Java Streams

AMOL R PATIL - 9822291613

Optional Class

- A container object which may or may not contain a non-null value. If a value is present, isPresent() will return true and get() will return the value.

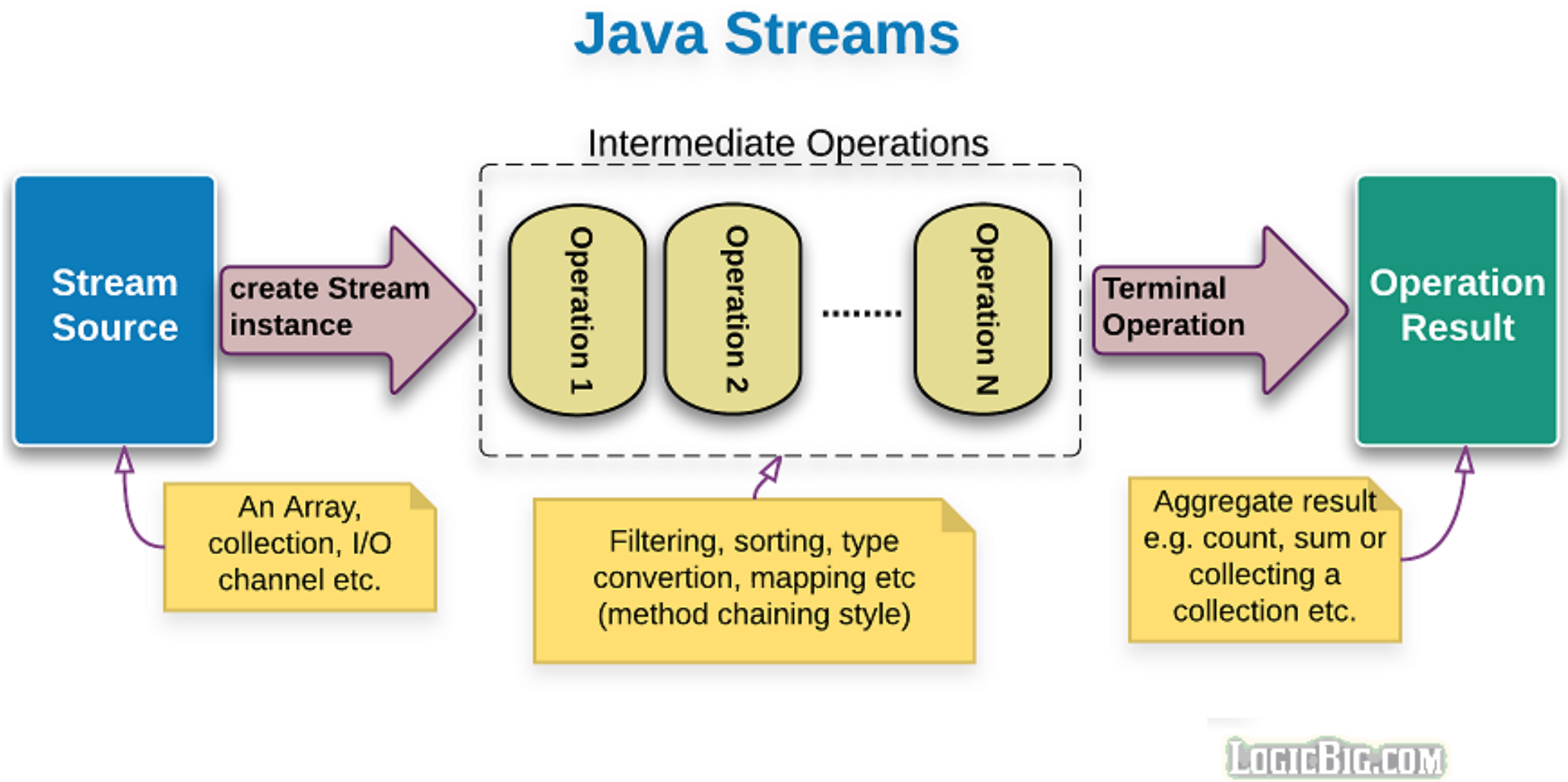
- It was introduced as a way to help reduce the number of NullPointerExceptions that occur in Java code. It is a part of the java.util package and was added to Java as part of Java 8.

- Optional.ofNullable (T)
- orElse
- orThrow

Streams

- Streams are wrappers around a data source, allowing us to operate with that data source and making bulk processing convenient and fast.
- A stream does not store data and, in that sense, is not a data structure. It also never modifies the underlying data source.
- This functionality – java.util.stream – supports functional-style operations on streams of elements, such as map-reduce transformations on collections.
- A Java stream does not involve a change in the input data structure.
- A Java stream does not alter the source. Instead, it generates output by pipelining methods accordingly.
- Java streams undergo intermediate and terminal operations
- In Java streams, intermediate operations are pipelined and lazily evaluated. They are terminated by terminal functions. This forms the basic format of using a Java stream.

Working of Streams



Creating Streams

- Create a stream from Collection

  The Java Collection framework provides two methods, stream() and parallelStream(), to create a sequential and parallel stream from any collection, respectively.

- Create a stream from specified values

  We can create a sequential stream from the specified values using the Stream.of() method.

- Using Arrays.stream() method

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

Creating Streams

- Create a stream using builder

```
Stream<String> streamBuilder =
      Stream.<String>builder().add("a").add("b").add("c").build();
```

- Create a stream using generator [using supplier]

```
Stream<String> streamGenerated =
        Stream.generate(() -> (new Random()).nextInt(100)).limit(10);
```

- Create a stream using iterate [infinite]

```
Stream<Integer> streamIterated =
      Stream.iterate(40, n -> n + 2).limit(20);
```

Primitive Streams

- Java 8 offers the possibility to create streams out of three primitive types: int, long and double.

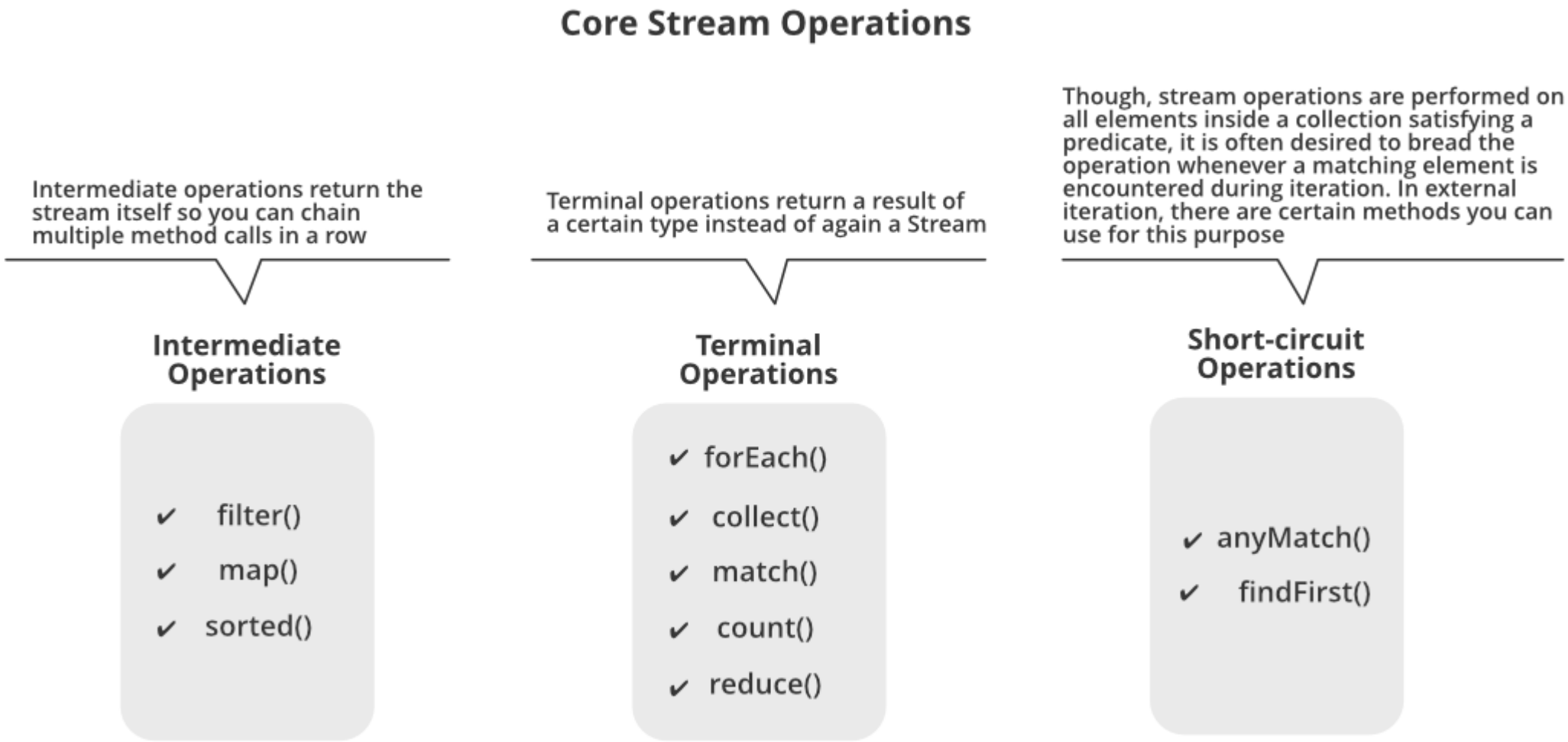
- As Stream<T> is a generic interface, and there is no way to use primitives as a type parameter with generics, three new special interfaces were created: IntStream, LongStream, DoubleStream.

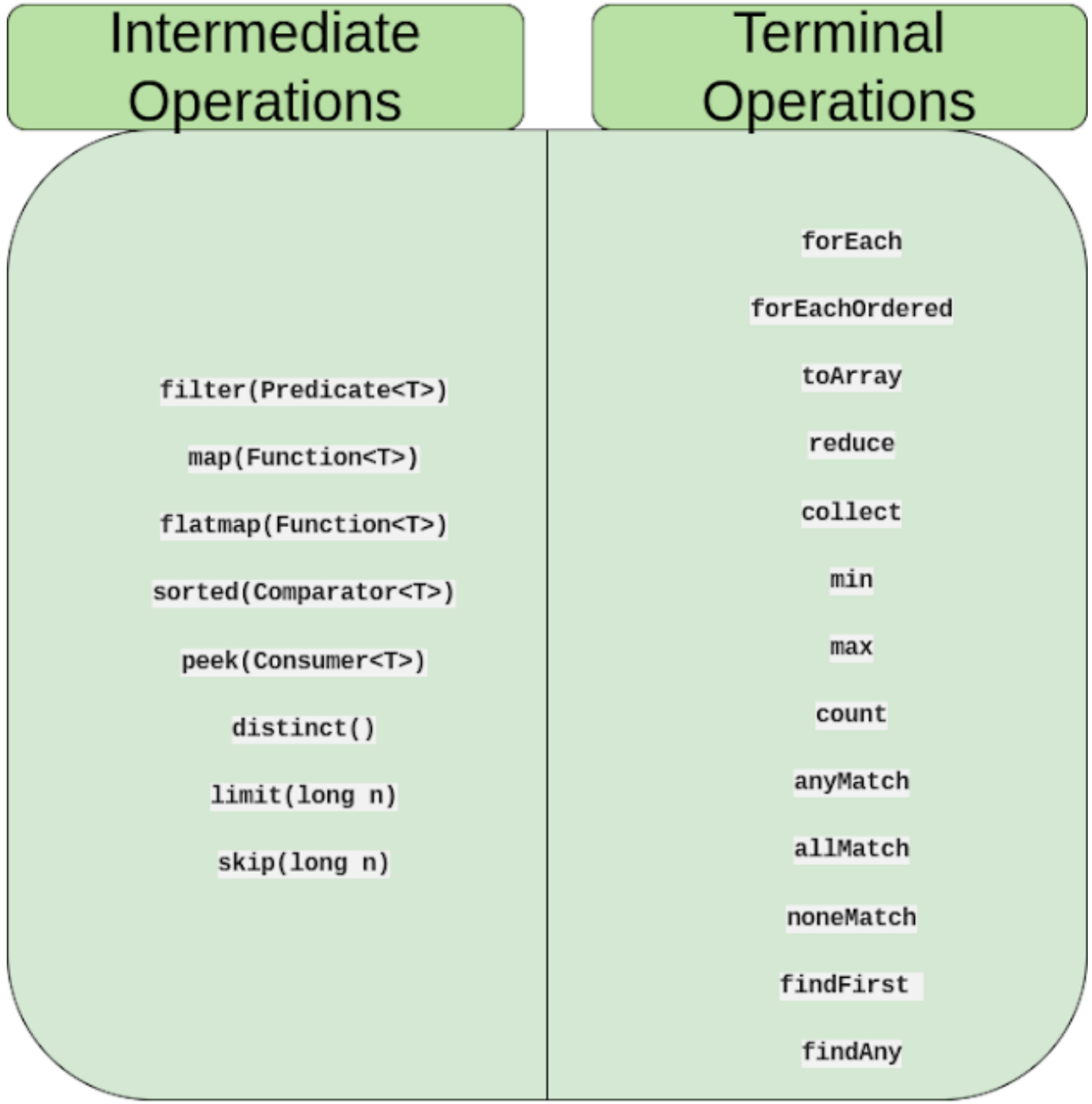
```
IntStream intStream = IntStream.range(1, 3);
LongStream longStream = LongStream.rangeClosed(1, 3);

Random random = new Random();
DoubleStream doubleStream = random.doubles(3);
```

Stream of Strings

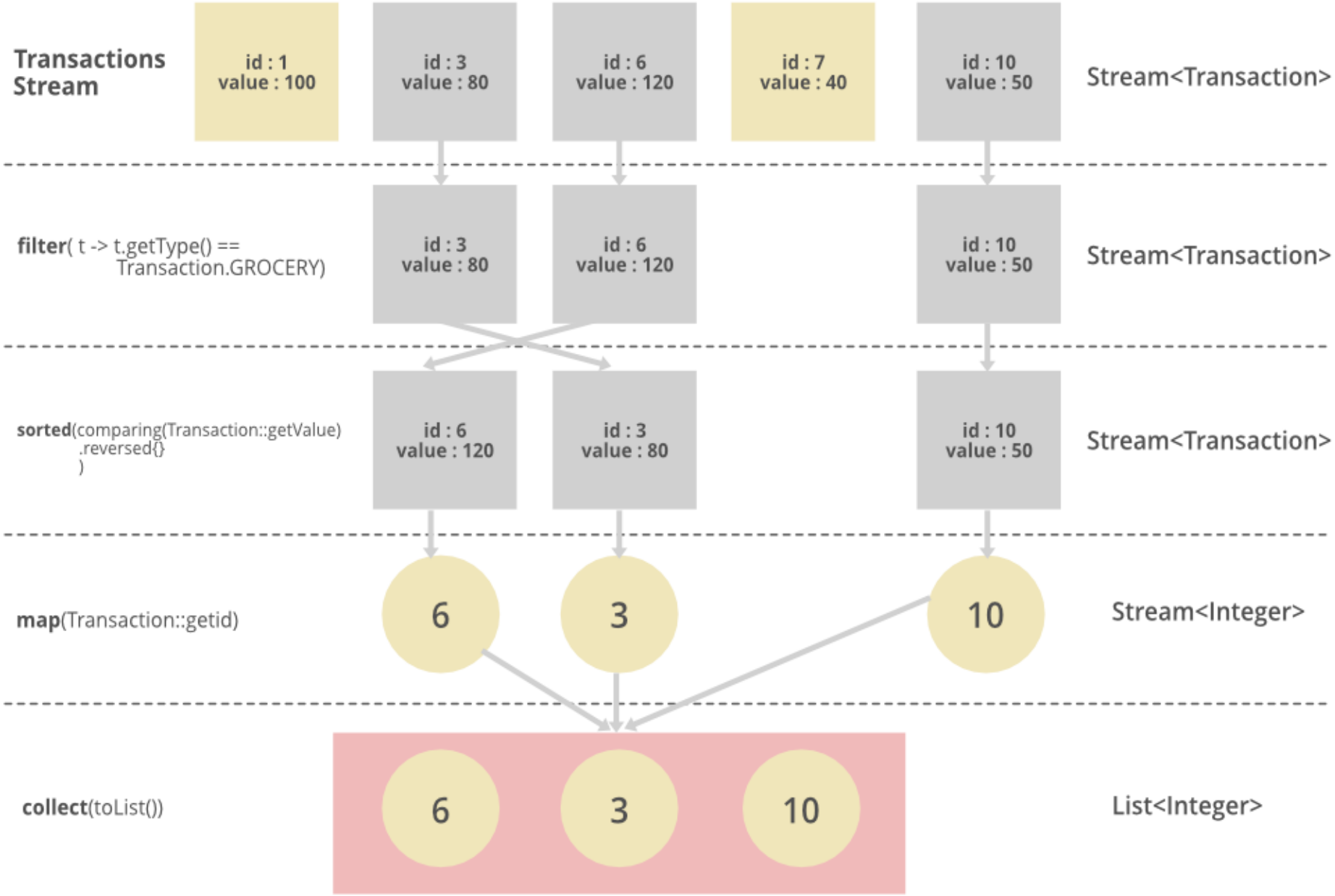
```
IntStream streamOfChars = "abc".chars();

Stream<String> streamOfString =
  Pattern.compile(", ").splitAsStream("a, b, c");
```

Stream Intermediate and Terminal Operations




Intermediate Operations
Terminal Operations
filter(Predicate<T>)
map(Function<T>)
flatmap(Function<T>)
sorted(Comparator<T>)
peek(Consumer<T>)
distinct()
limit(long n)
skip(long n)
forEach
forEachOrdered
toArray
reduce
collect
min
max
count
anyMatch
allMatch
noneMatch
findFirst
findAny

Real Life Example



Java Stream flatMap()

- The Stream flatMap() method is used to flatten a Stream of collections to a Stream of objects. The objects are combined from all the collections in the original Stream.
- Stream.flatMap() helps in converting Stream<Collection<T>> to Stream<T>.
- flatMap() = Flattening + map()
- flatMap() is an intermediate operation and return a new Stream.

Java Stream flatMap()

Merging Lists into a Single List

```
List<Integer> list1 = Arrays.asList(1,2,3);
List<Integer> list2 = Arrays.asList(4,5,6);
List<Integer> list3 = Arrays.asList(7,8,9);

List<List<Integer>> listOfLists = Arrays.asList(list1, list2, list3);

List<Integer> listOfAllIntegers = listOfLists.stream()
          .flatMap(x -> x.stream())
          .collect(Collectors.toList());

System.out.println(listOfAllIntegers);
```

Primitive Streams

The stream API has inbuilt support for representing primitive streams using the following specialized classes. All these classes support the sequential and parallel aggregate operations on stream items.

- IntStream : represents sequence of primitive int-valued elements.
- LongStream : represents sequence of primitive long-valued elements.
- DoubleStream : represents sequence of primitive double-valued elements.

```
IntStream stream = IntStream.of(1, 2, 3, 4, 5);
LongStream stream = LongStream.of(1, 2, 3, 4, 5);
DoubleStream stream = DoubleStream.of(1.0, 2.0, 3.0, 4.0, 5.0);
```

Primitive Streams

The stream API has inbuilt support for representing primitive streams using the following specialized classes. All these classes support the sequential and parallel aggregate operations on stream items.

- IntStream : represents sequence of primitive int-valued elements.
- LongStream : represents sequence of primitive long-valued elements.
- DoubleStream : represents sequence of primitive double-valued elements.

```
IntStream stream = IntStream.of(1, 2, 3, 4, 5);
LongStream stream = LongStream.of(1, 2, 3, 4, 5);
DoubleStream stream = DoubleStream.of(1.0, 2.0, 3.0, 4.0, 5.0);


 IntStream stream = IntStream.range(1, 10);  //1,2,3,4,5,6,7,8,9
 LongStream stream = LongStream.range(10, 100);
```

Creating Primitive Streams

```
IntStream stream = IntStream.of(1, 2, 3, 4, 5);
LongStream stream = LongStream.of(1, 2, 3, 4, 5);
DoubleStream stream = DoubleStream.of(1.0, 2.0, 3.0, 4.0, 5.0);
```

```
IntStream stream = IntStream.range(1, 10);  //1,2,3,4,5,6,7,8,9
LongStream stream = LongStream.range(10, 100);
```

```
List<Integer> integerList = List.of(1, 2, 3, 4, 5);
IntStream stream = integerList.stream().mapToInt(i -> i);

Stream<Employee> streamOfEmployees = getEmployeeStream();
DoubleStream stream = streamOfEmployees.mapToDouble(e -> e.getSalary());
```

Summary Statistics

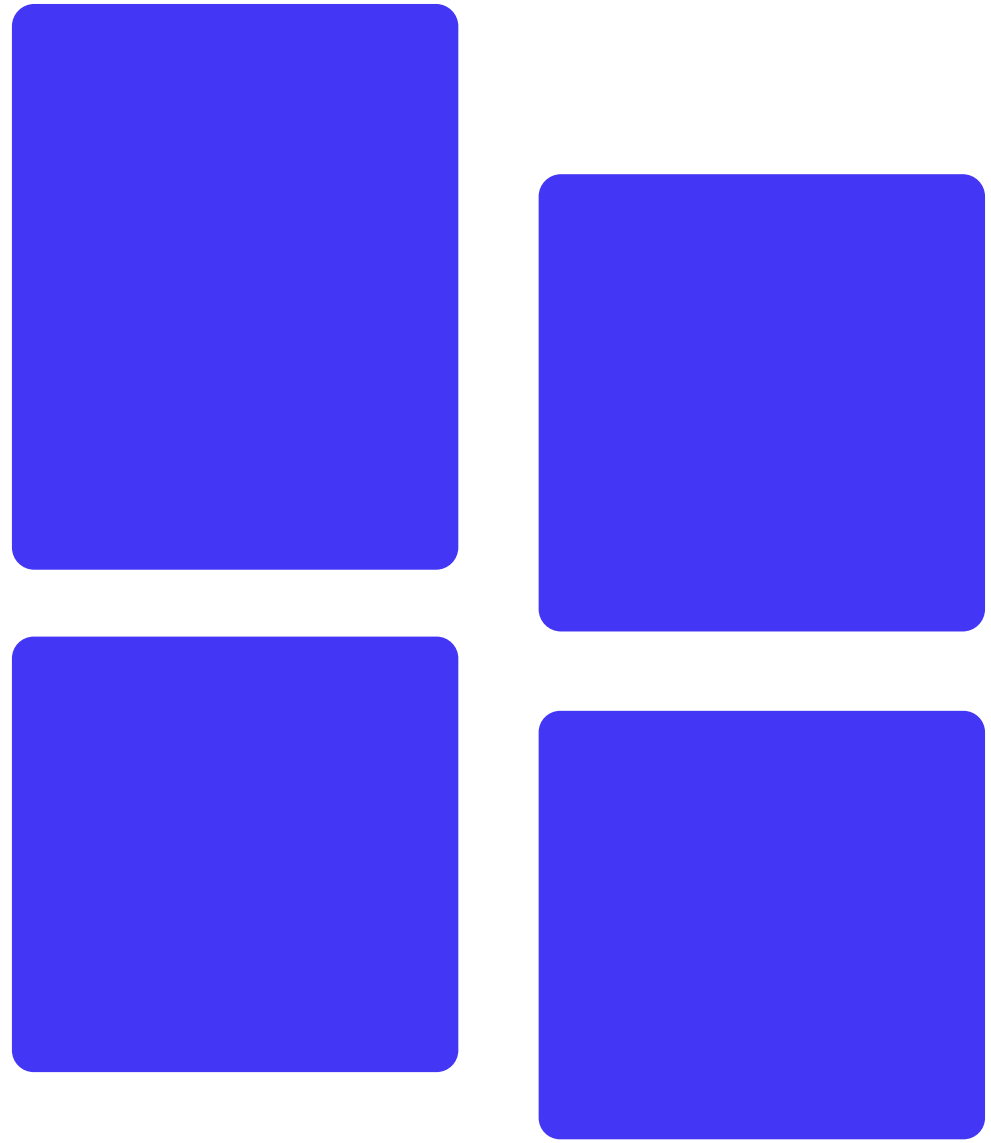
```
int max = IntStream.of(10, 18, 12, 70, 5)
  .max()
  .getAsInt();

double avg = IntStream.of(1, 2, 3, 4, 5)
  .average()
  .getAsDouble();

int sum = IntStream.range(1, 10)
  .sum();
```

```
IntSummaryStatistics summary = IntStream.of(10, 18, 12, 70, 5)
    .summaryStatistics();

int max = summary.getMax();
```

Summary Statistics

```
int max = IntStream.of(10, 18, 12, 70, 5)
  .max()
  .getAsInt();

double avg = IntStream.of(1, 2, 3, 4, 5)
  .average()
  .getAsDouble();

int sum = IntStream.range(1, 10)
  .sum();
```

```
IntSummaryStatistics summary = IntStream.of(10, 18, 12, 70, 5)
    .summaryStatistics();

int max = summary.getMax();
```



CopyOnWriteArrayList class

Java CopyOnWriteArrayList is a thread-safe variant of ArrayList in which all mutative operations (add, set, and so on) are implemented by making a fresh copy of the underlying array.

- CopyOnWriteArrayList class implement **List and RandomAccess interfaces** and thus provide all functionalities available in ArrayList class.
- Using CopyOnWriteArrayList is **costly for update operations, because each mutation creates a cloned copy of underlying array and add/update element to it.**
- It is **thread-safe version of ArrayList**. Each thread accessing the list sees its own version of snapshot of backing array created while initializing the iterator for this list.
- Because it gets snapshot of underlying array while creating iterator, **it does not throw ConcurrentModificationException.**
- Mutation operations on iterators (remove, set, and add) are not supported. These methods throw UnsupportedOperationException.
- CopyOnWriteArrayList is a concurrent replacement for a synchronized List and offers better concurrency when iterations outnumber mutations.
- It allows duplicate elements and heterogeneous Objects (use generics to get compile time errors).
- Because it creates a new copy of array everytime iterator is created, performance is slower than ArrayList.

ConcurrentHashMap class

- The underlined data structure for ConcurrentHashMap is Hashtable.

- ConcurrentHashMap class is thread-safe i.e. multiple threads can operate on a single object without any complications.

- At a time any number of threads are applicable for a read operation without locking the ConcurrentHashMap object which is not there in HashMap.

- In ConcurrentHashMap, the Object is divided into a number of segments according to the concurrency level.
- The default concurrency-level of ConcurrentHashMap is 16.
- In ConcurrentHashMap, at a time any number of threads can perform retrieval operation but for updated in the object, the thread must lock the particular segment in which the thread wants to operate. This type of locking mechanism is known as **Segment locking or bucket locking**. Hence at a time, 16 update operations can be performed by threads.