasPrevious;

    public Stream<R> apply(T t) {
      Stream<R> result;
      if (!hasPrevious) {
        hasPrevious = true;
        result = Stream.empty();
      } else {
        result = Stream.of(mapper.apply(previous, t));
      }
      previous = t;
      return result;
    }
  };
}
```

## Problem #7:

Process adjacent overlapping pairs from the stream.

```
List<String> input = List.of("lion", "fox", "hare", "carrot");

input.stream().flatMap(pairMap((a, b) -> a + " eats " + b))
              .forEach(System.out::println);
lion eats fox
fox eats hare
hare eats carrot
```

## Problem #7:

Process adjacent overlapping pairs from the stream.

```
List<String> input = List.of("lion", "fox", "hare", "carrot");

input.stream().flatMap(pairMap((a, b) -> a + " eats " + b))
              .forEach(System.out::println);
lion eats fox
fox eats hare
hare eats carrot

List<String> input = List.of("lion", "fox", "hare", "carrot");

input.parallelStream().flatMap(pairMap((a, b) -> a + " eats " + b))
    .forEach(System.out::println);
hare eats carrot
hare eats lion
hare eats fox
```

`public interface Spliterator<T>`

An object for traversing and partitioning elements of a source. The source of elements covered by a Spliterator could be, for example, an array, a `Collection`, an IO channel, or a generator function.

A Spliterator may traverse elements individually (`tryAdvance()`) or sequentially in bulk (`forEachRemaining()`).

A Spliterator may also partition off some of its elements (using `trySplit()`) as another Spliterator, to be used in possibly-parallel operations. Operations using a Spliterator that cannot split, or does so in a highly imbalanced or inefficient manner, are unlikely to benefit from parallelism. Traversal and splitting exhaust elements; each Spliterator is useful for only a single bulk computation.

A Spliterator also reports a set of `characteristics()` of its structure, source, and elements from among `ORDERED`, `DISTINCT`, `SORTED`, `SIZED`, `NONNULL`, `IMMUTABLE`, `CONCURRENT`, and `SUBSIZED`. These may be employed by Spliterator clients to control, specialize or simplify computation. For example, a Spliterator for a `Collection` would report `SIZED`, a Spliterator for a `Set` would report `DISTINCT`, and a Spliterator for a `SortedSet` would also report `SORTED`. Characteristics are reported as a simple unioned bit set. Some characteristics additionally constrain method behavior; for example if `ORDERED`, traversal methods must conform to their documented ordering. New characteristics may be defined in the future, so implementors should not assign meanings to unlisted values.

https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html

## Problem #7:

Process adjacent overlapping pairs from the stream.

```
static <T, R> Stream<R> pairMap(Stream<T> input, BiFunction<T, T, R> mapper) {
  return StreamSupport.stream(
      new PairMapSpliterator<>(input.spliterator(), mapper), input.isParallel())
      .onClose(input::close);
}
```

## Problem #7:

Process adjacent overlapping pairs from the stream.

```
private static class PairMapSpliterator<T, R> extends
    Spliterators.AbstractSpliterator<R> {
  private final Spliterator<T> spliterator;
  private final BiFunction<T, T, R> mapper;

  PairMapSpliterator(Spliterator<T> spliterator, BiFunction<T, T, R> mapper) {
    super(calcSize(spliterator),
          spliterator.characteristics() & (SIZED | ORDERED));
    this.spliterator = spliterator;
    this.mapper = mapper;
  }

  private static long calcSize(Spliterator<?> spliterator) { … }
}
```

## Problem #7:

Process adjacent overlapping pairs from the stream.

```
private static class PairMapSpliterator<T, R> extends
    Spliterators.AbstractSpliterator<R> {
  private final Spliterator<T> spliterator;
  private final BiFunction<T, T, R> mapper;

  PairMapSpliterator(Spliterator<T> spliterator, BiFunction<T, T, R> mapper) {
    super(calcSize(spliterator),
          spliterator.characteristics() & (SIZED | ORDERED));
    this.spliterator = spliterator;
    this.mapper = mapper;
  }

  private static long calcSize(Spliterator<?> spliterator) { … }
}
```

## Problem #7:

Process adjacent overlapping pairs from the stream.

```
private static class PairMapSpliterator<T, R> extends
    Spliterators.AbstractSpliterator<R> {
  …

  PairMapSpliterator(Spliterator<T> spliterator, BiFunction<T, T, R> mapper) {
    super(calcSize(spliterator),
          spliterator.characteristics() & (SIZED | ORDERED));
    …
  }

  private static long calcSize(Spliterator<?> spliterator) {
    long size = spliterator.estimateSize();
    return size <= 0 ? 0 : size == Long.MAX_VALUE ? Long.MAX_VALUE : size - 1;
  }
}
```

## Problem #7:

Process adjacent overlapping pairs from the stream.

```
private static class PairMapSpliterator<T, R> extends
    Spliterators.AbstractSpliterator<R> {
  …

  PairMapSpliterator(Spliterator<T> spliterator, BiFunction<T, T, R> mapper) {
    super(calcSize(spliterator),
          spliterator.characteristics() & (SIZED | ORDERED));
    …
  }

  private static long calcSize(Spliterator<?> spliterator) {
    long size = spliterator.estimateSize();
    return size <= 0 ? 0 : size == Long.MAX_VALUE ? Long.MAX_VALUE : size - 1;
  }
}
```

## Problem #7:

Process adjacent overlapping pairs from the stream.

```
private static class PairMapSpliterator<T, R> extends
    Spliterators.AbstractSpliterator<R> {
  …

  PairMapSpliterator(Spliterator<T> spliterator, BiFunction<T, T, R> mapper) {
    super(calcSize(spliterator),
          spliterator.characteristics() & (SIZED | ORDERED));
    …                                NONNULL    DISTINCT    SORTED
  }

  private static long calcSize(Spliterator<?> spliterator) {
    long size = spliterator.estimateSize();
    return size <= 0 ? 0 : size == Long.MAX_VALUE ? Long.MAX_VALUE : size - 1;
  }
}
```

## Problem #7:

Process adjacent overlapping pairs from the stream.

```
private static class PairMapSpliterator<T, R> extends AbstractSpliterator<R> {
  private final Spliterator<T> spliterator;
  private final BiFunction<T, T, R> mapper;
  private T prev;
  private boolean hasPrevious;

  @Override
  public boolean tryAdvance(Consumer<? super R> action) {
    if (!hasPrevious) {
      if (!spliterator.tryAdvance(first -> prev = first)) return false;
      hasPrevious = true;
    }
    return spliterator.tryAdvance(
        next -> action.accept(mapper.apply(prev, prev = next)));
  }
}
```

## Problem #7:

Process adjacent overlapping pairs from the stream.

```
private static class PairMapSpliterator<T, R> extends AbstractSpliterator<R> {
  private final Spliterator<T> spliterator;
  private final BiFunction<T, T, R> mapper;
  private T prev;
  private boolean hasPrevious;

  @Override public void forEachRemaining(Consumer<? super R> action) {
    spliterator.forEachRemaining(next -> {
      if (!hasPrevious) {
        hasPrevious = true;
      } else {
        action.accept(mapper.apply(prev, next));
      }
      prev = next;
    });
  }
}
```

# Terminal operations

# Collectors

```
public interface Collector<T, A, R> {
    Supplier<A> supplier();

    BiConsumer<A, T> accumulator();

    BinaryOperator<A> combiner();

    Function<A, R> finisher();

    Set<Characteristics> characteristics();
}
```

## Problem #8:

Transform List<Map<K, V>> into Map<K, List<V>>

Input:

```
[{a=1, b=2},
 {a=3, d=4, e=5},
 {a=5, b=6, e=7}
]
```

Output:

```
{a=[1, 3, 5],
 b=[2, 6],
 d=[4],
 e=[5, 7]
}
```

## Problem #8:

Transform List<Map<K, V>> into Map<K, List<V>>

```
input.stream()
    .flatMap(map -> map.entrySet().stream())
    .collect(Collectors.groupingBy(Map.Entry::getKey,
                Collectors.mapping(Map.Entry::getValue,
                    Collectors.toList())));
```

## Problem #8:

Transform List<Map<K, V>> into Map<K, List<V>>

```
input.stream()
    .flatMap(map -> map.entrySet().stream())
    .collect(Collectors.groupingBy(Map.Entry::getKey,
                Collectors.mapping(Map.Entry::getValue,
                    Collectors.toList())));

StreamEx.of(input)
    .flatMapToEntry(m -> m)
    .grouping();
```

## Problem #8:

Transform List<Map<K, V>> into Map<K, List<V>>

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

input.stream()
    .flatMap(map -> map.entrySet().stream())
    .collect(groupingBy(Map.Entry::getKey,
        mapping(Map.Entry::getValue,
            toList())));
```

## Problem #9:

Having a list of employees, find all the departments where total employees salary exceeds $1,000,000; sort the result by salary in decreasing order.

```
interface Employee {
  Department getDepartment();
  long getSalary();
}
```

## Problem #9:

Having a list of employees, find all the departments where total employees salary exceeds $1,000,000; sort the result by salary in decreasing order.

```
Map<Department, Long> deptSalaries = employees.stream().collect(
    groupingBy(Employee::getDepartment,
        summingLong(Employee::getSalary)));
```

```
deptSalaries.entrySet().stream()
    .filter(e -> e.getValue() > 1_000_000)
    .sorted(Map.Entry.comparingByValue(Comparator.reverseOrder()))
    .collect(toMap(Map.Entry::getKey, Map.Entry::getValue,
        (a, b) -> a, LinkedHashMap::new));
```

## Problem #9:

Having a list of employees, find all the departments where total employees salary exceeds $1,000,000; sort the result by salary in decreasing order.

```
Map<Department, Long> deptSalaries = employees.stream().collect(
    groupingBy(Employee::getDepartment,
        summingLong(Employee::getSalary)));
```

```
deptSalaries.entrySet().stream()
    .filter(e -> e.getValue() > 1_000_000)
    .sorted(Map.Entry.comparingByValue(Comparator.reverseOrder()))
    .collect(toMap(Map.Entry::getKey, Map.Entry::getValue,
        (a, b) -> a, LinkedHashMap::new));
```

## Problem #9:

Having a list of employees, find all the departments where total employees salary exceeds $1,000,000; sort the result by salary in decreasing order.

```
Map<Department, Long> deptSalaries = StreamEx.of(employees)
    .groupingBy(Employee::getDepartment,
        summingLong(Employee::getSalary));

EntryStream.of(deptSalaries)
    .filterValues(salary -> salary > 1_000_000)
    .reverseSorted(Map.Entry.comparingByValue())
    .toCustomMap(LinkedHashMap::new);
```

## Problem #9:

Having a list of employees, find all the departments where total employees salary exceeds $1,000,000; sort the result by salary in decreasing order.

```
Map<Department, Long> result = employees.stream().collect(
    collectingAndThen(groupingBy(Employee::getDepartment,
        summingLong(Employee::getSalary)),
        deptSalaries -> deptSalaries.entrySet().stream()
            .filter(e -> e.getValue() > 1_000_000)
            .sorted(Map.Entry.comparingByValue(Comparator.reverseOrder()))
            .collect(toMap(Map.Entry::getKey, Map.Entry::getValue,
                (a, b) -> a, LinkedHashMap::new))));
```

## Problem #9:

Having a list of employees, find all the departments where total employees salary exceeds $1,000,000; sort the result by salary in decreasing order.

```
Map<Department, Long> result = employees.stream().collect(
    collectingAndThen(groupingBy(Employee::getDepartment,
        summingLong(Employee::getSalary)),
        deptSalaries -> deptSalaries.entrySet().stream()
            .filter(e -> e.getValue() > 1_000_000)
            .sorted(Map.Entry.comparingByValue(Comparator.reverseOrder()))
            .collect(toMap(Map.Entry::getKey, Map.Entry::getValue,
                (a, b) -> a, LinkedHashMap::new))));
```

**Streamosis**

## Problem #10:

Find all the maximal elements using given comparator.

```
List<String> langs = List.of("Java", "Kotlin", "Scala", "Groovy",
                             "Ceylon", "JRuby");

List<String> longestLangs = langs.stream().collect(
                            maxAll(Comparator.comparing(String::length)));
>> [Kotlin, Groovy, Ceylon]
```

## Problem #10:

Find all the maximal elements using given comparator.

```
public static <T> Collector<T, ?, List<T>> maxAll(Comparator<T> cmp) {
  BiConsumer<List<T>, T> accumulator = (list, t) -> {
    if (!list.isEmpty()) {
      int c = cmp.compare(list.get(0), t);
      if (c > 0)
        return;
      if (c < 0)
        list.clear();
    }
    list.add(t);
  };
  BinaryOperator<List<T>> combiner = (l1, l2) -> {
    l2.forEach(t -> accumulator.accept(l1, t));
    return l1;
  };
  return Collector.of(ArrayList::new, accumulator, combiner);
}
```

## Problem #10:

Find all the maximal elements using given comparator.

```
public static <T> Collector<T, ?, List<T>> maxAll(Comparator<T> cmp) {
  BiConsumer<List<T>, T> accumulator = (list, t) -> {
    if (!list.isEmpty()) {
      int c = cmp.compare(list.get(0), t);
      if (c > 0)
        return;
      if (c < 0)
        list.clear();
    }
    list.add(t);
  };
  BinaryOperator<List<T>> combiner = (l1, l2) -> {
    l2.forEach(t -> accumulator.accept(l1, t));
    return l1;
  };
  return Collector.of(ArrayList::new, accumulator, combiner);
}
```

## Problem #10:

Find all the maximal elements using given comparator.

```
public static <T> Collector<T, ?, List<T>> maxAll(Comparator<T> cmp) {
  BiConsumer<List<T>, T> accumulator = (list, t) -> {
    if (!list.isEmpty()) {
      int c = cmp.compare(list.get(0), t);
      if (c > 0)
        return;
      if (c < 0)
        list.clear();
    }
    list.add(t);
  };
  BinaryOperator<List<T>> combiner = (l1, l2) -> {
    l2.forEach(t -> accumulator.accept(l1, t));
    return l1;
  };
  return Collector.of(ArrayList::new, accumulator, combiner);
}
```

Collector issues

1. Combiner associativity required
2. No short-circuit collectors

## Problem #11:

Check if all stream elements are unique (true or false).

```
List<String> langs = List.of("Java", "Kotlin", "Scala", "Groovy",
                             "Ceylon", "Java", "JRuby");

System.out.println(langs.stream().???);
>> false
```

## Problem #11:

Check if all stream elements are unique (true or false).

```
List<String> langs = List.of("Java", "Kotlin", "Scala", "Groovy",
                             "Ceylon", "Java", "JRuby");

System.out.println(langs.stream().allMatch(ConcurrentHashMap.newKeySet()::add));
```

## Problem #12:

Calculate the string hashCode using standard algorithm: $\sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}$

```
"JavaDay".hashCode();
>> -155243014
```

## Problem #12:

Calculate the string hashCode using standard algorithm: $\sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}$

```
"JavaDay".hashCode();
>> -155243014
"JavaDay".chars().reduce(0, (a, b) -> a * 31 + b);
>> -155243014
```

## Problem #12:

Calculate the string hashCode using standard algorithm: $\sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}$

```
"JavaDay".hashCode();
>> -155243014
"JavaDay".chars().reduce(0, (a, b) -> a * 31 + b);
>> -155243014
"JavaDay".chars().parallel().reduce(0, (a, b) -> a * 31 + b);
>> 266442
```

**reduce**

```
int reduce(int identity,
           IntBinaryOperator op)
```

Performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value. This is equivalent to:

```
int result = identity;
for (int element : this stream)
    result = accumulator.applyAsInt(result, element)
return result;
```

but is not constrained to execute sequentially.

The `identity` value must be an identity for the accumulator function. This means that for all `x`, `accumulator.apply(identity, x)` is equal to `x`. The `accumulator` function must be an associative function.

This is a terminal operation.

https://docs.oracle.com/javase/8/docs/api/java/util/stream/IntStream.html

## Problem #12:

Calculate the string hashCode using standard algorithm: $\sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}$

```
"JavaDay".hashCode();
>> -155243014
"JavaDay".chars().reduce(0, (a, b) -> a * 31 + b);
>> -155243014
"JavaDay".chars().parallel().reduce(0, (a, b) -> a * 31 + b);
>> 266442
```

```
(a * 31 + b) * 31 + c != a * 31 + (b * 31 + c)
```

## Problem #12:

Calculate the string hashCode using standard algorithm: $\sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}$

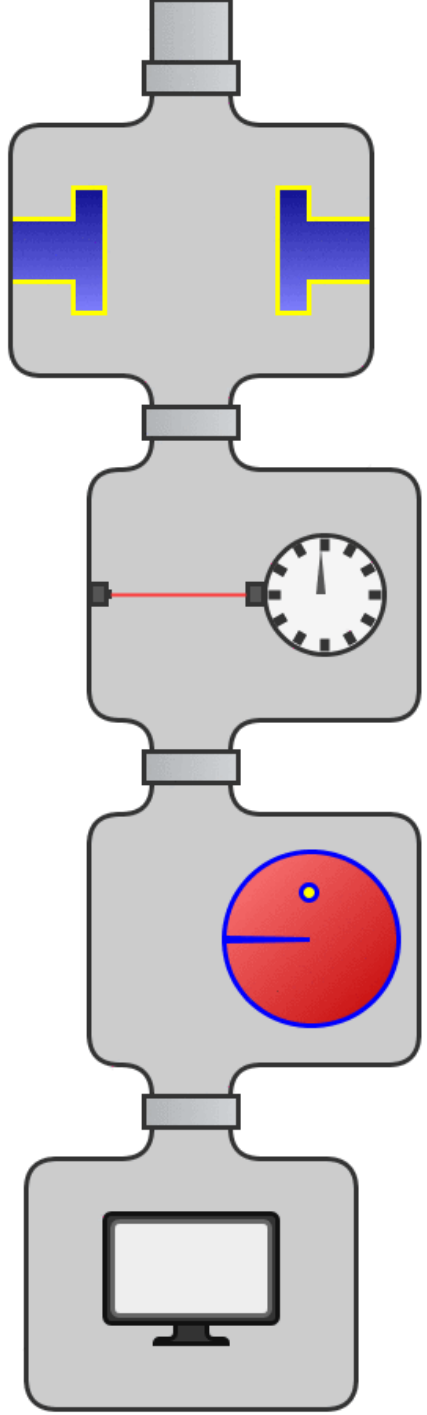
```
public static int foldLeft(IntStream input, int seed,
                           IntBinaryOperator op) {
  int[] box = { seed };
  input.forEachOrdered(t -> box[0] = op.applyAsInt(box[0], t));
  return box[0];
}

foldLeft("JavaDay".chars().parallel(), 0, (a, b) -> a * 31 + b);
```

## Problem #12:

Calculate the string hashCode using standard algorithm: $\sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}$

```
static int hashCode(String s) {
  int hash = 0;
  for (int i = 0; i < s.length(); i++) {
    hash = hash * 31 + s.charAt(i);
  }
  return hash;
}
```



# Thank you!

https://twitter.com/tagir_valeev
https://github.com/amaembo
https://github.com/amaembo/streamex
tagir.valeev@jetbrains.com