TECHIN

STREAMS

Jaroslav Grablevski

Stream API

- Srautas (angl. Stream) - tai grandinė (angl. pipeline) sudaryta iš funkcijų. Kai duomenys iš srauto šaltinio pradeda judėti šia grandine, tai kiekvienos funkcijos įvykdymas gali pakeisti srauto turinį (objektų tipus, srauto elementų kiekį bei jų rūšiavimą).

Streams vs lists

- Streams have more convenient methods than Lists
  - forEach, filter, map, reduce, min, sorted, distinct, limit, etc.
- Streams have cool properties that Lists lack
  - Making streams more powerful, faster, and more memory efficient than Lists
  - The three coolest properties
    - Lazy evaluation
    - Automatic parallelization
    - Infinite (unbounded) streams
- Streams do not store data
  - They are just programmatic wrappers around existing data sources

Term stream in java

- **I/O streams**
  - Input streams: low-level data structures for reading from socket or file or other input source.
    - InputStream, ObjectInputStream, FileInputStream, ByteArrayInputStream, etc.
  - Output streams: low-level data structures for sending data to socket or file.
    - OutputStream, ObjectOutputStream, FileOutputStream, ByteArrayOutputStream, etc.
- **Java 8 Stream interface**
  - Stream<T> (e.g., Stream<String>): High-level wrapper around arrays, Lists, and other data source. Introduced in Java 8.
  - IntStream, DoubleStream, etc. Specializations of Java 8 streams for numbers.

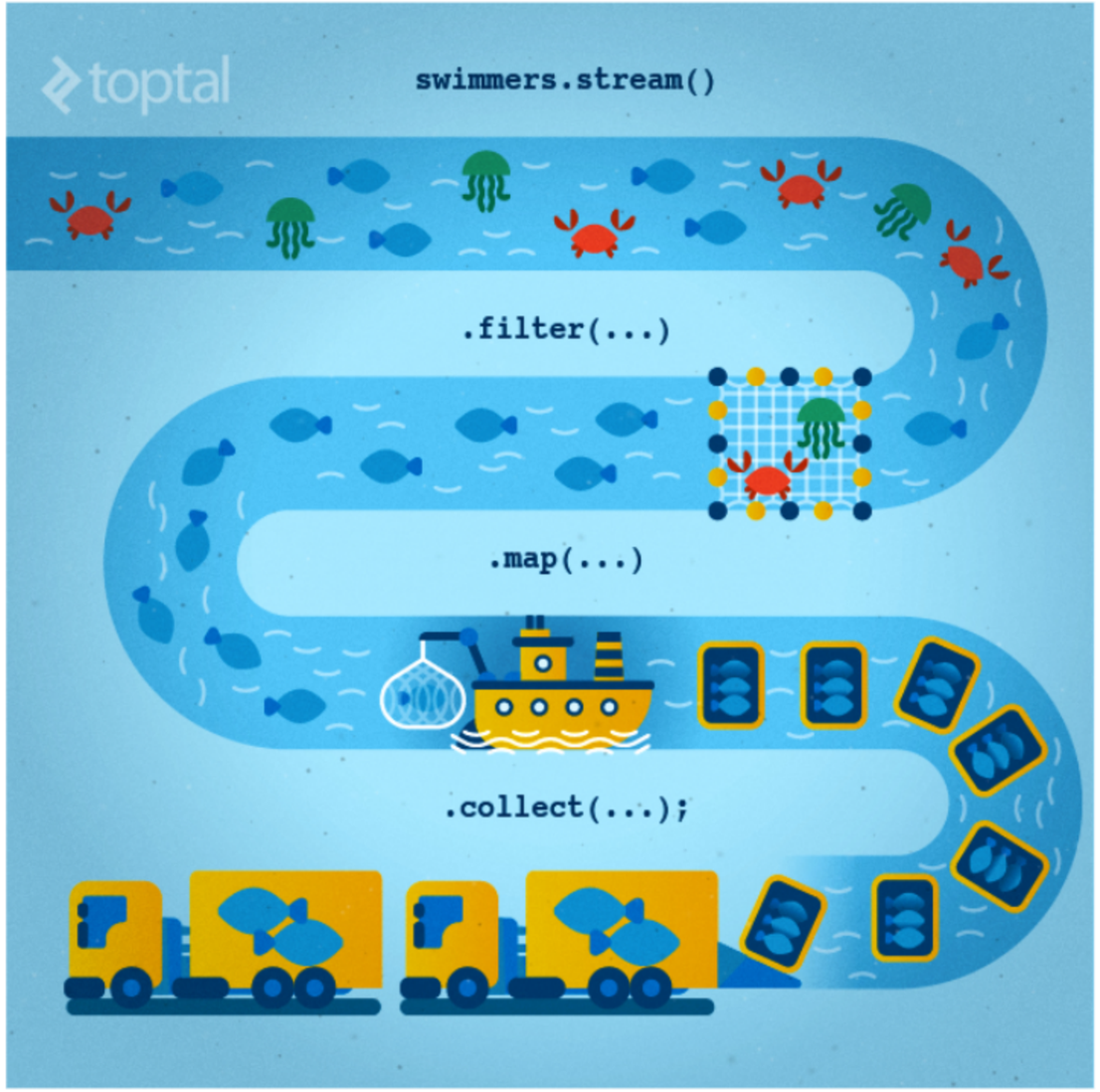
Characteristics of Streams

- Not data structures
  - Streams have no storage; they carry values from a source through a pipeline of operations.
  - They also never modify the underlying data structure (e.g., the List or array that the Stream wraps)
- Designed for lambdas
  - All Stream operations take lambdas as arguments
- Do not support indexed access
  - You can ask for the first element, but not the second or third or last element
- Can easily be output as Lists or arrays
  - Simple syntax to build a List or array from a Stream

Characteristics of Streams

- Lazy
  - Most Stream operations are postponed until it is known how much data is eventually needed
- Parallelizable
  - If you designate a Stream as parallel, then operations on it will automatically be done in parallel, without having to write explicit fork/join or threading code
- Can be unbounded
  - Unlike with collections, you can designate a generator function, and clients can consume entries as long as they want, with values being generated on the fly

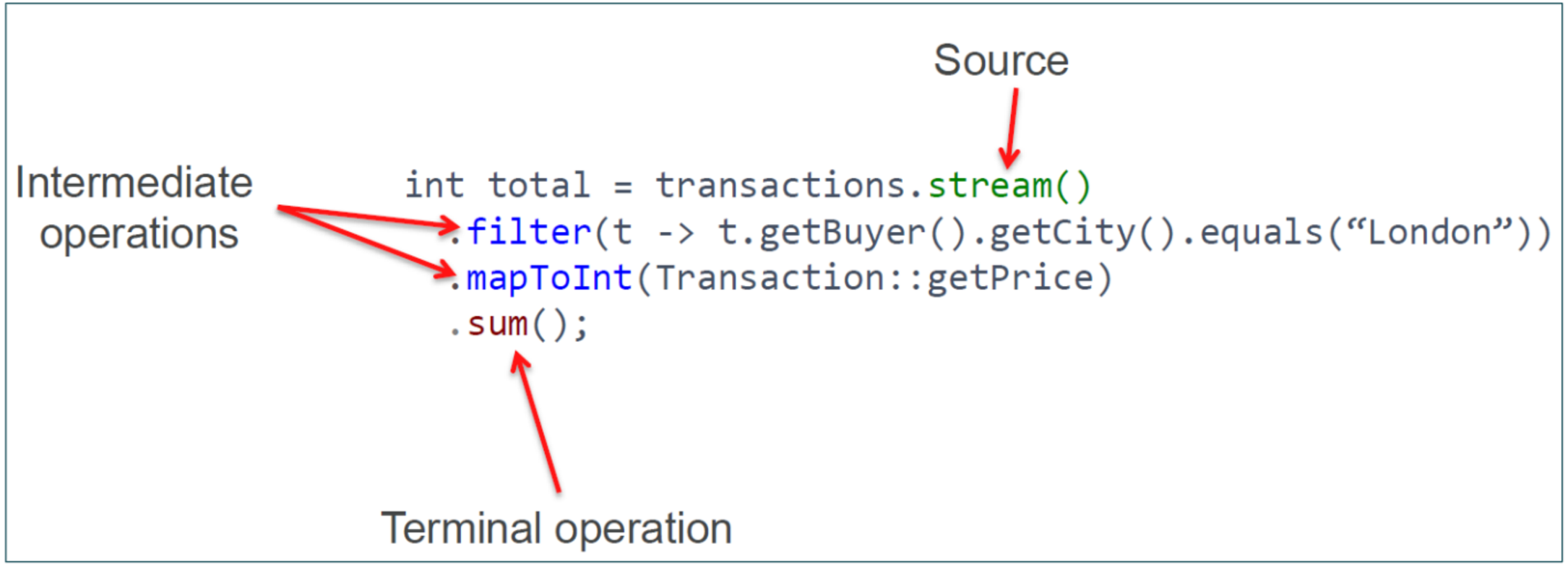
Stream API



Stream API

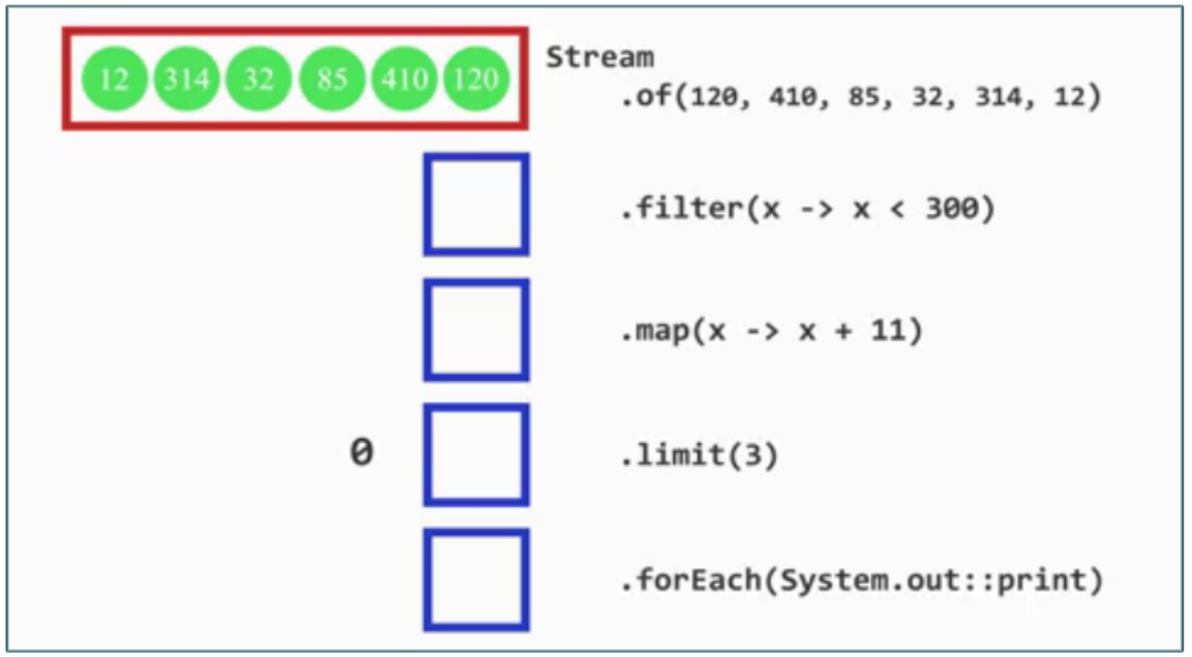
- A stream pipeline consists of three types of things
  - A source
  - Zero or more intermediate operations
  - A terminal operation
    - Producing a result or a side-effect

Stream API


Source
Intermediate operations
int total = transactions.stream()
.filter(t -> t.getBuyer().getCity().equals("London"))
.mapToInt(Transaction::getPrice)
.sum();
Terminal operation

Stream API

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



Stream pavyzdžiai

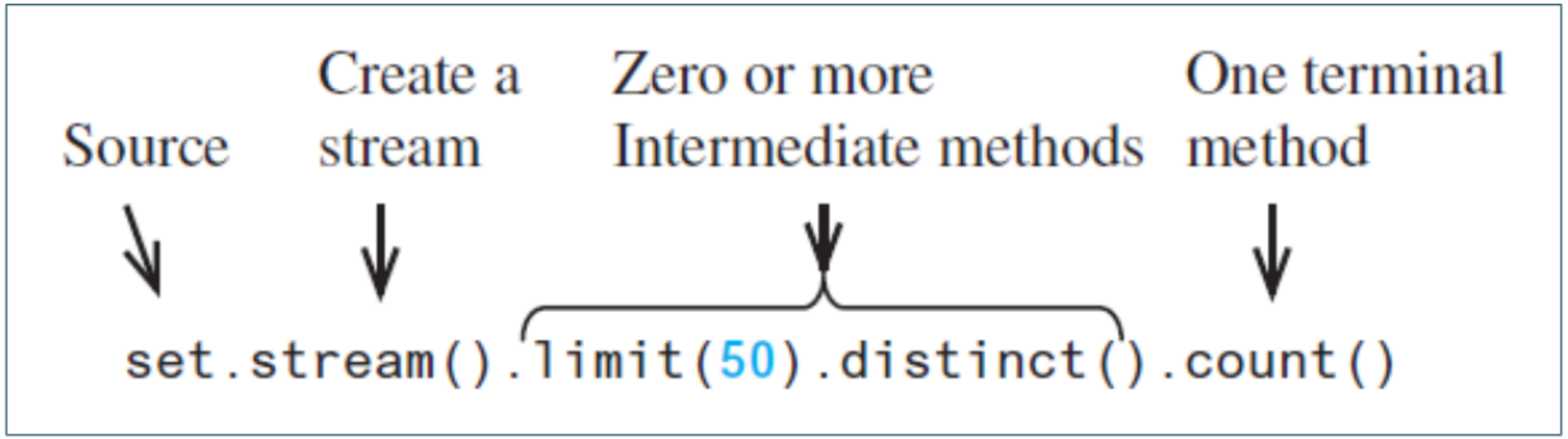
```java
//count the numbers that are greater than 5
List<Integer> numbers = List.of(1, 4, 7, 6, 2, 9, 7, 8);

long count = 0;
for (int number : numbers) {
    if (number > 5) {
        count++;
    }
}
System.out.println(count); // 5

long countS = numbers.stream()
        .filter(number -> number > 5)
        .count(); // 5
```

Stream Sources

- There are 95 methods in 23 classes that return a Stream
  - Many of them, though are intermediate operations in the Stream interface
- 71 methods in 15 classes can be used as practical Stream sources



Stream Sources

- Collection Interface
  - `stream()`
    - Provides a sequential stream of elements in the collection
  - `parallelStream()`
    - Provides a parallel stream of elements in the collection
    - Uses the fork-join framework for implementation

```
List<String> list = Arrays.asList("JAVA", "Spring", "Hibernate");
Stream<String> stream3 = list.stream();
```

Stream Sources

- Arrays Class
  - `stream()`
    - An array is a collection of data, so logical to be able to create a stream
    - Provides a sequential stream
    - overloaded methods for different types
      - double, int, long, Object

```
String[] arr = new String[] { "a", "b", "c" };
Stream<String> streamOfArrayFull = Arrays.stream(arr);
Stream<String> streamOfArrayPart = Arrays.stream(arr, 1, 3);
```

Stream Sources

- Files Class
  - `find(Path, BiPredicate, FileVisitOption)`
    - A stream of File references that match a given BiPredicate
  - `list(Path)`
    - A stream of entries from a given directory
  - `lines(Path)`
    - A stream of strings that are the lines read from a given file
  - `walk(Path, FileVisitOption)`
    - A stream of file references walking from a given Path

```
Path path = Paths.get("C:\\file.txt");
Stream<String> streamOfStrings = Files.lines(path);
```

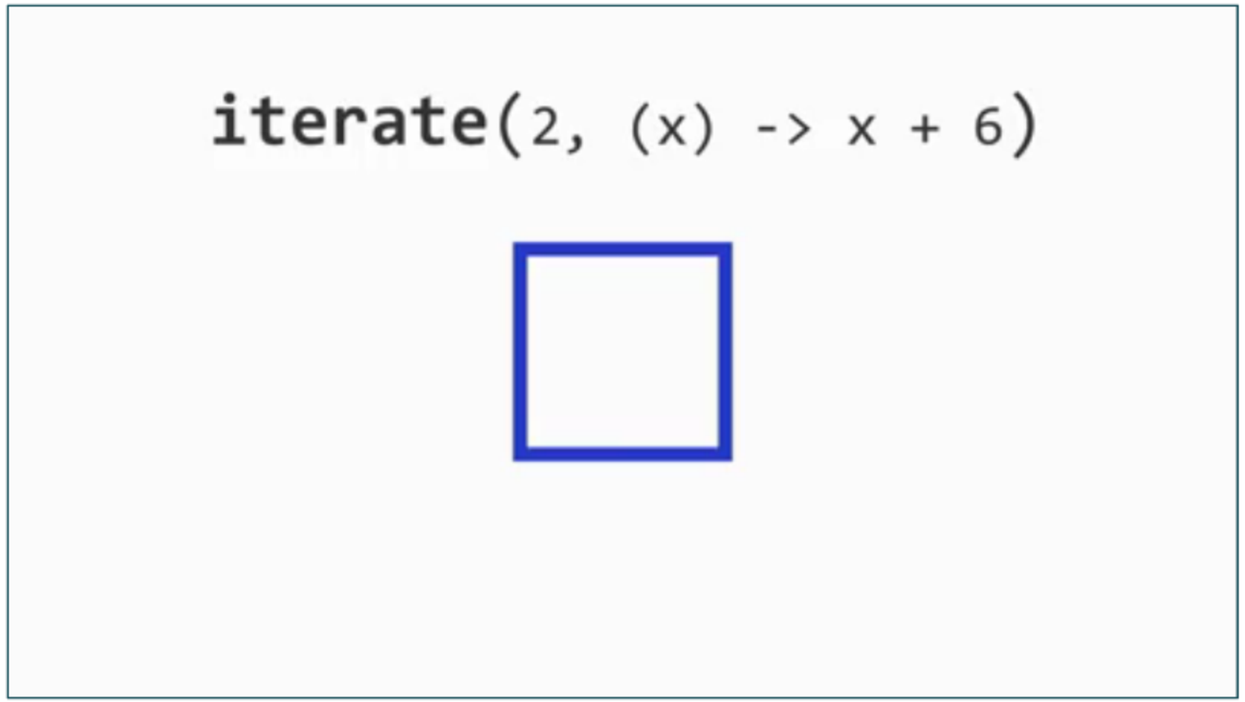
Stream Sources

- Stream Static Methods
  - `generate(IntSupplier)`,
  - `iterate(int, IntUnaryOperator)`
    - An infinite stream created by a given Supplier
    - iterate() uses a seed to start the stream

```
Stream<String> streamGenerated =
        Stream.generate(() -> "element").limit(10);
Stream<Integer> streamIterated =
        Stream.iterate(1, n -> n + 2).limit(5);
```

Stream Sources

```
Stream.iterate(2, x -> x + 6)
      .limit(6)
      .forEach(System.out::println);
// 2, 8, 14, 20, 26, 32
```



Stream Sources

- IntStream, DoubleStream, LongStream

  These interfaces are primitive specialisations of the Stream interface
  - `range(int, int)`,
  - `rangeClosed(int, int)`
    - A stream from a start to an end value (exclusive or inclusive)

```
IntStream intStream = IntStream.range(1, 3);
LongStream longStream = LongStream.rangeClosed(1, 3);
Random random = new Random();
DoubleStream doubleStream = random.doubles(3);
```

Stream Sources

- Stream Static Methods
  - `concat(Stream, Stream), empty()`
    - Concatenates two specified streams, returns an empty stream
  - `of(T... values)`
    - A stream that consists of the specified values

```
Stream<String> streamOfValues = Stream.of("a", "b", "c");

public Stream<String> streamOf(List<String> list) {
    return list == null || list.isEmpty()
            ? Stream.empty() : list.stream();
}
```

Duomenų apdorojimas naudojant „Stream API“

- Srautas (angl. Stream) - tai grandinė (angl. pipeline) sudaryta iš funkcijų. Kai duomenys iš srauto šaltinio pradeda judėti šia grandine, tai kiekvienos funkcijos įvykdymas gali pakeisti srauto turinį (objektų tipus, srauto elementų kiekį bei jų rūšiavimą). Srauto funkcijos yra skirstomos į dvi rūšis:
  - Tarpinės funkcijos (angl. Intermediate operations)
    - Atidėtai vykdomos (angl. Lazily executed)
    - Visada grąžina srautą
  - Galutinės funkcijos (angl. Terminal operations)
    - Grąžina konkretaus tipo objektus arba generuoja šalutinius efektus.
    - Vykdomos nedelsiant (angl. Eagerly executed)

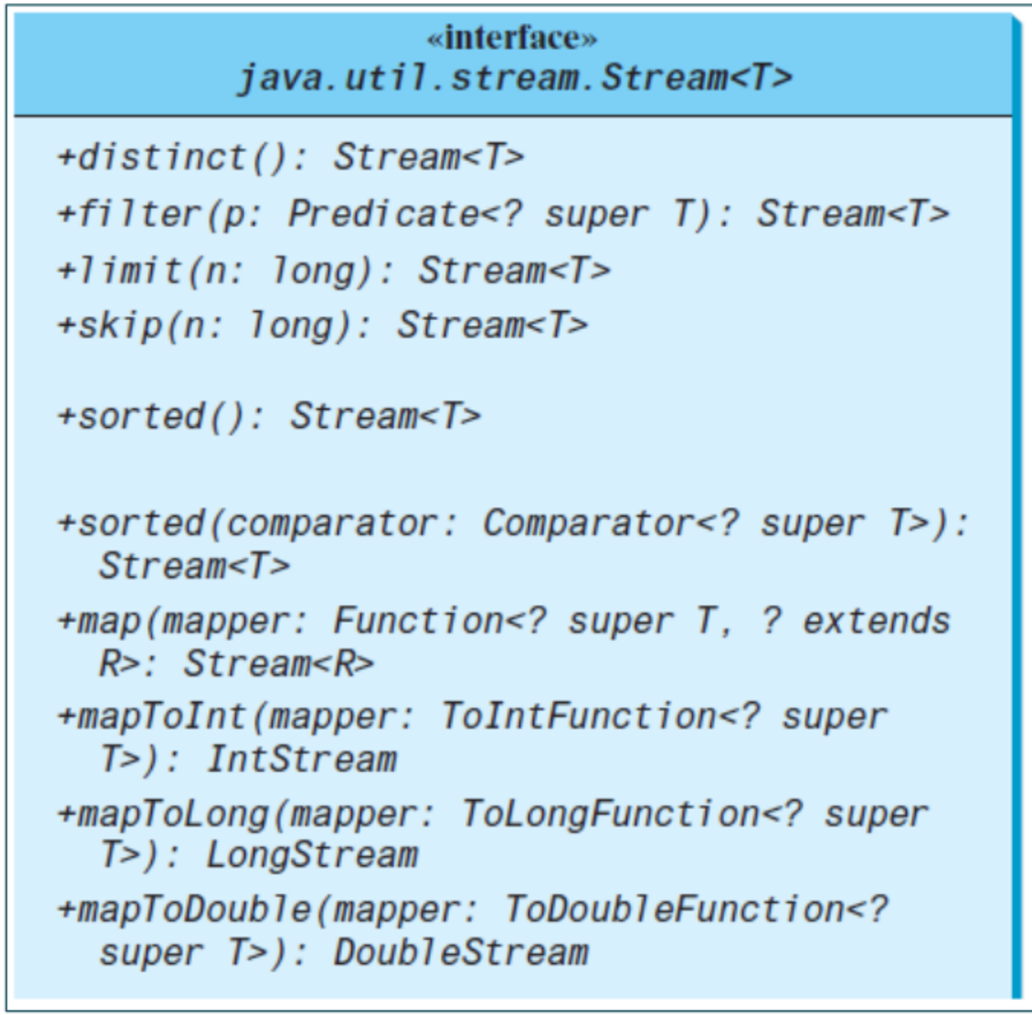
Method Types

- Intermediate methods
  - These are methods that produce other Streams. These methods don't get processed until there is some terminal method called.
- Terminal methods
  - After one of these methods is invoked, the Stream is considered consumed and no more operations can be performed on it.
  - These methods can do a side-effect (forEach) or produce a value (findFirst)
- Short-circuit methods
  - These methods cause the earlier intermediate methods to be processed only until the short-circuit method can be evaluated.
  - Short-circuit methods can be intermediate (limit, skip) or terminal (findFirst, allMatch)

Method Types

- Intermediate methods
  - map (and related mapToInt, flatMap, etc.), filter, distinct, sorted, peek, limit, skip, parallel, sequential, unordered
- Terminal methods
  - forEach, forEachOrdered, toArray, reduce, collect, min, max, count, anyMatch, allMatch, noneMatch, findFirst, findAny, iterator
- Short-circuit methods
  - anyMatch, allMatch, noneMatch, findFirst, findAny, limit, skip

Stream intermediate operations



Returns a stream consisting of distinct elements from this stream.

Returns a stream consisting of the elements matching the predicate.

Returns a stream consisting of the first n elements from this stream.

Returns a stream consisting of the remaining elements in this stream after discarding the first n elements.

Returns a stream consisting of the elements of this stream sorted in a natural order.

Returns a stream consisting of the elements of this stream sorted using the comparator.

Returns a stream consisting of the results of applying the function to the elements of this stream.

Returns an `IntStream` consisting of the results of applying the function to the elements of this stream.

Returns a `LongStream` consisting of the results of applying the function to the elements of this stream.

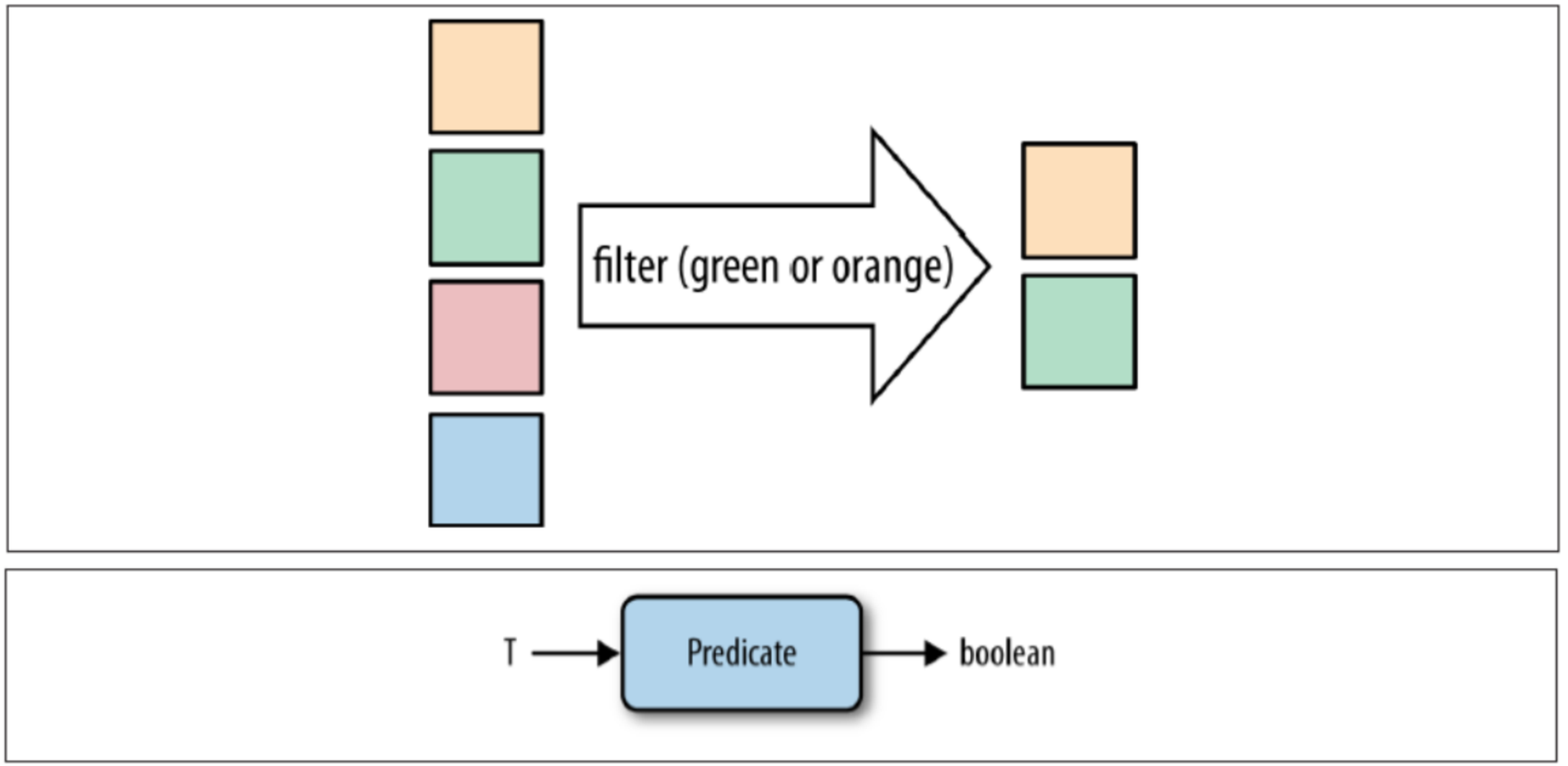
Returns a `DoubleStream` consisting of the results of applying the function to the elements of this stream.

Stream tarpinės operacijos

Metodas	Aprašymas
<R> Stream<R> **map**(Function<? super T, ? extends R> mapper)	Grąžinamas srautas sudarytas iš elementų, pritaikius funkciją kiekvienam iš esamo srauto elementų.
Stream<T> **filter**(Predicate<? super T> predicate)	Grąžinamas srautas suradytas iš elementų, kuriems predikato įvykdymas grąžina rezultatą “true”.
Stream<T> **distinct**()	Srautas sudarytas iš unikalių elementų.
Stream<T> **sorted**()	Srautas sudarytas iš surūšiuotų elementų (pagal natūralią tvarką)
Stream<T> **peek**(Consumer<? super T> action)	Grąžina tą patį srautą, tačiau leidžia atlikti kokią nors operaciją to srauto elementams.
Stream<T> **limit**(long maxSize)	Srautas sudarytas iš ne daugiau nei nurodytas kiekis elementų.

Stream tarpinės operacijos (filter)

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



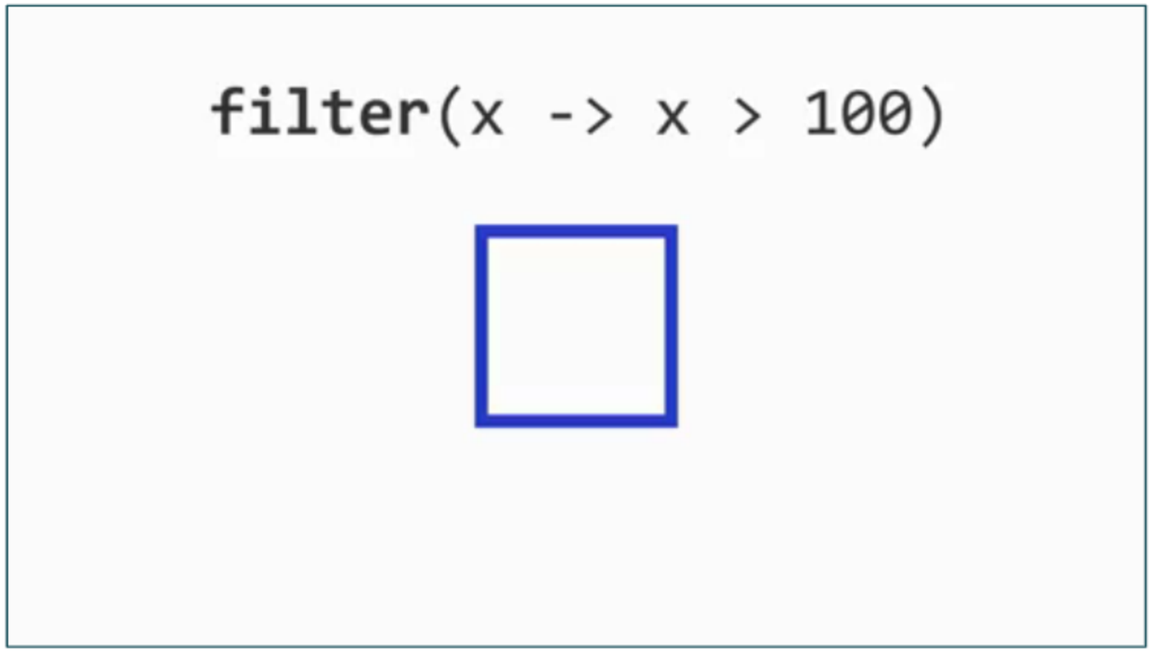
Stream tarpinės operacijos (filter)

```java
// Looping over a list and using an if statement
List<String> beginningWithNumbers = new ArrayList<>();
for (String value : Arrays.asList("a", "1abc", "abc1")) {
    if (Character.isDigit(value.charAt(0))) {
        beginningWithNumbers.add(value);
    }
}
```

```java
// Functional style
List<String> beginningWithNumbers = Stream.of("a", "1abc", "abc1")
        .filter(value -> Character.isDigit(value.charAt(0)))
        .collect(Collectors.toList());
```

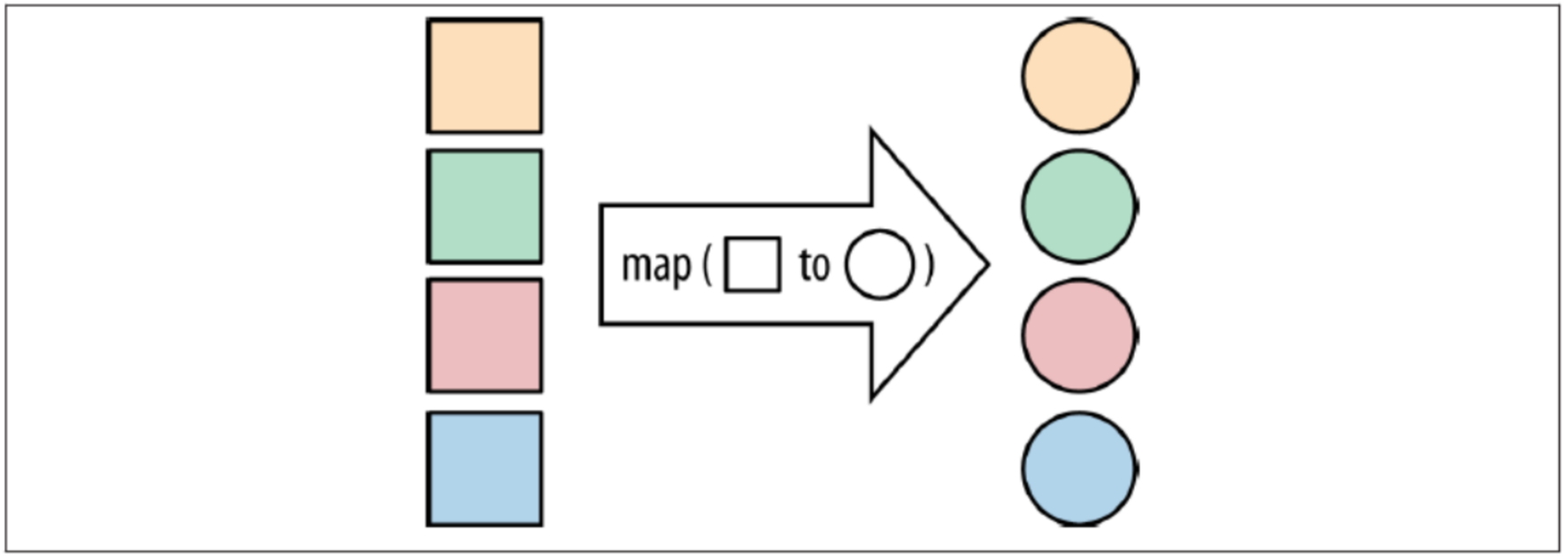
Stream tarpinės operacijos (filter)

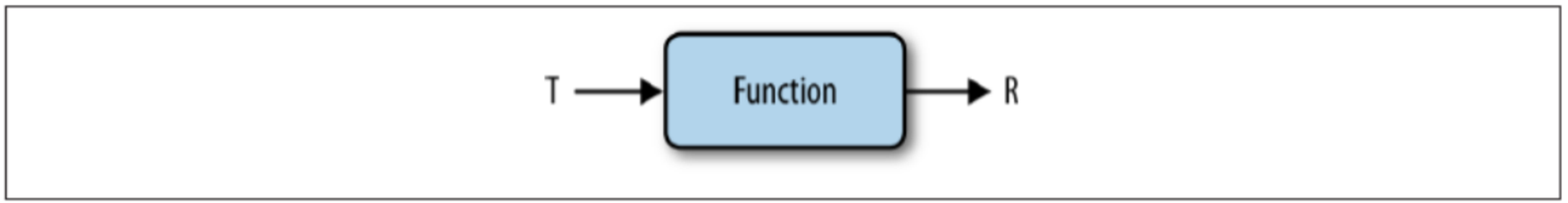
```
Stream.of(120, 410, 85, 32, 314, 12)
    .filter(x -> x > 100)
    .forEach(System.out::println);
// 120, 410, 314
```



Stream tarpinės operacijos (map)

- Returns a stream consisting of the results of applying the given function to the elements of this stream.





Stream tarpinės operacijos (map)

```java
//Converting strings to uppercase equivalents using a for loop
List<String> collected = new ArrayList<>();
for (String string : Arrays.asList("a", "b", "hello")) {
    String uppercaseString = string.toUpperCase();
    collected.add(uppercaseString);
}
```

```java
//Converting strings to uppercase equivalents using map
List<String> collected = Stream.of("a", "b", "hello")
        .map(string -> string.toUpperCase())
        .collect(Collectors.toList());
```

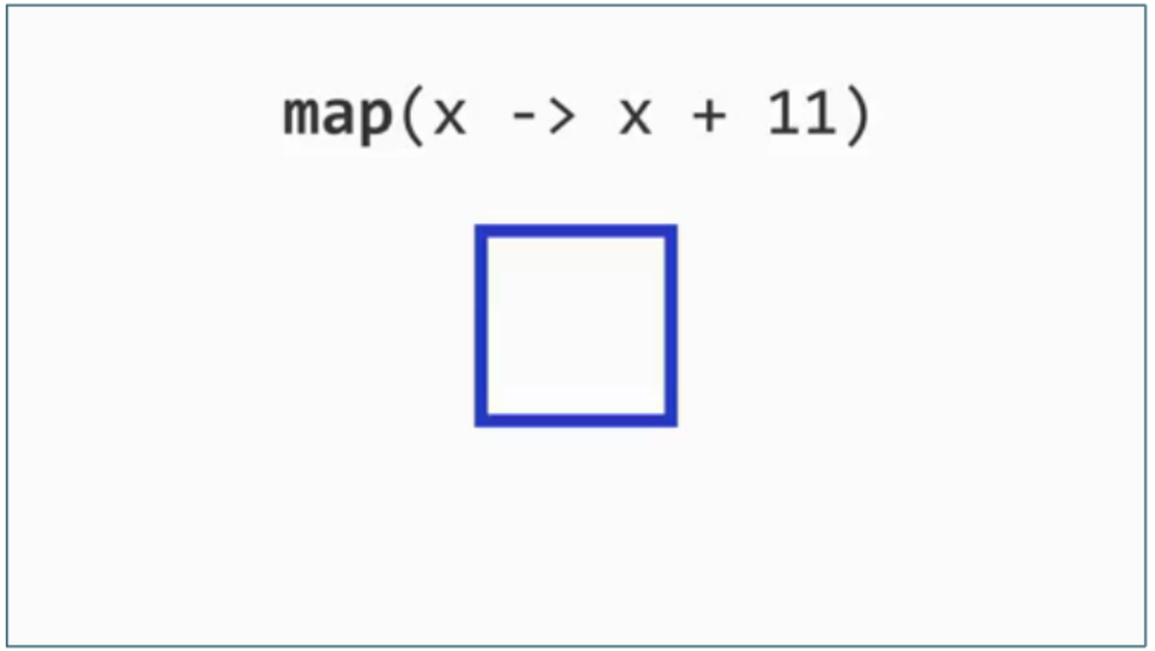
Stream tarpinės operacijos (map)

```
public class Job {
    private String title;
    private String description;
    private double salary;
    // getters and setters
}
```

```
List<String> titles = jobs.stream()
        .map(Job::getTitle) // get title of each job
        .collect(Collectors.toList()); //collect titles
                                       //to a new list
```

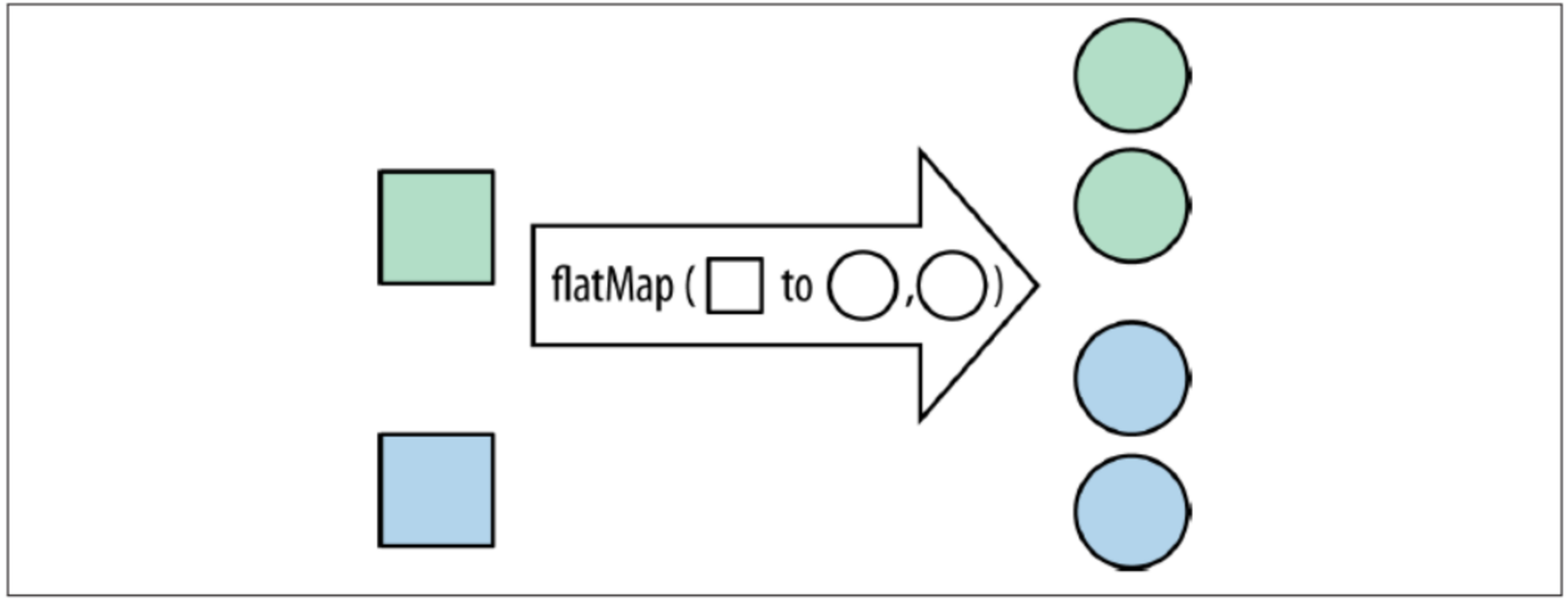
Stream tarpinės operacijos (map)

```
Stream.of(120, 410, 85, 32, 314, 12)
      .map(x -> x + 11)
      .forEach(System.out::println);
// 131, 421, 96, 43, 325, 23
```



Stream tarpinės operacijos (flatMap)

- flatMap lets you replace a value with a Stream and concatenates all the streams together.




```
List<Integer> together = Stream.of(Arrays.asList(1, 2), Arrays.asList(3, 4))
        .flatMap(numbers -> numbers.stream())
        .collect(Collectors.toList());
```

Stream tarpinės operacijos (flatMap)

```
List<Book> javaBooks = List.of(
        new Book("Effective Java", 2017, List.of("Joshua Bloch")),
        new Book("Algorithms", 2011, List.of("Robert Sedgewick", "Kevin Wayne")),
        new Book("Clean code", 2014, List.of("Robert Martin"))
);
```

```
List<String> authors = javaBooks.stream()
        .flatMap(book -> book.getAuthors().stream())
        .distinct()
        .collect(Collectors.toList());
```

```
[Joshua Bloch, Robert Sedgewick, Kevin Wayne, Robert Martin]
```

Stream terminal operations

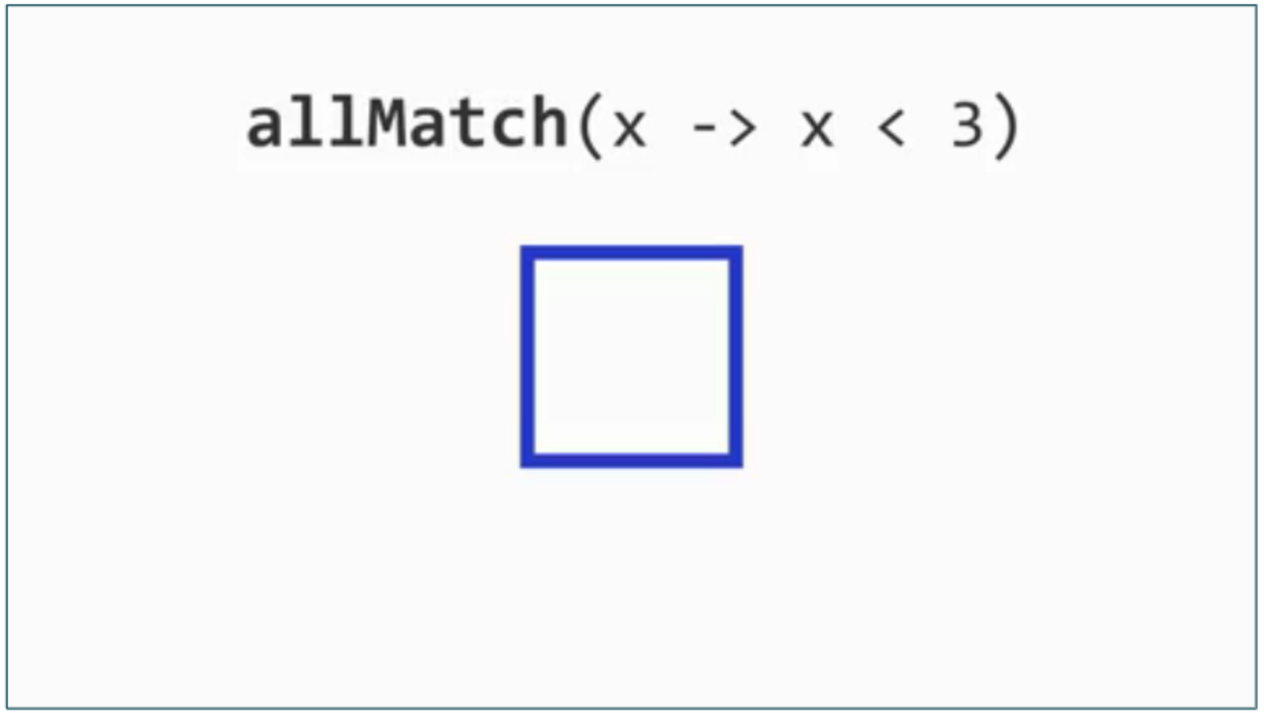
Metodas	Aprašymas
T **reduce**(T identity, BinaryOperator<T> accumulator) Optional<T> **reduce**(BinaryOperator<T> accumulator)	Grąžina akumuliuotą reikšmę, kur rezultato tipas toks pat, kaip ir elementų iš kurių sudarytas srautas.
<R, A> R **collect**(Collector<? super T, A, R> collector)	Atlieka elementų surinkimo operaciją.
void **forEach**(Consumer<? super T> action)	Atlieka tam tikrą veiksmą kiekvienam iš srauto elementų.
Optional<T> **min**(Comparator<? super T> comparator)	Grąžina minimalią reikšmę (minimalumas nustatomas pagal nurodyta Comparator interfeiso realizaciją).
Optional<T> **max**(Comparator<? super T> comparator)	Grąžina maksimalią srauto reikšmę (maksimalumas nustatomas pagal Comparator interfeiso realizacija)

Stream terminal operations

`+count(): long`	Returns the number of elements in this stream.
`+max(c: Comparator<? super T>): Optional<T>`	Returns the maximum element in this stream based on the comparator.
`+min(c: Comparator<? super T>): Optional<T>`	Returns the minimum element in this stream based on the comparator.
`+findFirst(): Optional<T>`	Returns the first element from this stream.
`+findAny(): Optional<T>`	Returns any element from this stream.
`+allMatch(p: Predicate<? super T): boolean`	Returns true if all the elements in this stream match the predicate.
`+anyMatch(p: Predicate<? super T): boolean`	Returns true if one element in this stream matches the predicate.
`+noneMatch(p: Predicate<? super T): boolean`	Returns true if no element in this stream matches the predicate.
`+forEach(action: Consumer<? super T>): void`	Performs an action for each element of this stream.
`+reduce(accumulator: BinaryOperator<T>): Optional<T>`	Reduces the elements in the stream to a value using the identity and an associative accumulation function. Return an Optional describing the reduced value.
`+reduce(identity: T, accumulator: BinaryOperator<T>): T`	Reduces the elements in the stream to a value using the identity and an associative accumulation function. Return the reduced value.
`+collect(collector: <? super <T, A, R>>): R`	Performs a mutable reduction operation on the elements of this stream using a `Collector`.
`+toArray(): Object[]`	Returns an array consisting of the elements in this stream.

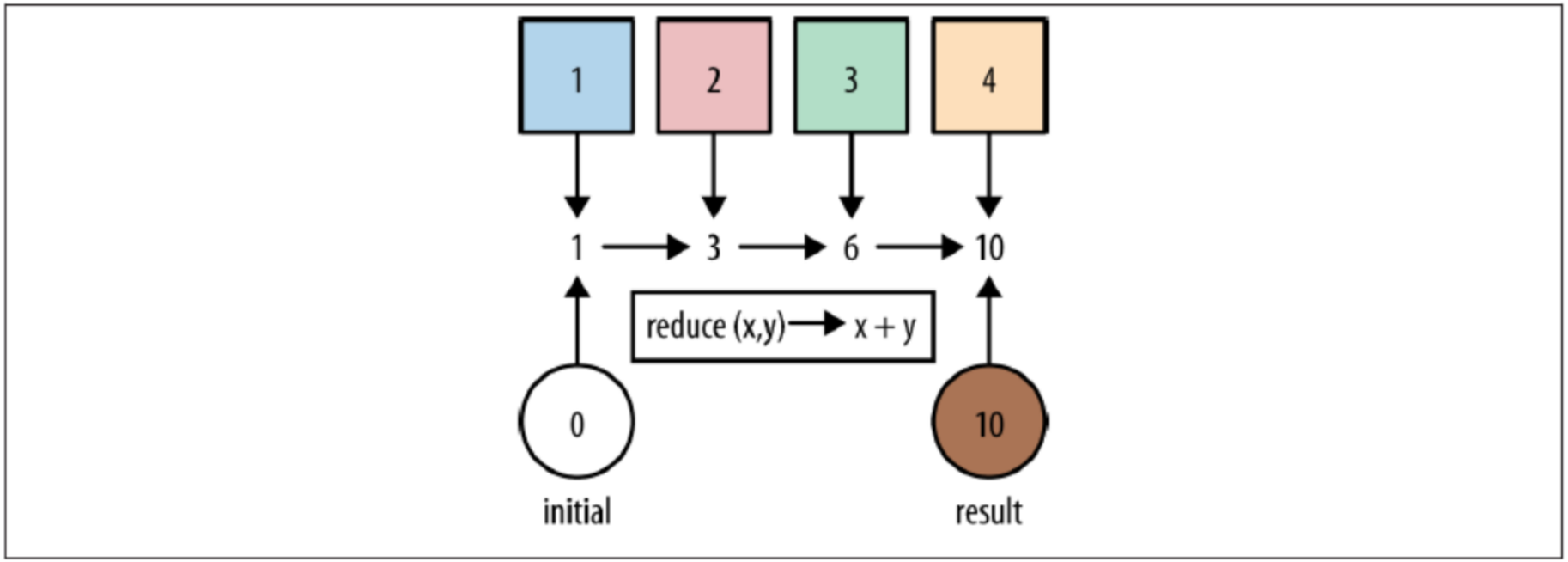
Stream terminal operations (allMatch)

```
boolean result = Stream.of(1, 2, 3, 4, 5)
        .allMatch(x -> x < 3);
// result: false
```



Stream terminal operations (reduce)

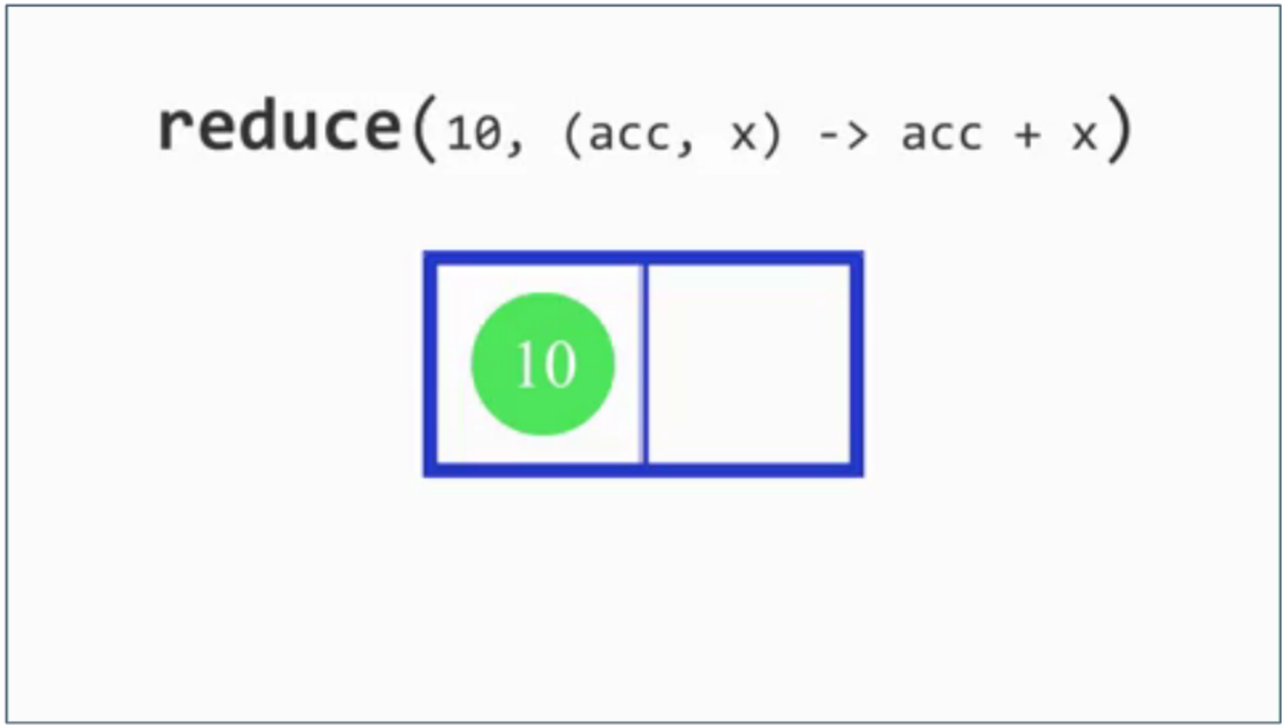
- Use the reduce operation when you've got a collection of values and you want to generate a single result.




```
int count = Stream.of(1, 2, 3)
        .reduce(0, (acc, element) -> acc + element);
```

Stream terminal operations (reduce)

```
int sum = Stream.of(1, 2, 3, 4, 5)
          .reduce(10, (acc, x) -> acc + x);
// sum: 25
```



Collectors

```
List<Account> l = accountStream.collect(Collectors.toList());
Set<Account> s = accountStream.collect(Collectors.toSet());
//If you need more control over producing collections
LinkedList<Account> ll = accountStream
        .collect(Collectors.toCollection(LinkedList::new));
```

```
//Since Java 16. The returned List is unmodifiable.
List<Account> u = accountStream.toList();
```

Class Optional<T>

- Optional either stores a T or stores nothing.
- allows programmers to avoid null references that may lead to NPE;
- reduces the boilerplate code for checking null;
- provides a rich set of functional methods.
- Making an Optional (usually done behind the scenes by built-in methods):

```
Optional<String> empty = Optional.empty();
String name = "java";
Optional<String> opt = Optional.of(name);
String name2 = null;
Optional<String> opt2 = Optional.ofNullable(name);
```

Class Optional<T>

- Most common operations on an Optional:

```
//returns true if value is present
opt.isPresent();
//invokes consumer if value is present, otherwise do nothing
opt.ifPresent(s -> System.out.println(s.length()));
//returns value if present or throws exception
opt.get();
//returns value if present or returns other
opt.orElse("default");
//returns value if present or calls function
opt.orElseGet(()->getMessage());
//throws an exception if value is not present
opt.orElseThrow(IllegalArgumentException::new);
```

Stream pavyzdžiai

```
// Legacy code finding names of tracks over a minute in length
public Set<String> findLongTracks(List<Album> albums) {
    Set<String> trackNames = new HashSet<>();
    for (Album album : albums) {
        for (Track track : album.getTrackList()) {
            if (track.getLength() > 60) {
                String name = track.getName();
                trackNames.add(name);
            }
        }
    }
    return trackNames;
}
```

Stream pavyzdžiai

```
// Finding names of tracks over a minute in length
public Set<String> findLongTracks2(List<Album> albums) {
    return albums.stream()
            .flatMap(album -> album.getTracks())
            .filter(track -> track.getLength() > 60)
            .map(track -> track.getName())
            .collect(Collectors.toSet());
}
```

Stream pavyzdžiai

```
//Kiekvienas skaičius dauginamas iš 2, konvertuojamas į String
//tipo objektą bei akumuliuojamas į simbolių eilutę,
//kiekvieną skaičių atskiriant kableliu.
final List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
final String result = numbers.stream()
                             .map((num) -> Integer.toString(num * 2))
                             .collect(Collectors.joining(","));
System.out.println(result);
```

```
2,4,6,8,10,12,14,16
```

Stream pavyzdžiai

```
// Atspausdinama lygiai 5 didėjimo tvarka surūšiuotų
// atsitiktinai parinktų skaičių, kurie yra lyginiai.
Stream.generate(() -> (int) (Math.random() * Integer.MAX_VALUE))
      .filter((num) -> num % 2 == 0)
      .distinct()
      .limit(5)
      .sorted()
      .forEach(System.out::println);
```

```
71120548
668827488
798701008
1327420956
1439082596
```

Stream pavyzdžiai

```
// Tas pats kaip ankstesnėje skaidrėje, tik papildomai išspausdinant
// informaciją apie skaičius prieš ir po funkcijos “filter” atlikimo.
Stream.generate(() -> (int) (Math.random() * Integer.MAX_VALUE))
      .peek((num) -> System.out.format("pre-filter %d\n", num))
      .filter((num) -> num % 2 == 0)
      .peek((num) -> System.out.format("post-filter %d\n", num))
      .distinct()
      .limit(5).sorted()
      .forEach(System.out::println);
```

```
pre-filter 1642239545
pre-filter 2126829424
post-filter 2126829424
pre-filter 1134937172
post-filter 1134937172
pre-filter 1579114538
post-filter 1579114538
pre-filter 860244282
post-filter 860244282
```

```
pre-filter 661566451
pre-filter 44677504
post-filter 44677504
44677504
860244282
1134937172
1579114538
2126829424
```

Stream pavyzdžiai

```
// Filtruojamas sąrašas pagal predikatą, kuris sukuriams metode.
// Demonstruojama, kaip galima pernaudoti lambda išraiškas.
// Išraiškai, sukurtai metode yra prieinami to metodo argumentai,
// su kuriais jis buvo iškviestas (angl. closure).
List<String> names =
        Arrays.asList("Darius", "Jonas", "Karolis", "Andrius");
names.stream()
     .filter(predicate("n"))
     .forEach(System.out::println);
```

```
private static Predicate<String> predicate(final String letter) {
    return word -> word.contains(letter);
```

```
Jonas
Andrius
```

Stream pavyzdžiai

```
    // Randama pirmus 10 pirminių skaičių.
    // Dirbama su begaliniais srautais.
    Stream.iterate(1, x -> x + 1)
          .filter(x -> isPrime(x))
          .limit(10)
          .forEach(System.out::println);
```

```
2
3
5
7
11
13
17
19
23
29
```

```
private static boolean isPrime(int number) {
    if (number == 1) {
        return false;
    }
return IntStream.rangeClosed(2, (int) Math.sqrt(number))
                .noneMatch(num -> number % num == 0);
}
```

Stream pavyzdžiai

```
List<String> names =
  Arrays.asList("Egidijus", "Aloyzas", "Dainius", "Karolis");
names.stream()
     .filter((name) -> length(name) == 7)
     .map((name) -> toUpperCase(name))
     .findFirst()
     .ifPresent(System.out::println);
```

```
private static int length(final String text) {
    System.out.format("calling length(%s)\n", text);
    return text.length();
}
private static String toUpperCase(final String text) {
    System.out.format("calling toUpperCase(%s)\n", text);
    return text.toUpperCase();
}
```

```
calling length(Egidijus)
calling length(Aloyzas)
calling toUpperCase(Aloyzas)
ALOYZAS
```

Stream pavyzdžiai

```
//Given very large file of words of various lengths in
//mixed case with possible repeats, create sorted
//uppercase file of n-letter words

List<String> words = Files.lines(Paths.get(inputFileName))
        .filter(s -> s.length() == n)
        .map(String::toUpperCase)
        .distinct()
        .sorted()
        .collect(Collectors.toList());
Files.write(Paths.get(outputFileName), words,
        Charset.defaultCharset());
```

Stream pavyzdžiai

```
// Išspausdinamas surūšiuotas sąrašas java failų.

Path start = Paths.get(".");
int maxDepth = Integer.MAX_VALUE;
try (Stream<Path> stream = Files.walk(start, maxDepth)) {
    stream.map(String::valueOf)
          .filter(path -> path.endsWith(".java"))
          .sorted()
          .forEach(System.out::println);
}
```

Stream pavyzdžiai

```
List<Person> presidents = new ArrayList<>();
try {
    // reading the "presidents.txt" file line by line
    Files.lines(Paths.get("presidents.txt"))
        // splitting the row into parts on the ";" character
        .map(row -> row.split(";"))
        // deleting the split rows that have less than two parts
        .filter(parts -> parts.length >= 2)
        // creating persons from the parts
        .map(parts -> new Person(parts[0], Integer.valueOf(parts[1])))
        // and finally add the persons to the list
        .forEach(person -> presidents.add(person));
} catch (Exception e) {
    System.out.println("Error: " + e.getMessage());
}
```