JAVA PROGRAMMING BASICS

Module 2: Java Object-oriented Programming

Training program

1. Classes and Instances
2. The Methods
3. The Constructors
4. Static Elements
5. Initialization sections
6. Package
7. Inheritance and Polymorphism
8. Abstract classes and Interfaces
9. String processing
10. Wrapper classes for primitive types
11. Exceptions and Assertions
12. Nested classes
13. Enums
14. Generics
15. Collections
16. Method overload resolution
17. Multithreads
18. Core Java classes
19. Object Oriented Design
20. **Functional Programming**

Module contents

1. Functional Programming in Java
   - Functional Interfaces
   - Lambda Expressions
   - Predicates
   - Functions
   - Operators
   - Consumers
   - Suppliers
   - Use in Traversing Objects
   - Use in Collections
   - Use in Comparing Objects
   - Use in Optionals
   - Use in Streams

Java Streams benefit

Task: To count cars with MPG (miles per gallon) > 50

Imperative style: "how it do" and external iteration

```
long count = 0;
for (Car car : cars) {
    if (car.mpg > 50)
        count++;
}
```

```
Iterator<Car> iterator = cars.iterator();
long count = 0;
while (iterator.hasNext()) {
    if (iterator.next().mpg > 50)
        count++;
}
```

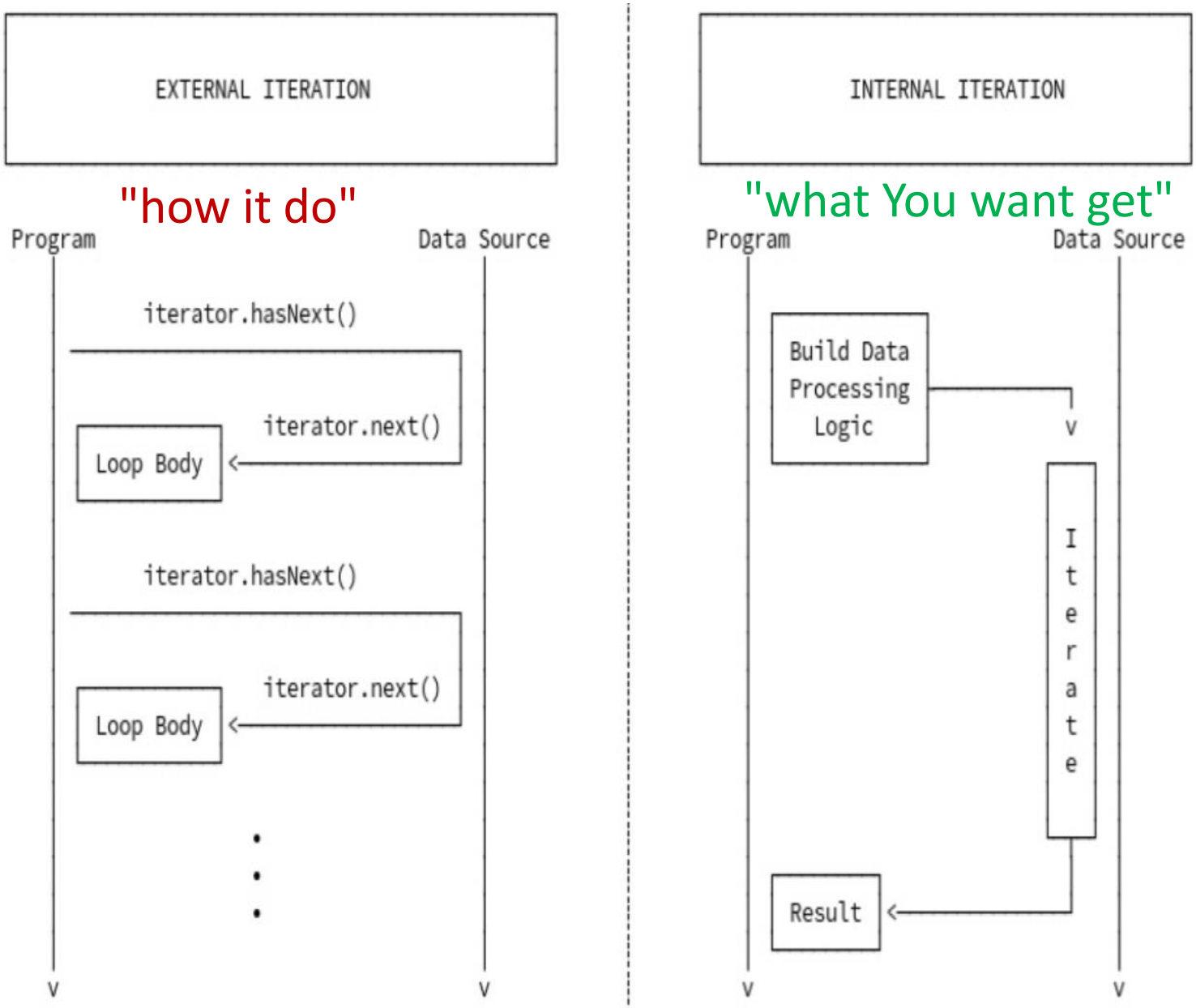
Declarative style: "what You want get" and internal iterating

```
long count = cars.stream()
        .filter(x -> x.mpg > 50)
        .count();
```

The Stream interface provides chainable operations that can be performed on a series of values.

since JDK 8

Java Streams



Java Streams

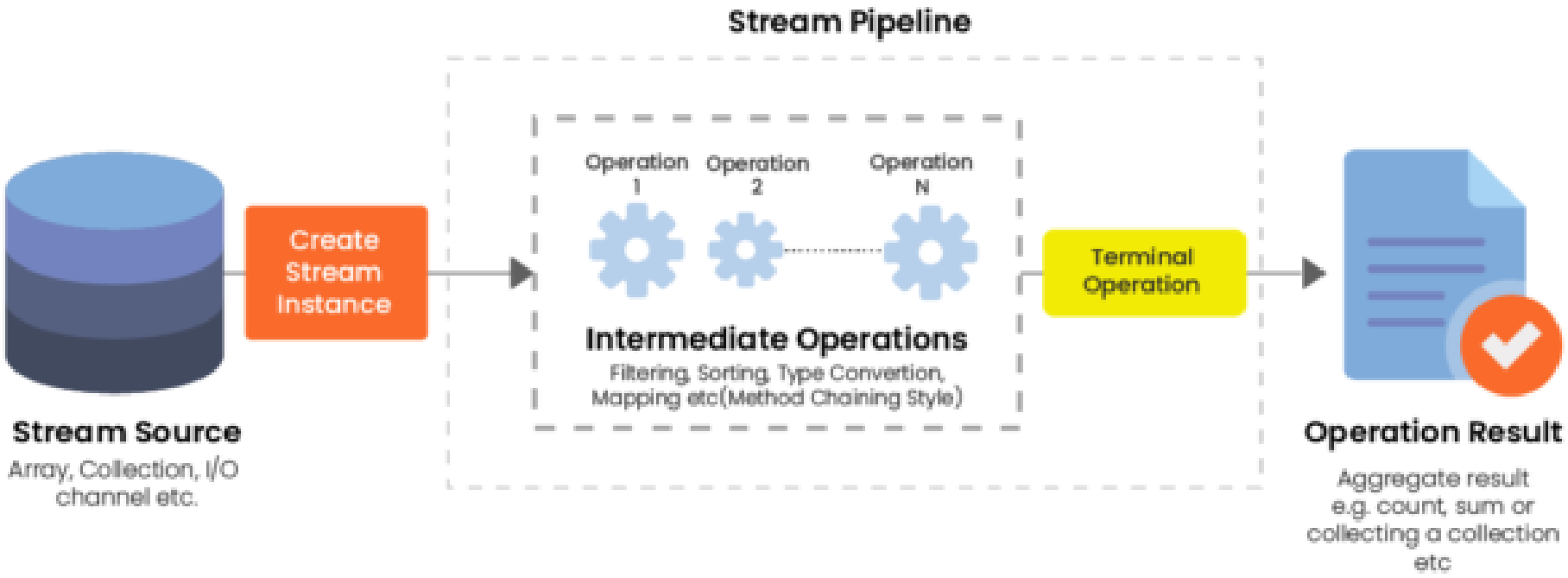
- A **stream** is a sequence of objects that supports various methods which can be *pipelined* to produce the desired result.
- An object that implements the Stream interface is like an Optional that *can contain several values instead of just one*.
- Since many of the methods in the Stream interface return streams, very powerful chains of streams can be created.

The features of Java stream are:

- A stream *is not a data structure* instead it takes input from the Collections, Arrays, I/O channels, etc
- Streams *don't change the original data structure*, they only provide the result as per the pipelined methods.
- Stream data can process sequential or parallel (use Spliterator).
- Each ***intermediate operation*** is *lazily executed* and returns a stream as a result, hence various *intermediate operations can be pipelined*.
- ***Terminal operation*** invoke marks the end of the stream and inits data processing by stream pipeline with result return.

Java 8 Streams should not be confused with Java I/O streams.

Java Streams



interface **java.util.stream.Stream<T>**

Operations:

- Intermediate
- Terminal

The streams can be used only once,
the stream is no longer valid after a terminal operation completes

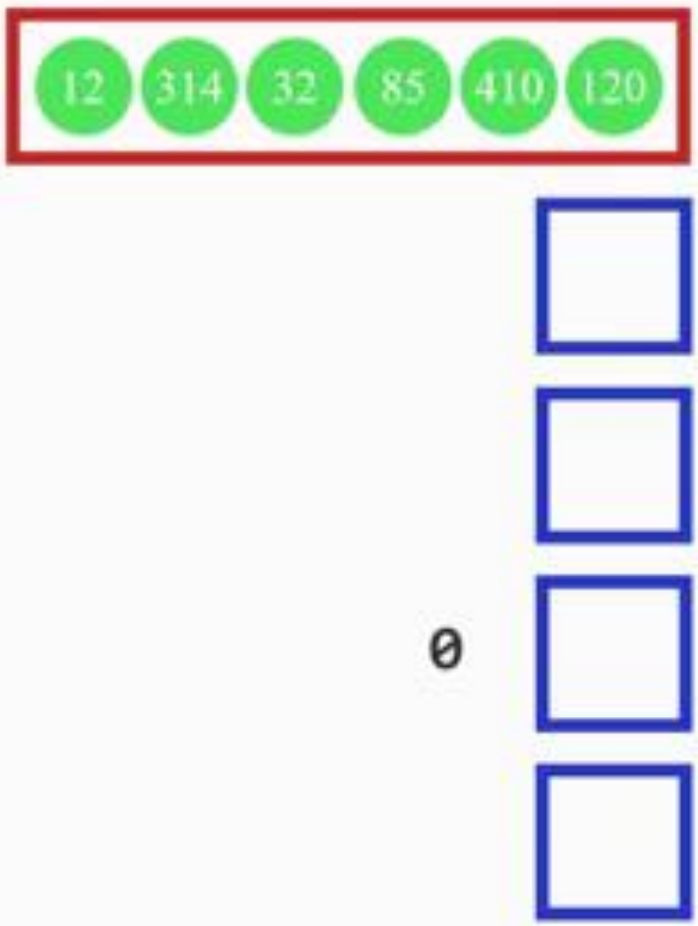
Java Stream types

When creating a stream from a data source, certain aspects to consider include whether the stream is:

- Sequential or parallel
- Ordered or unordered
- Finite or infinite
- Object or numeric

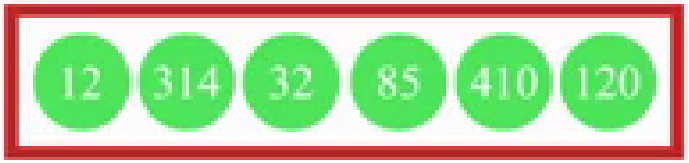
Interface **java.util.stream.Stream<T>** is not Functional Interface - the most of it methods are abstract.

Stream example




```
Stream
    .of(120, 410, 85, 32, 314, 12)
    .filter(x -> x < 300)
    .map(x -> x + 11)
    .limit(3)
    .forEach(System.out::print)
```

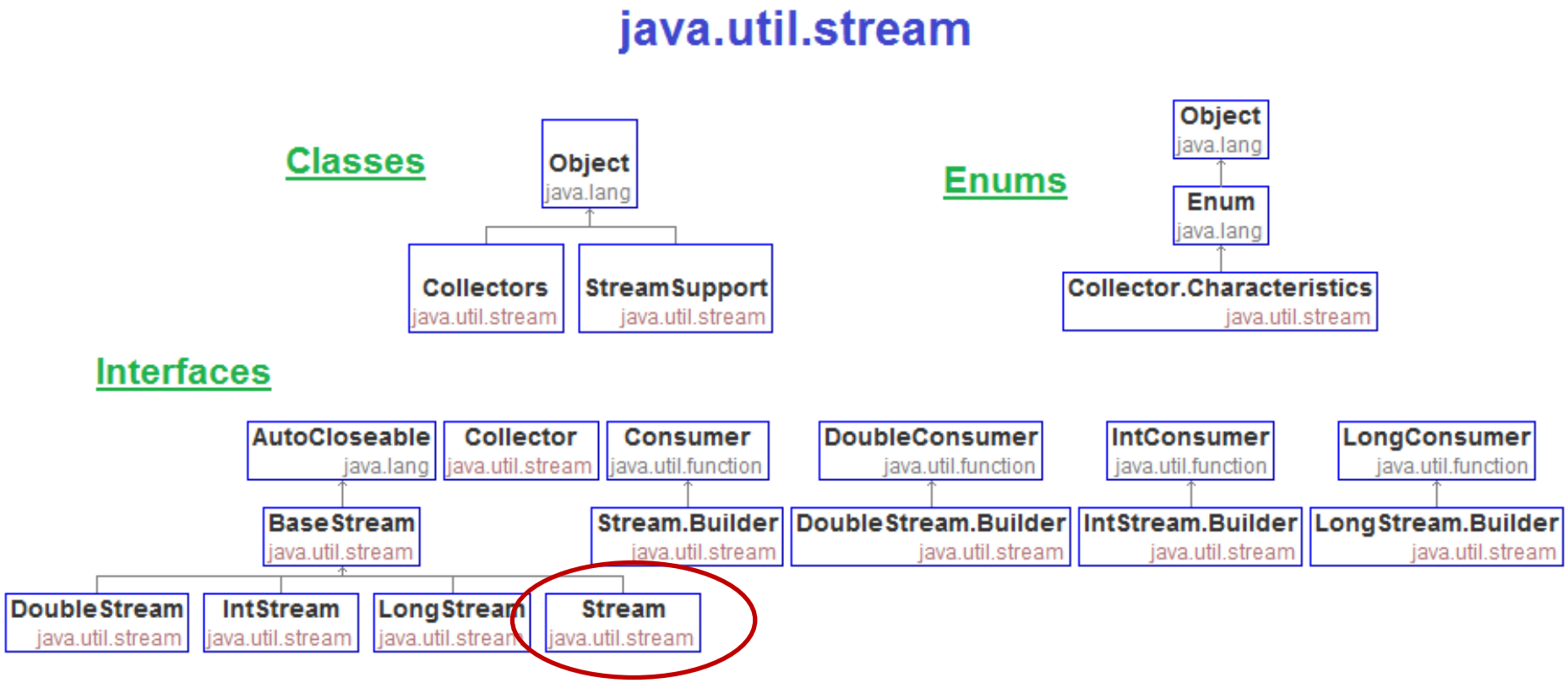
Stream example




```
Stream
      .of(120, 410, 85, 32, 314, 12)
```

- The intermediate operations do not run until the terminal operation runs.
- Processing occurs from the terminal operator to the source.

Java Stream API



Creating Stream source - elements generation

- A Stream is generic for type parameter T which is the type of its values.

```
public interface Stream<T> extends BaseStream<T, Stream<T>> {…}
```

- Stream<T> has methods for elements generation:

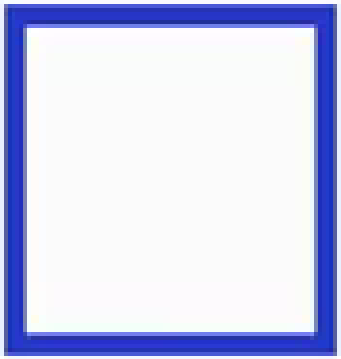
```
public static<T> Stream<T> empty()

public static<T> Stream<T> of(T... values)
public static<T> Stream<T> iterate(T seed,
                    Predicate<? super T> hasNext, UnaryOperator<T> next)
public static<T> Stream<T> generate(Supplier<? extends T> s)
public static<T> Builder<T> builder()

public static <T> Stream<T> concat(Stream<? extends T> a,
                                   Stream<? extends T> b)
```

- Stream<T> has void **forEach**(Consumer<? super T> action) method like Iterable<T>.

See streams\creation\StreamElementsGeneration

Generating Stream elements

```
iterate(2, (x) -> x + 6)
```

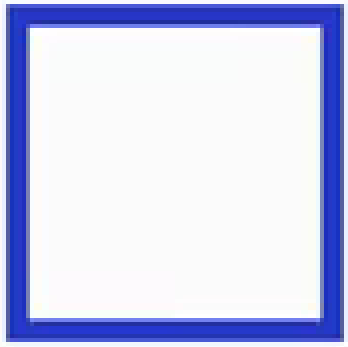


Stream<T> has:

```
public static<T> Stream<T> iterate(T seed,
                Predicate<? super T> hasNext, UnaryOperator<T> next
```

Generating Stream elements

```
generate(() -> 6)
```

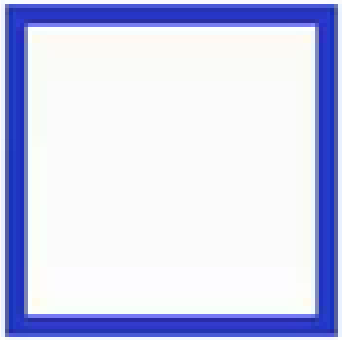


Stream<T> has:

public static<T> Stream<T> **generate**(Supplier<? extends T> s)

Generating Stream elements

limit(4)



Stream<T> has:

Stream<T> limit(long maxSize);

Creating Stream source - elements generation

- The public IntStream **ints()** method of java.util.Random creates random IntStream with elements in the range [Integer.MIN_VALUE, Integer.MAX_VALUE]
- The public IntStream **ints(long streamSize)** method of java.util.Random creates random IntStream with **streamSize** elements in the range [Integer.MIN_VALUE, Integer.MAX_VALUE]
- The public IntStream **ints(int randomNumberOrigin, int randomNumberBound)** method of java.util.Random creates random IntStream with elements in the range [randomNumberOrigin, randomNumberBound)
- The public IntStream **ints(long streamSize, int randomNumberOrigin, int randomNumberBound)** method of java.util.Random creates random IntStream with **streamSize** elements in the range [randomNumberOrigin, randomNumberBound]

See streams\creation\RandomNumStreamGeneration

Creating non-generic Stream source

- The Java API provides the **non-generic IntStream, LongStream, and DoubleStream** interfaces (we will call them ***NumStreams***).

```
public interface IntStream extends BaseStream<Integer, IntStream>
```

- Some of the methods for creating a NumStreams are equivalent to how we created the source for a regular Stream<T>:

See streams\creation\NonGenericStreamElementsGeneration

- For primitive streams (only **IntStream** and **LongStream**) there are additional methods that can generate a range of numbers:

```
public static IntStream range(int startInclusive, int endExclusive)
public static IntStream rangeClosed(int startInclusive,
                                    int endInclusive)
```

See streams\creation\NonGenericStreamElementsGeneration

Converting an object to Stream

- Interface java.util.Collection<E> has methods:

```
default Stream<E> stream() {
    return StreamSupport.stream(spliterator(), false);
}
```

and

```
default Stream<E> parallelStream() {
   return StreamSupport.stream(spliterator(), true);
}
```

- Interface Map<K, V> has not stream() method, but we can use Set<Map.Entry<K, V>> **entrySet**() method and get Stream instance from the Set.

See streams\creation\CollectionsToStreamConverting

Converting an object to Stream

- Class java.util.Arrays has methods:

```
public static <T> Stream<T> stream(T[] array, int startInclusive,
                                   int endExclusive) {
    return StreamSupport.stream(spliterator(array, startInclusive,
                                            endExclusive), false);
}
public static <T> Stream<T> stream(T[] array) {
    return stream(array, 0, array.length);
}
```

- Also java.util.Arrays class has methods that return non-generic IntStream, LongStream or DoubleStream:

```
public static IntStream stream(int[] array, int startInclusive,
                               int endExclusive) {...}
public static IntStream stream(int[] array) {...}
```

See streams\creation\ArraysToStreamConverting

Converting an object to Stream

- Class java.util.**Optional**<T> class has has method:
  public Stream<T> **stream()**
  it returns a stream of the Optional value.
- Class java.lang.**String** class has method:
  public IntStream **chars()**
  it returns IntStream of charcodes.
- Also class java.lang.**String** class has method :
  public Stream<String> **lines()**
  it returns a stream of lines extracted from the string.

See streams\creation\ObjectsToStreamConverting

Converting an object to Stream

- Class java.nio.file.Files has method:

  public static Stream<Path> walk(Path start, FileVisitOption... options)
  throws IOException

  it recursively returns Path's instances for current directory's files and subdirectories.

- Class java.nio.file.Files has method:

  public static Stream<Path> list(Path dir) throws IOException

  it returns stream of the Path's instances only for current directory's files and subdirectories.

- Class java.nio.file.Files has method:

  public static Stream<String> lines(Path path) throws IOException

  it returns stream of the lines of textfile.

See streams\creation\ObjectsToStreamConverting

Stream creating methods

Method	Returned Stream Type	Finite/ Infinite	Sequential /Parallel	Ordered/ Unordered
Stream.**empty**() NumTypeStream.**empty()**	Stream<T> NumTypeStream	Finite	Sequential	Ordered
Stream.**of**(varargs) Stream.**ofNullable**(varargs) NumTypeStream.**of**(varargs)	Stream<T> Stream<T> NumTypeStream	Finite	Sequential	Ordered
Stream.**generate**(Supplier) NumTypeStream.**generate**(NumTypeSupplier)	Stream<T> NumTypeStream	Infinite	Sequential	Unordered
Stream.**iterate**(seed, UnaryOperator) NumTypeStream.**iterate**(seed, NumTypeOperator)	Stream<T> NumTypeStream	Infinite	Sequential	Ordered
Stream.**iterate**(seed, Predicate, UnaryOperator)	Stream<T>	Finite	Sequential	Ordered

Stream creating methods

Method	Returned Stream Type	Finite/ Infinite	Sequential /Parallel	Ordered/ Unordered
IntStream.**range**(startInclusive, endExclusive) (or LongStream)	IntStream or LonfStream	Finite	Sequential	Ordered
IntStream.**rangeClosed**(startInclusive, endInclusive) (or LongStream)	IntStream or LonfStream	Finite	Sequential	Ordered
collection.**stream**()	Stream<T>	Finite	Sequential	Ordered if collection ordered
collection.**parallelStream**()	Stream<T>	Finite	Parallel	Ordered if collection ordered
Arrays.**stream**(arr) Arrays.**stream**(arr, startInclusive, endExclusive)	Stream<T> NumTypeStream	Finite	Sequential	Ordered

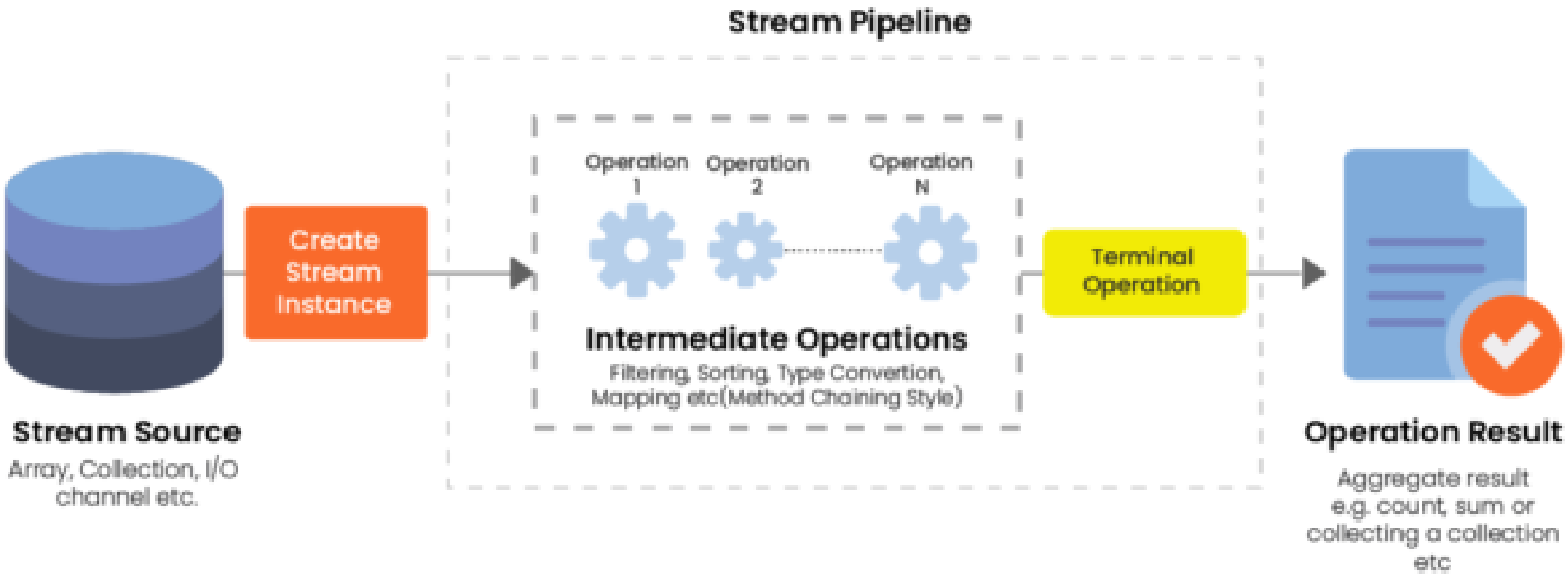
Stream creating methods

Method	Returned Stream Type	Finite/ Infinite	Sequential /Parallel	Ordered/ Unordered
optional.**stream**()	Stream<T>	Finite	Sequential	one value or Stream. empty()
charSequence.**chars**()	IntStream	Finite	Sequential	Ordered
string.**lines**()	Stream<String>	Finite	Sequential	Ordered
bufferedReader.**lines**()	Stream<String>	Finite	Sequential	Ordered
Stream.**builder**() NumTypeStream.**builder**()	Stream<T> NumTypeStream	Finite	Sequential	Ordered
Stream.**concat**(stream1, stream2) NumTypeStream.**concat**(stream1, stream2)	Stream<T> NumTypeStream	Finite if both finite	Parallel if either parallel	Ordered if both ordered

Stream creating methods

Method	Returned Stream Type	Finite/ Infinite	Sequential /Parallel	Ordered/ Unordered
random.**ints**() random.**ints**(streamSize) random.**ints**(randomNumberOrigin, randomNumberBound) random.**ints**(streamSize, randomNumberOrigin, randomNumberBound)	IntStream or LongStream or DoubleStream	Infinite Finite Infinite Finite	Sequential methods longs() and doubles(), similarly	Unordered
Files.**lines**(path) Files.**lines**(path, charset)	Stream<String>	Finite	Sequential	Ordered
Files.**list**(dir-path)	Stream<Path>	Finite	Sequential	Ordered
Files.**walk**(dir-path)	Stream<Path>	Finite	Sequential	Ordered

Java Streams



interface **java.util.stream.Stream<T>**

Operations:

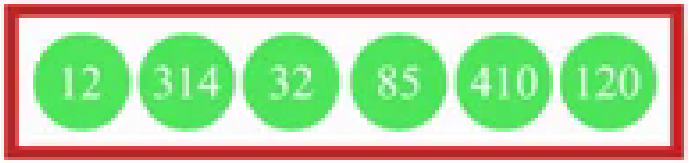
- Intermediate
- Terminal

The streams can be used only once,
the stream is no longer valid after a terminal operation completes

Intermediate stream operations

- Unlike terminal operations, intermediate operations *produce a stream as its result*.
- Each intermediate operation *maps the elements of its input stream to an output stream*, the type of the output elements may vary from the type of the input elements.
- Intermediate operations use *lazy execution* - it will never be executed unless a terminal operation is invoked.
- An intermediate operation *can deal with an infinite stream*. Their lazy nature allows truncate such infinite streams by certain intermediate and termination methods - *short-circuit execution*.
- Some intermediate operations are *stateful* - it needs to retain state from previously processed elements in order to process a new element. The most of intermediate operations are *stateless*.

Intermediate operations lazy execution



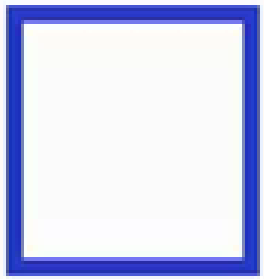

```
Stream
     .of(120, 410, 85, 32, 314, 12)
```

- The intermediate operations do not run until the terminal operation runs.
- Processing occurs from the terminal operator to the source.

Filtering Stream elements

- The **filter** method removes elements from the stream that do not match its predicate.

```
filter(x -> x > 100)
```



Stream<T> filter(Predicate<? **super** T> predicate);

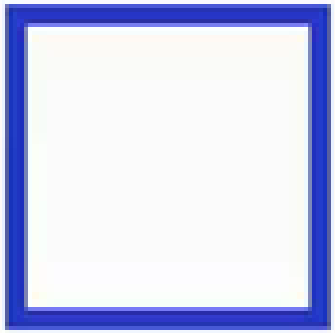
See streams\intermediate\StreamElementsFiltering

Restricting Stream elements

- The **limit**(**long** maxSize) method returns the Stream truncated to first maxSize elements of original Stream.
- The **skip**(**long** n) method returns the Stream with discarded first n elements of original Stream.
- The limit and skip methods can make a Stream smaller, or they could make a finite stream out of an infinite stream.
- The Stream<T> **distinct()** method returns a stream with duplicate values removed.
  Java calls equals() to determine whether the objects are the same.
  The duplicates do not need to be adjacent to be removed.

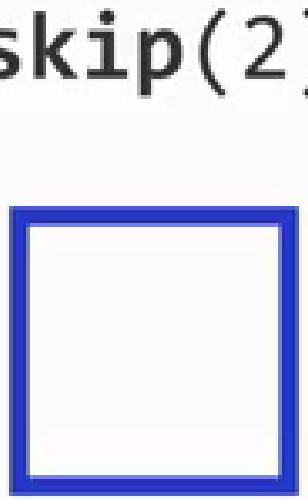
Restricting Stream elements

```
limit(4)
```



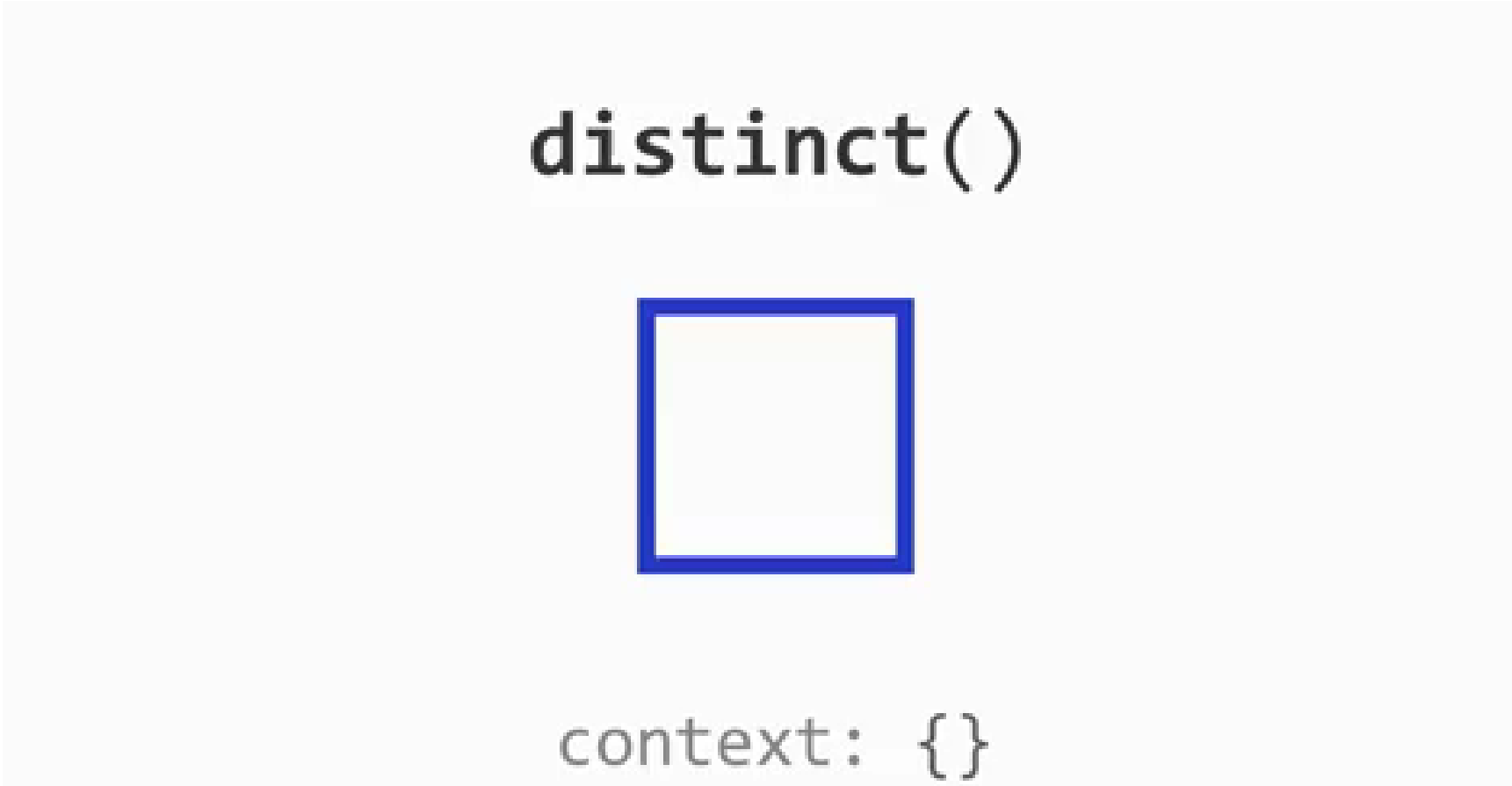
```
Stream<T> limit(long maxSize);
```

Restricting Stream elements




```
Stream<T> skip(long n);
```

Restricting Stream elements



Stream<T> distinct();

See streams\intermediate\StreamElementsRestricting

Restricting Stream elements

- In JDK 9 the methods have been added to the Stream<T> interface:

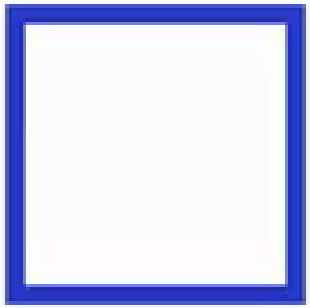
  default Stream<T> **takeWhile**(Predicate<? **super** T> predicate)

  default Stream<T> **dropWhile**(Predicate<? **super** T> predicate)

- The **takeWhile** method returns the elements of the stream as long as they satisfy the Predicate condition.
- If the predicate returns false for the next element, method terminates.
- The method works similarly to the Stream **<T> limit(long maxSize)**, but with a condition.
- The **takeWhile** method skips the elements as long as they satisfy the Predicate condition.
- If the predicate returns false for the next element, no other elements will be skipped - method returns all of them.
- The method is similar to the Stream **<T> skip(long n)**, but with a condition.

Restricting Stream elements

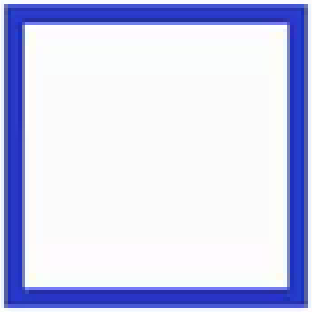
```
takeWhile(x -> x < 3)
```



default Stream<T> **takeWhile**(Predicate<? **super** T> predicate)

Restricting Stream elements

```
dropWhile(x -> x < 3)
```



default Stream<T> **dropWhile**(Predicate<? **super** T> predicate)

See streams\intermediate\StreamElementsRestricting

Examining elements in a Stream

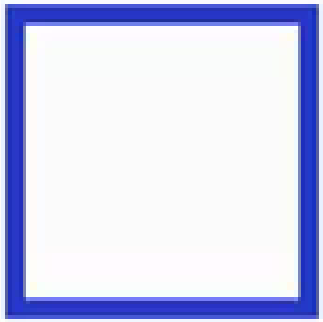
- The **peek**() method allows stream elements to be examined at the point where the method invoked.
- This method allows us to perform a stream operation without actually changing the stream.
- This method is useful for reporting and debugging purposes.
- Java doesn't prevent us from writing bad peek code that change the stream.

```
Stream<T> peek(Consumer<? super T> action);
```

- Recommendation:
  Use peek() for printing and debugging purpose only

Examining elements in a Stream

```
peek(x -> System.out.format("%s, ", x)
```

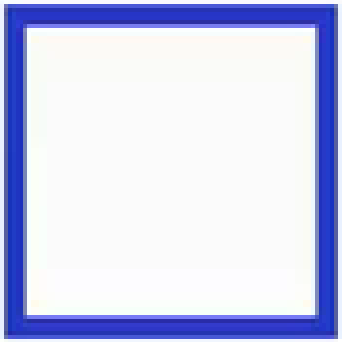


Stream<T> **peek**(Consumer<? **super** T> action);

See streams\intermediate\StreamElementsExaming

Sorting Stream elements

sorted()



The following Stream<T> intermediate methods can be used to sort the elements of the stream:

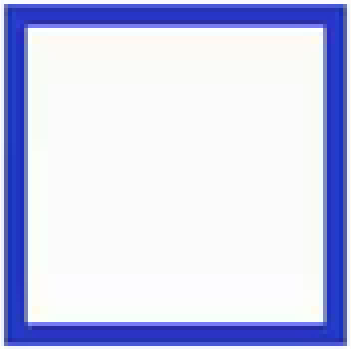
```
Stream<T> sorted();
Stream<T> sorted(Comparator<? super T> comparator);
```

See streams\intermediate\StreamElementsSorting

Transforming Stream elements

- The **map** method of the Stream <T> interface applies *the mapper function* to each element, and then returns a stream in which the elements will be the function results.

```
map(x -> x + 11)
```



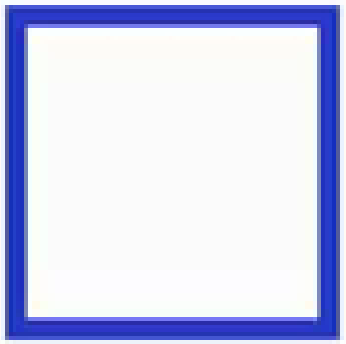
<R> Stream <R> **map** (Function <? Super T ,? extends R> mapper)

See streams\intermediate\StreamElementsTransforming - 1s part

Transforming Stream elements

- The **flatMap** method of the Stream<T> interface differs from the map in that it returns a stream of output types.
- This allows you to combine the elements of *multiple* input streams into *one* result stream (it removes empty input streams also).

```
flatMap(x -> Stream.range(0, x))
```



<R> Stream <R> **flatMap** (Function <? Super T, **? extends Stream <? extends R >>** mapper

See streams\intermediate\StreamElementsTransforming - 2nd part

Transforming Stream elements

- Since JDK 16
  default <R> Stream<R> mapMulti(BiConsumer<? super T,
  ? super Consumer<R>> mapper)
  was added to Stream<T> interface
- The implementation of the BiConsumer takes a Stream element T, if necessary, transforms it into type R, and invokes the mapper's Consumer::accept.
- Inside Java's **mapMulti** method implementation, the mapper is a buffer that implements the Consumer functional interface.
- This allows you to combine the elements of *multiple* input streams into *one* result stream like flatmap, but without multiple stream creation that avoids the overhead of creating a new Stream for every group of mapped elements.
- There are **mapMultiToNum** methods for Num=Int, Long, Double in the Stream<T> interface:
  default NumStream mapMultiToNum(BiConsumer<? super T,
  ? super NumConsumer> mapper)

See streams\intermediate\StreamElementsTransforming - 3d part

Stream<T>-NumStream mapping

- We can map Stream<T> instance to non-generic NumStreams and vice versa && map between NumStreams:

Source stream type	To Stream<T>	To IntStream	To LongStream	To DoubleStream
Stream<T>	**map**(Function<? super T, ? extends R> mapper)	**mapToInt**(ToIntFunction<? super T> mapper)	**mapToLong**(ToLongFunction<? super T> mapper)	**mapToDouble**(ToDoubleFunction<? super T> mapper)
IntStream	**mapToObj**(IntFunction<? extends U> mapper)	**map**(IntUnaryOperator mapper)	**mapToLong**(IntToLongFunction mapper)	**mapToDouble**(IntToDoubleFunction mapper)
LongStream	**mapToObj**(LongFunction<? extends U> mapper)	**mapToInt**(LongToIntFunction mapper)	**map**(LongUnaryOperator mapper)	**mapToDouble**(LongToDoubleFunction mapper)
DoubleStream	**mapToObj**(DoubleFunction<? extends U> mapper)	**mapToInt**(DoubleToIntFunction mapper)	**mapToLong**(DoubleToLongFunction mapper)	**map**(DoubleUnaryOperator mapper)

See streams\intermediate\StreamsMapping - 1st part

flatMap & mapMulti to NumStreams

- We can use **flatMapToNum** method for mapping between Stream<T> and non-generic NumStreams instances:

```
NumStream flatMapToNum(Function<? super T,
                       ? extends NumStream> mapper);
```

- There no **flatmap** method for mapping between non-generic NumStream instance and Stream<T> instance

- We can use **boxed**() method of non-generic NumStreams for primitive type elements boxing to it Wrappers:

```
Stream<Integer> boxed();
```

- We can use **flatMap** method for mapping between THE SAME TYPE non-generic NumStreams:

```
NumStream flatMap(NumFunction<? extends NumStream> mapper);
```

See streams\intermediate\StreamsMapping - 2nd part

Stream pipeline methods order

```
Stream.generate(() ->"Elsa")
        .filter(x -> x.length() == 4)
        .sorted()
        .limit(2)
        .forEach(System.out::println);
```

Program hangs because of sorted() (STATEFUL) doesn't know if all elements processed

```
Stream.generate(() -> "Elsa")
        .filter(x -> x.length() == 4)
        .limit(2)
        .sorted()
        .forEach(System.out::println);    //Elsa Elsa
```

```
Stream.generate(() ->"Peter")
.filter(x -> x.length() == 4)
.limit(2)
.sorted()
.forEach(System.out::println);
```

Program hangs because of filter doesn't receive any data

See streams\intermediate\StreamPipelineMethodsOrder

Intermediate stream operations

Method	Stateful/ Stateless	Can change stream size	Can change stream type	Encounter order
Stream<T> **distinct**() NumStream **distinct**()	Stateful	Yes	No	Unchanged
Stream<T> **dropWhile**(Predicate p) NumStream **dropWhile**(NumPredicate p)	Stateful	Yes	No	Unchanged
Stream<T> **filter**(Predicate p) NumStream **filter(**NumPredicate p)	Stateless	Yes	No	Unchanged
Stream<T> **flatMap**(Function m) NumStream **flatMapToNum**(Function m) NumStream **flatMap**(Function m)	Stateless	Yes	Yes	Not guaranteed

Intermediate stream operations

Method	Stateful/ Stateless	Can change stream size	Can change stream type	Encounter order
Stream<T> **map**(Function m) NumStream **mapToNum**(ToNumFunction m); <R> Stream<R> **mapMulti**(BiConsumer bi) NumStream **mapMultiToNum**(BiConsumer bi) NumStream **map**(NumUnaryOperator m) NumStream **mapMulti**(NumMapMultiConsumer m) NumStream **mapToNum**(NumToNumFunction m) <U> Stream<U> **mapToObj**(NumFunction m)	Stateless	No	Yes	Not guaranteed

Intermediate stream operations

Method	Stateful/ Stateless	Can change stream size	Can change stream type	Encounter order
Stream<T> **limit**(long maxSize) NumStream **limit**(long maxSize)	Stateful, short-circuited	Yes	No	Unchanged
Stream<T> **peek**(Consumer a) NumStream **peek**(NumConsumer a)	Stateless	No	No	Unchanged
Stream<T> **skip**(long n) NumStream **skip**(long n)	Stateful	Yes	No	Unchanged
Stream<T> **sorted**() NumStream **sorted**()	Stateful	No	No	Ordered

Intermediate stream operations

Method	Stateful/ Stateless	Can change stream size	Can change stream type	Encounter order
Stream<T> **takeWhile**(Predicate p) NumStream **takeWhile**(NumPredicate p)	Stateful, short-circuited	Yes	No	Unchanged
Stream<Num> **boxed**()	Stateful	No	Yes	Unchanged

Terminal stream operations

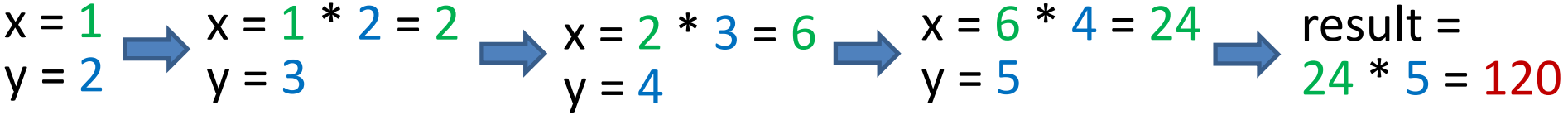
- Terminal operations *do not produce a stream as its result*.
- Terminal operations *produce single result* as T instance or boolean or Optional<T> or collection depend on the method invoked.
- Terminal operations use *eager execution* - they are executed immediately when the terminal operation is invoked on the stream.
- Terminal operations *can also deal with an infinite stream* - they terminate or not terminate it depend on the method invoked.
- There are terminal operation types:
  - Reductions
  - Aggregations (Collections)
  - Finding and matching
  - Consuming

Reducing Stream elements

- Reduction operations reduce the Stream's elements *to a single result* by repeatedly applying an *accumulator* - BinaryOperator<T>.
- Such an operator uses the previous result to combine it with the current element to generate a new result as Optional<T> instance.

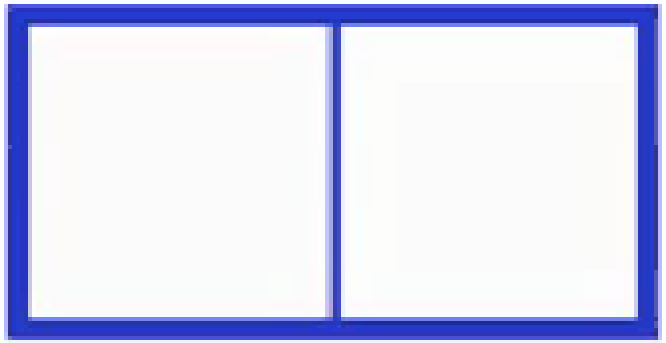
```
Optional<T> reduce(BinaryOperator<T> accumulator);
```

```
BinaryOperator<Integer> product = (x, y) -> x * y;
Stream.of(1, 2, 3, 4, 5)                    // Stream<Integer>
      .reduce(product)                      // Optional<Integer>
      .ifPresent(System.out::println);      //120
```



Reducing Stream elements

```
reduce((acc, x) -> acc + x)
```



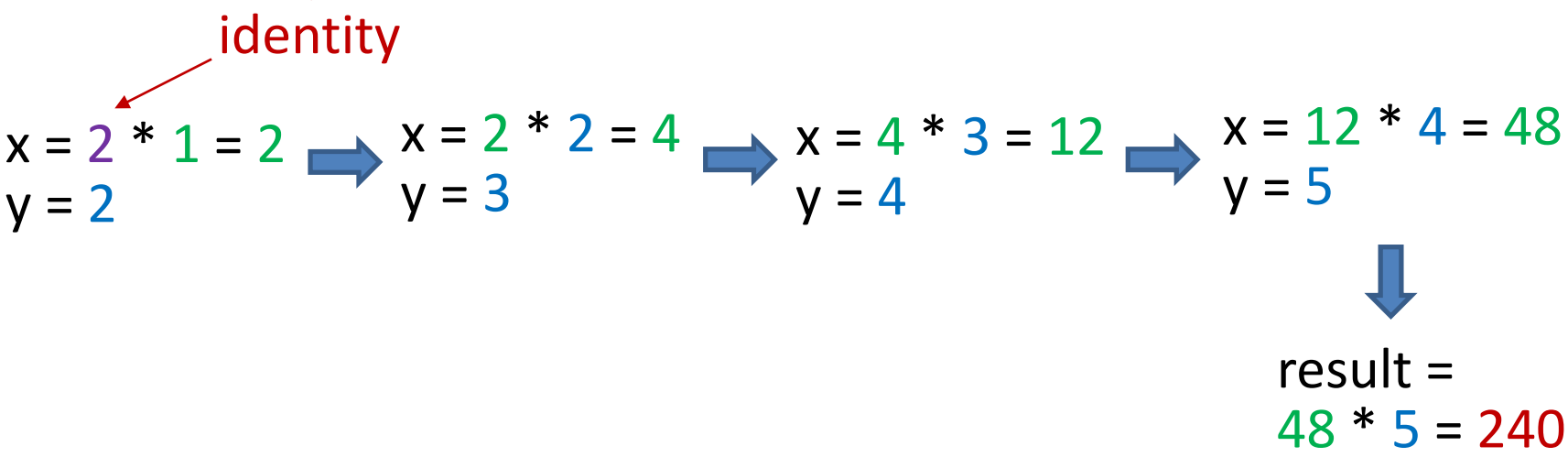
Stream<T> has:
public T reduce(BinaryOperator<T> accumulator);

Reducing Stream elements

- There is overloaded version of the **reduce** method with an *identity* - the initial value for the running computation.
- This method returns the T instance.

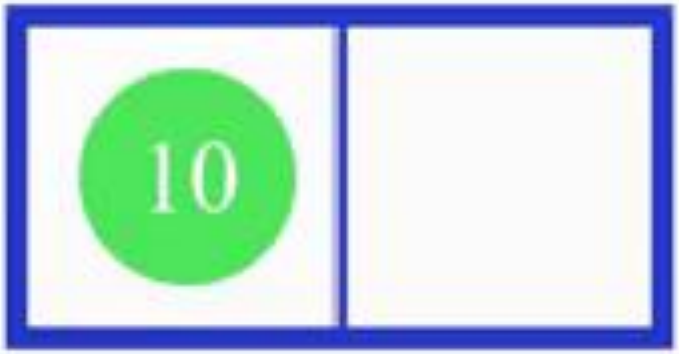
```
T reduce(T identity, BinaryOperator<T> accumulator);
```

```
var res = Stream.of(1, 2, 3, 4, 5)          //Stream<Integer>
        .reduce(2, (x, y) -> x * y);        //Integer
System.out.println(res);                    //240
```



Reducing Stream elements

```
reduce(10, (acc, x) -> acc + x)
```



Stream<T> has:
public T reduce(T identity, BinaryOperator<T> accumulator);

Reducing Stream elements

- There is overloaded version of the **reduce** method with:
  - an *identity* - the initial output type U value for the running computation,
  - BiFunction<U, ? **super** T, U> *accumulator* - that transforms the input type T element to output type U element,
  - BinaryOperator<U> *combiner* - that combines output type U elements.
- This method returns the U instance.

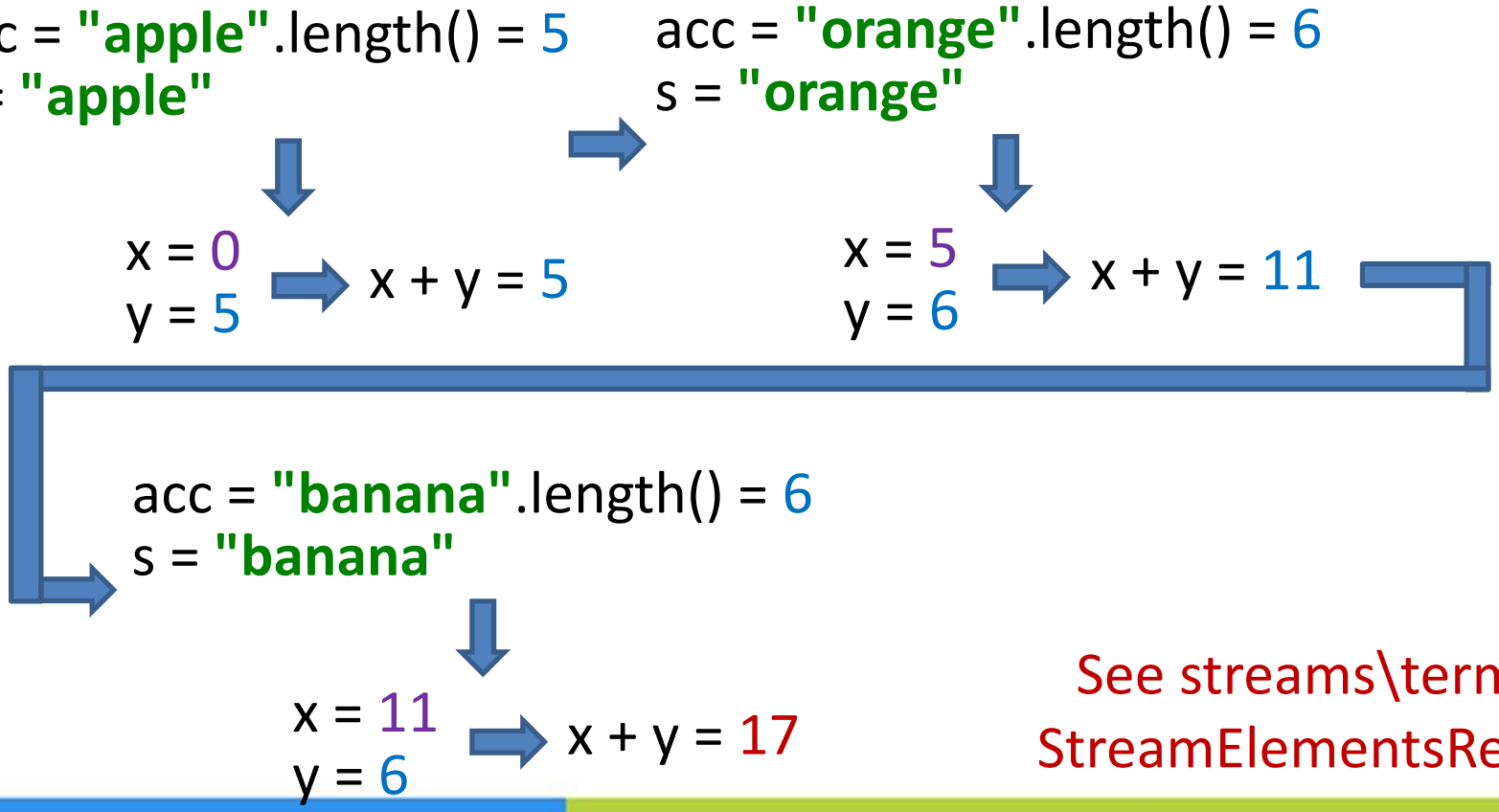
```
<U> U reduce(U identity,
             BiFunction<U, ? super T, U> accumulator,
             BinaryOperator<U> combiner);
```

- This method combines mapping and reducing.

Reducing Stream elements

```
var res = Stream.of("apple", "orange", "banana") //Stream<String>
        .reduce(0,                                //Integer
                (acc, s) -> acc + s.length(),     //Integer
                (x, y) -> x + y);                 //Integer
System.out.println(res);                          //17
```

Reduce only



See streams\terminal\StreamElementsReducing

Reducing Stream elements

- The Java API provides the **non-generic IntStream, LongStream, and DoubleStream** interfaces (we will call them ***NumStreams***) which support stream of Integers, Longs, and Doubles, respectively (we will call them ***Nums***).
- They have additional **reduce** methods:

```
OptionalInt reduce(IntBinaryOperator op);

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

See streams\terminal\NumStreamElementsReducing

Stream elements reducing methods

- The Stream<T> **min** method selects the smallest element in the stream according to the comparator provided in its argument. The Stream<T> **max** method selects the largest element in the stream according to the comparator provided in its argument.
- Both methods return an **Optional** of the same type as the stream elements.

```
Optional<T> min(Comparator<? super T> comparator);
Optional<T> max(Comparator<? super T> comparator);
```

- The Stream<T> **count**() method determines the number of elements in a finite stream:

```
long count();
```

See streams\terminal\ReducingMethods - 1st part

NumStream elements reducing methods

- The Java API provides the **non-generic IntStream, LongStream, and DoubleStream** interfaces (we will call them ***NumStreams***) which support stream of Integers, Longs, and Doubles, respectively (we will call them ***Nums***) .
- They have additional reducing methods:

```
OptionalNum min();
OptionalNum max();
OptionalDouble average();
num sum();
long count();
NumSummaryStatistics summaryStatistics();
```

See streams\terminal\ReducingMethods - 2nd part

Collecting Stream elements

- ***Collecting*** is a special type of reduction called a ***mutable reduction***. Collecting is more efficient than a regular reduction because we use the same *mutable* object while accumulating.
- A ***collector*** is a general construct for generating *composite values* from streams. With the collectors, You can collect all the elements in a list, set or other collection, group elements by criterion, combine elements into a string, etc.
- Collector behavior can be defined with *supplier*, *accumulator* and *combiner* instances that are parameters of the Stream<T> **collect** method:

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

- Also Stream<T> has overloaded **collect** method with java.util.stream.Collector<T, A, R> instance as parameter:

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

Collecting Stream elements

- Collecting is a *reduction to mutable container*, so we can define collection to list with reduce operation:

```
<U> U reduce(U identity, BiFunction<U,
                         ? super T, U> accumulator,
                         BinaryOperator<U> combiner);:
```

See streams\terminal\StreamElementsCollecting - 1st part

Problems:

- a lot of verbose code,
- new instances of ArrayList created for each element,
- go against the general concept of reduce of being an immutable reduction operation

Collecting Stream elements

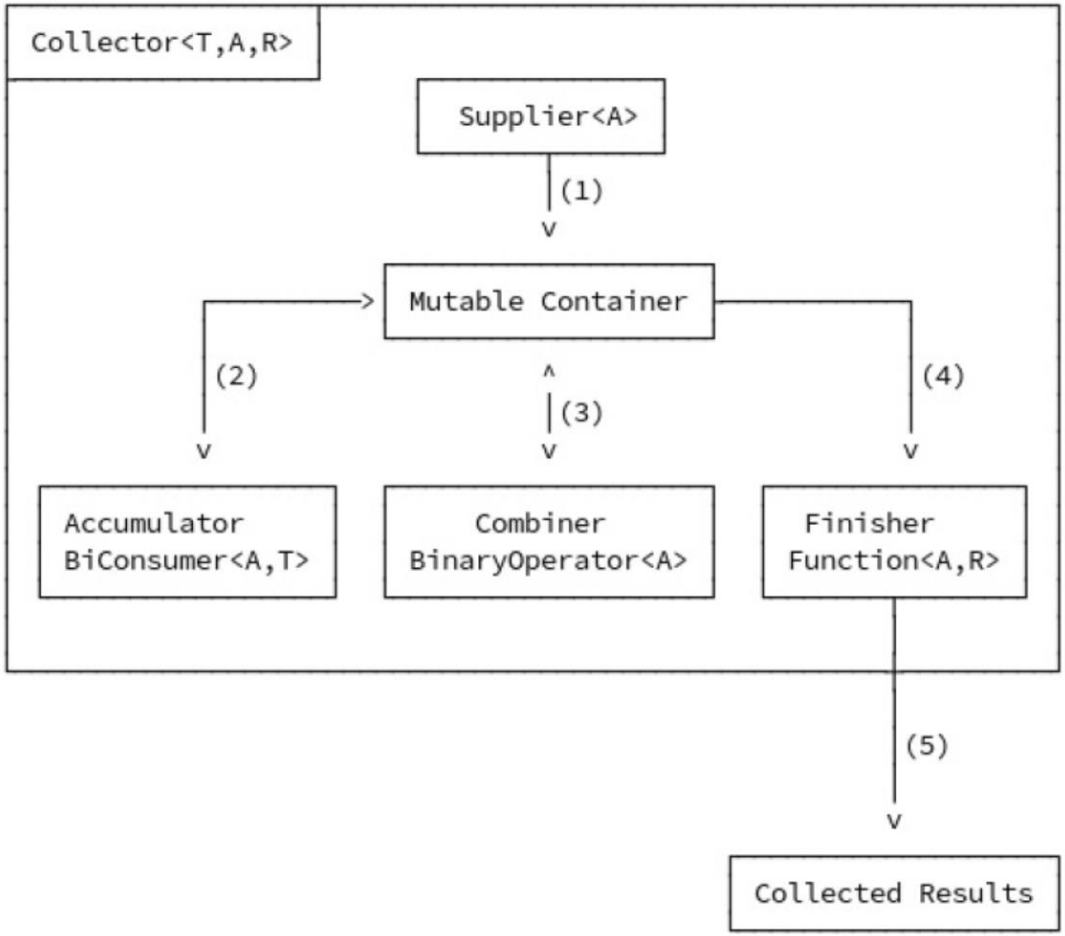
- There's a better solution available - aggregation operations - collecting:

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

See streams\terminal\StreamElementsCollecting - 2nd part

Collecting Stream elements

- Optionally, you can write your own collector by implementing the **java.util.stream.Collector** interface - is an abstraction of a reduction operation that accumulates input elements into a mutable container, converting the accumulated result into a final representation after processing all input elements.



Collecting Stream elements

```
public interface Collector<T, A, R> {
    Supplier<A> supplier();
    BiConsumer<A, T> accumulator();
    BinaryOperator<A> combiner();
    Function<A, R> finisher();
    Set<Characteristics> characteristics();
    public static<T, R> Collector<T, R, R> of(Supplier<R> supplier,
                                              BiConsumer<R, T> accumulator,
                                              BinaryOperator<R> combiner,
                                              Characteristics... characteristics) {...}
    public static<T, A, R> Collector<T, A, R> of(Supplier<A> supplier,
                                                 BiConsumer<A, T> accumulator,
                                                 BinaryOperator<A> combiner,
                                                 Function<A, R> finisher,
                                                 Characteristics... characteristics) {...}
    enum Characteristics {...}   ... }
```

Collecting Stream elements

The **Collector<T, A, R>** interface has three type variables:

- **T** - type of input elements of the reduction operation;
- **A** - type of accumulator instance for the reduction operation (often hidden as a detail of the implementation);
- **R** - type of result of the reduction operation.

The collector is defined by four functions that work together to accumulate records in a mutable result container, and perform the final transformation of the result:

- Supplier <A> supplier - a function that creates and returns a mutable result container (supplier);
- BiConsumer <A, T> accumulator - a function that performs the addition of a new data element to the results container (drive);
- BinaryOperator <A> combiner - a function that combines two partial results into one (combiner);
- Function <A, R> finisher - a function that performs the optional final conversion of the intermediate data type of the battery to the final data type of the result of the operation (finisher).

Collecting Stream elements

The Collector interface also contains a **set of characteristics** in the Characteristics enumerator, which provide hints on how implementations of the reduction operation functions can be used:

- **CONCURRENT** - indicates that the battery function can be called by several execution threads simultaneously;
- **UNORDERED** - indicates that the reduction operation does not guarantee the order of the elements equal to their input order;
- **IDENTITY_FINISH** - indicates that the finisher function is an identity function and can be removed.

A sequential implementation of the collector reduction will create a single result container. A parallel implementation will split the input, create a results container for each part.

See streams\terminal\CollectorUsing

Collecting Stream elements

- The **java.util.stream.Collector**s class has a lot of pre-built collecting methods for all occasions, we use method to get collector to list:

See streams\terminal\CollectorUsing

- The **java.util.stream.Collectors** class contains static methods which create **Collector** objects that solve various problems.

Prewritten Collectors - collect to collections

- There are methods which create Collector's instances for stream elements collect to collections of different types:

```
public static <T> Collector<T, ?, List<T>> toList()
public static <T> Collector<T, ?, List<T>> toUnmodifiableList()
public static <T> Collector<T, ?, Set<T>> toSet()
public static <T> Collector<T, ?, Set<T>> toUnmodifiableSet()
public static <T, K, U> Collector<T, ?, Map<K,U>> toMap(
                        Function<? super T, ? extends K> keyMapper,
                        Function<? super T, ? extends U> valueMapper)
public static <T, K, U> Collector<T, ?, Map<K,U>> toUnmodifiableMap(
                        Function<? super T, ? extends K> keyMapper,
                        Function<? super T, ? extends U> valueMapper)
public static <T, C extends Collection<T>> Collector<T, ?, C>
                        toCollection(Supplier<C> collectionFactory)
```

- Since JDK16 there is a default List<T> **toList()** method in the Stream<T> interface that returns an unmodifiable list.

See streams\terminal\PrewrittenCollectorsToCollections

Prewritten Collectors - for join, min/max, count

- These methods which creates Collector's instances for String stream elements joining, returning min or max value and counting of stream elements of any type:

```
public static Collector<CharSequence, ?, String> joining()
public static Collector<CharSequence, ?, String> joining(
                                                CharSequence delimiter)
public static Collector<CharSequence, ?, String> joining(
                                                CharSequence delimiter,
                                                CharSequence prefix,
                                                CharSequence suffix)
public static <T> Collector<T, ?, Optional<T>> minBy(
                                                Comparator<? super T> comparator)
public static <T> Collector<T, ?, Optional<T>> maxBy(
                                                Comparator<? super T> comparator)
public static <T> Collector<T, ?, Long> counting()
```

See streams\terminal\PrewrittenCollectors - 1st part

Prewritten Collectors - for filtering and reducing

- These methods creates Collector's instances for stream elements filtering before they are collected and stream elements reducing to Optional or T value.

```
public static <T, A, R> Collector<T, ?, R> filtering(
                        Predicate<? super T> predicate,
                        Collector<? super T, A, R> downstream)
public static <T> Collector<T, ?, Optional<T>>reducing(
                                        BinaryOperator<T> op)
public static <T> Collector<T, ?, T>reducing(T identity,
                                        BinaryOperator<T> op)
Collector<T, ?, U> reducing(U identity,
                        Function<? super T, ? extends U> mapper,
                        BinaryOperator<U> op)
```

- The reducing() collectors are useful when used in a multi-level reduction or downstream (see later). To perform a simple reduction on a stream, use reduce(BinaryOperator) method of Stream<T>.

See streams\terminal\PrewrittenCollectors - 2nd part

Prewritten Collectors - for mapping and grouping

- These methods creates Collector's instances for stream elements grouping and mapping to downstream collector:

```
public static <T, U, A, R> Collector<T, ?, R> mapping(
                    Function<? super T, ? extends U> mapper,
                    Collector<? super U, A, R> downstream)
public static <T, K> Collector<T, ?, Map<K, List<T>>> groupingBy(
                    Function<? super T, ? extends K> classifier)
public static <T, K, A, D> Collector<T, ?, Map<K, D>> groupingBy(
                    Function<? super T, ? extends K> classifier,
                    Collector<? super T, A, D> downstream)
public static <T, K, D, A, M extends Map<K, D>> Collector<T, ?, M>
                groupingBy(Function<? super T, ? extends K> classifier,
                    Supplier<M> mapFactory,
                    Collector<? super T, A, D> downstream)
```

See streams\terminal\PrewrittenCollectors - 3d part

Prewritten Collectors - for mapping and partitioning

- These methods creates Collector's instances for stream elements partitioning and mapping to downstream collector:

```
public static <T> Collector<T, ?, Map<Boolean, List<T>>>
                    partitioningBy(Predicate<? super T> predicate)
public static <T, D, A> Collector<T, ?, Map<Boolean, D>>
                    partitioningBy(Predicate<? super T> predicate,
                                   Collector<? super T, A, D> downstream)
```

See streams\terminal\PrewrittenCollectors - 4th part

Prewritten Collectors - for collector elements mapping and collectors composing

- The collectingAndThen method collects the elements of a stream using a Collector object and then transforms the result to another type using a function:

```
public static<T,A,R,RR> Collector<T,A,RR> collectingAndThen(
                                    Collector<T,A,R> downstream,
                                    Function<R,RR> finisher)
```

- The teeing method returns a Collector that is a composite of two downstream collectors. Every element passed to the resulting collector is processed by both downstream collectors, then their results are merged using the specified merge function into the final result.

```
public static <T, R1, R2, R> Collector<T, ?, R> teeing(
                        Collector<? super T, ?, R1> downstream1,
                        Collector<? super T, ?, R2> downstream2,
                        BiFunction<? super R1, ? super R2, R> merger)
```

See streams\terminal\PrewrittenCollectors - 5th part

Prewritten Specialized Collectors

- The Collectors class has methods which produce a sum, average or SummaryStatistics instance for NumStream, where Num = Int, Long and Double:

```
public static <T> Collector<T, ?, Integer> summingInt(
                        ToIntFunction<? super T> mapper)
public static <T> Collector<T, ?, Double> averagingInt(
                        ToIntFunction<? super T> mapper)
public static <T> Collector<T, ?, IntSummaryStatistics>
                    summarizingInt(ToIntFunction<? super T> mapper)
```

- The methods for Long and Double specialized streams are similar.

See streams\terminal\PrewrittenSpecializedCollectors

Finding an element of Stream

- The **findAny**() method returns an arbitrary element of the stream, usually the first one (but there is no guarantee for this) in the encounter order and terminates the stream. If the stream is empty, it returns an empty Optional.

  OptionalInt **findAny**();

- The **findFirst**() method returns the first element in the stream and terminates the stream. If the stream is empty, it returns an empty Optional.

  Optional<T> **findFirst**();

- These methods can terminate with an infinite stream. Since Java generates only the amount of stream you need, the infinite stream needs to generate only one element.
- These methods return a value based on the stream but do not reduce the entire stream into one value.

See streams\terminal\StreamElementsFinding

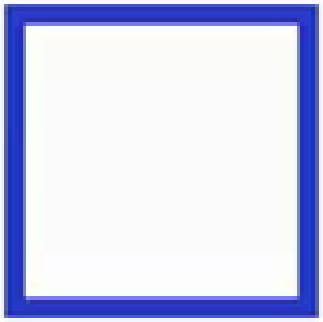
Matching elements of Stream

- The **allMatch**() method returns true if all elements in the stream match the given predicate. If the stream is empty, it returns true:
  boolean **allMatch**(Predicate<? super T> predicate);
- The **anyMatch**() method returns true if any element in the stream matches the given predicate. If the stream is empty, it returns false:
  boolean **anyMatch**(Predicate<? super T> predicate);
- The **noneMatch**() method returns true if no element in the stream matches the given predicate. If the stream is empty, it returns true:
  boolean **noneMatch**(Predicate<? super T> predicate);
- These methods may or may not terminate for infinite streams. It depends on the data.
- Like the find methods, these methods are not reductions because they do not necessarily look at all of the elements.

See streams\terminal\StreamElementsMatching

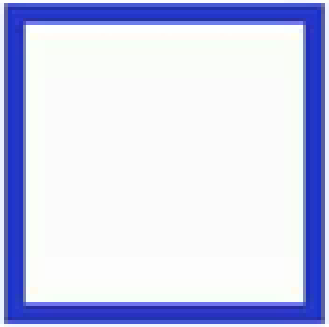
Matching elements of Stream

```
allMatch(x -> x <= 7)
```



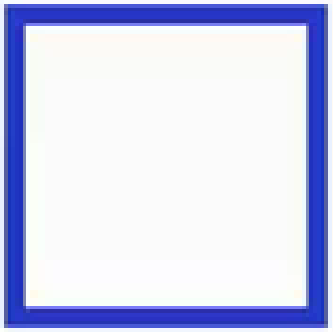
Matching elements of Stream

```
allMatch(x -> x < 3)
```



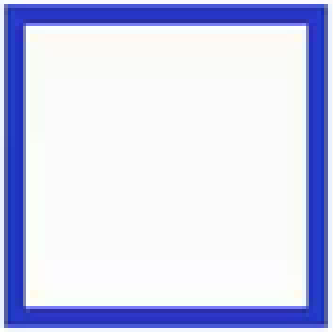
Matching elements of Stream

```
anyMatch(x -> x == 3)
```



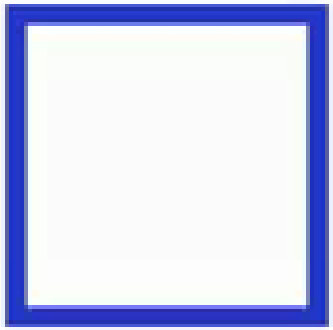
Matching elements of Stream

```
anyMatch(x -> x == 8)
```



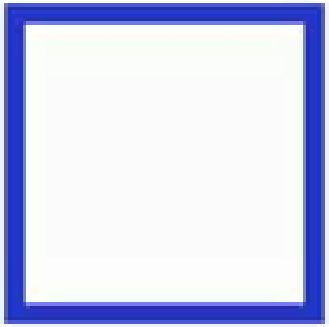
Matching elements of Stream

```
noneMatch(x -> x == 9)
```



Matching elements of Stream

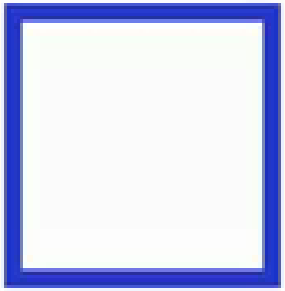
```
noneMatch(x -> x == 3)
```



Stream Elements Consuming

- Streams can be traversed by providing a consumer to the object's **forEach** method like collection.

```
forEach(x -> System.out.format("%s, ", x)
```



```
void forEach(Consumer<? super T> action);
```

See streams\terminal\StreamElementsConsuming

Streams chaining

- We can chain streams in implicit or explicit way.

See streams\StreamsChaining