Java Stream

Anurak Theanpurmpul

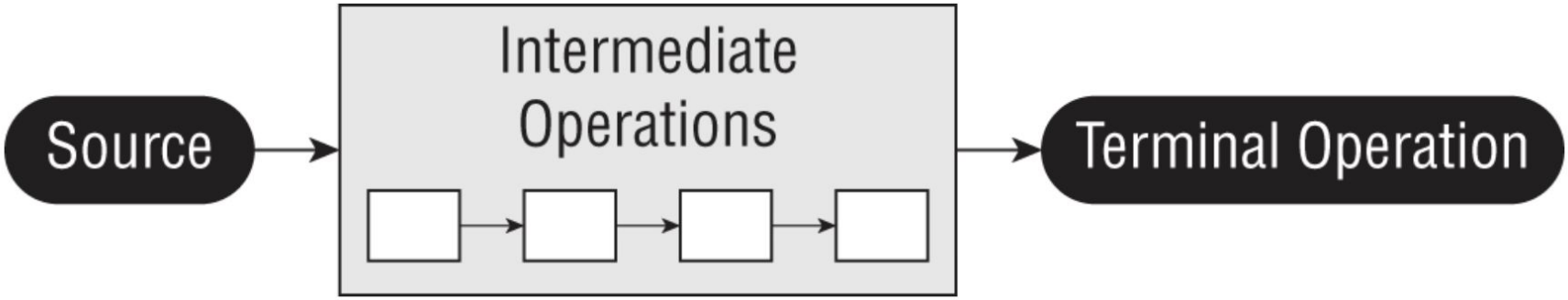
What is Java Stream API?

- **Stream** in Java is a sequence of data.
- **Stream pipeline** - the operations that run on a stream to produce a result.
- **Stream API** is a powerful framework for **sequential and parallel data processing**.
- Since JDK 8.
- **java.util.stream package**

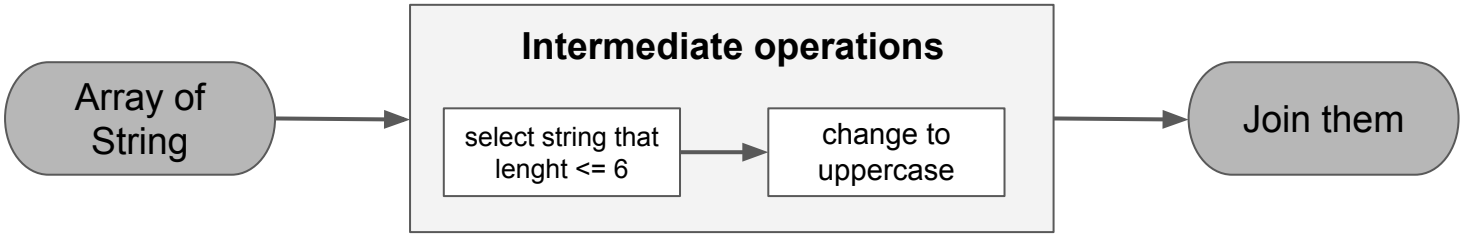
Designed by upklyak / Freepik

Stream Pipeline

- **Source:** where the stream of data comes from.
- **Intermediate operations (0..*):** transforms the stream into another one.
- **Terminal operation:** produces a non-stream result, such as a primitive value, a collection, or no value at all.
- Lazy Evaluation



Pipeline Example




```
List<String> list = Arrays.asList("anurak", "somchai", "ekachai", "som");
String result = list.stream()
        .filter(x -> x.length() >= 6) // Intermediate operations
        .map(x -> x.toUpperCase())    // Intermediate operations
        .collect(Collectors.joining("\n")); // Terminal operation
System.out.println(result);
```

- Demo **BasicPipelineExample.java**

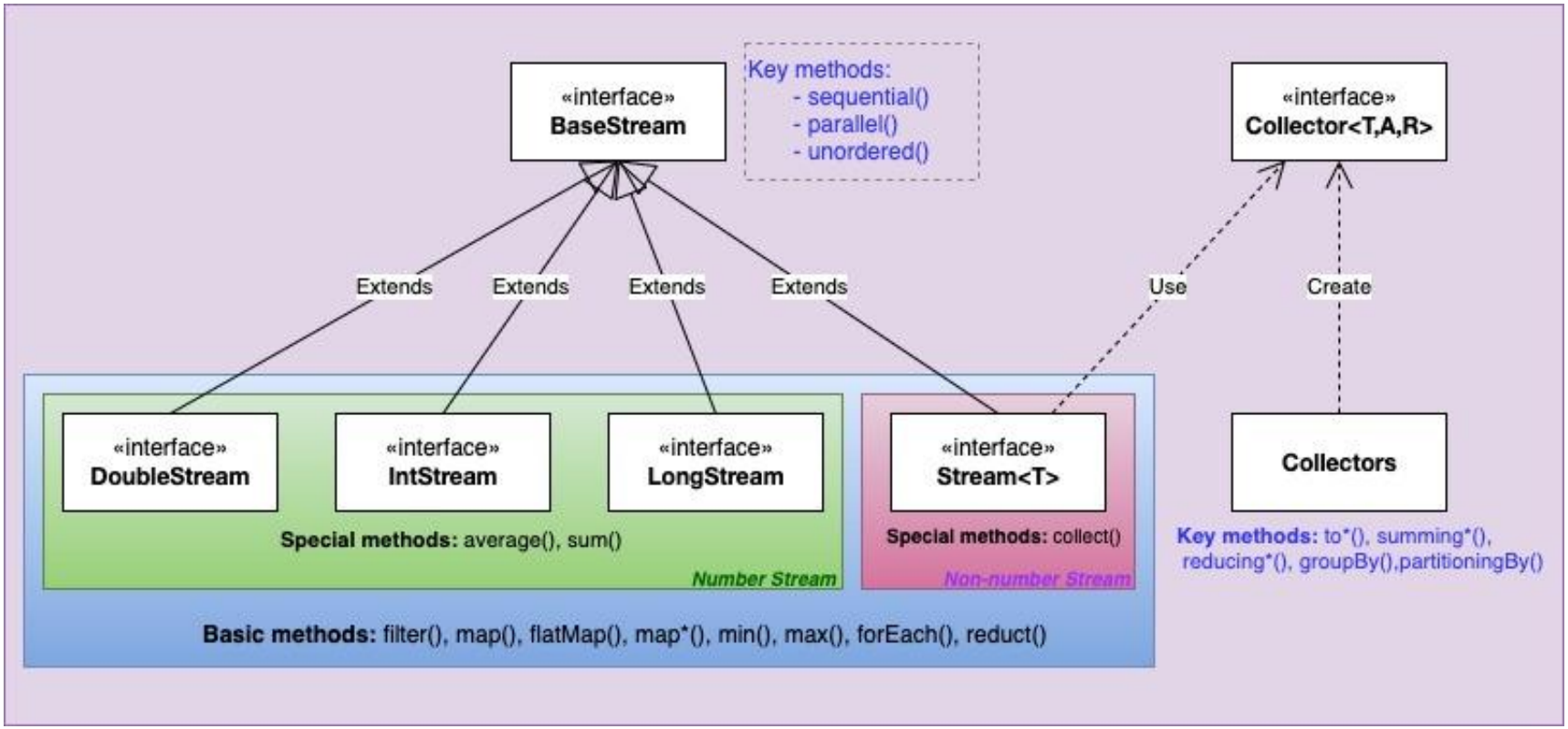
Stream Creation

- Stream<T> interface
- **Infinite streams**
  - `Stream<Double> randoms = Stream.generate(Math::random);`
  - `Stream<Integer> oddNumbers = Stream.iterate(1, n -> n + 2);`
- **Finite streams**
  - `Stream<String> singleStream = Stream.of("Hello");`
  - `Stream<String> arrayStream = Stream.of("Hello", "world");`
  - `Stream<String> emtryStream = Stream.empty();`
- **Parallel Streams**
  - **Create from parallel() of stream object**
    - ```
      Stream<Integer> listStream = Stream.iterate(1, n -> n + 1);
      Stream<Integer> parallelStream = listStream.parallel();
      ```
  - **Create from parallelStream() from collection object**
    - ```
      List<String> listForParallel = Arrays.asList("anurak", "somchai", "ekachai", "som");
      Stream<String> parallelStream = listForParallel.parallelStream();
      ```
- Demo **CreateStreamExample.java**

Stream Characteristics (compare with Collections)

- Streams have no storage.
- Infinite elements.
- The design of streams is based on internal iteration.
- Can be processed in parallel with no additional work from the developers.
- Support functional programming.
- Support lazy operations.
- Can be ordered or unordered.
- Streams cannot be reused

Stream API Architecture



- java.util.stream package

Terminal Operations

- Provide a **“final”** output of the stream pipeline, so they will not return a stream.
- A pipeline can **have only one terminal operation**.
- The stream is no longer valid after a terminal operation completes.
  - java.lang.**IllegalStateException**: stream has already been operated upon or closed
- **Reduction operation** - the contents of the stream are combined into a single primitive or Object, for examples **count()**, **min()**, **max()**, **collect()** and **reduce()**.
- **Non-Reduction operation** for examples **foreach()**, **findAny()**, **findFirst()**, **allMatch()**, **anyMatch()** and **noneMatch()**.
- **Infinite streams**
  - **Terminates**, for examples **findAny(), findFirst() and xxxMatch()**
  - **Does not terminate**, for examples **forEach()**, **reduce()** and **count()**
- Collectors - implementations of Collector that implement various useful reduction operations .
- Demo **TerminalOperationExample.java**

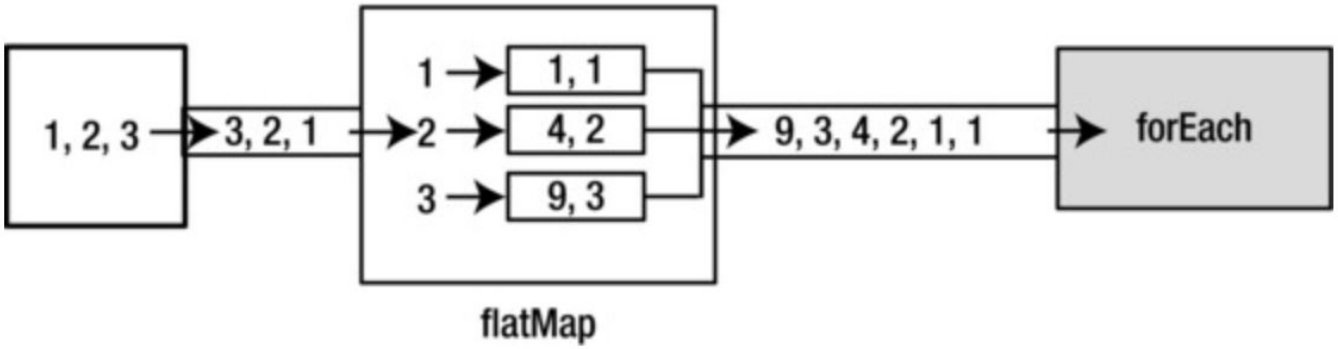
Intermediate Operations

- **Convert a stream into another stream.**
- Do not run until the terminal operation runs.
- **Stateful type**
  - **distinct()** - returns a stream with duplicate values removed.
  - **limit()** and **skip()** - can make a Stream smaller.
  - **sorted()** method returns a stream with the elements sorted.
- **Stateless type**
  - **filter()** - returns a Stream with elements that match a given expression.
  - **map()** - creates a one-to-one mapping from the elements in the stream to the elements of the next step in the stream.
  - **flatMap()** - takes each element in the stream and makes any elements it contains top-level elements in a single stream.
  - **peek()** - can use to shows how a particular step of the process is doing, without actually changing the stream (used for debugging).
- Demo **IntermediateOpExample.java**

flatMap()

- A combination of transformation and flattening (map then flat).
- Use to combine a stream of lists.
- Demo **FlatMapExample.java**

```
Stream.of(1, 2, 3)
    // map the elements of the input stream to streams, and then flatten the mapped
    // streams.
    .flatMap(n -> Stream.of(n, n * n))
    .forEach(System.out::println);
```



Parallel Stream Consideration

- For smaller streams, it might be faster to do it sequentially.
- Mutable data, share state and side effects.
- Stateful operations
- Having a performance problem in the first place.
- Don't already run the process in a multi-thread environment
- Synchronized access logic will make a process parallel will have no effect or even a negative one.
- Predictable stream sizes help the parallel performance.
- Demo **ParallelStreamExample.java, ParallelStreamExample2.java**