STREAMS

Streams

- An Interface defined in ***java.util.stream*** package.
- Streams are a ***sequence of elements from a source*** *that supports* ***aggregate operations***.
  - **filter, map, reduce, find, match, sort.**
  - Operations can be executed in **series or in parallel**.
- The *source* can be a Collection, IO Operation or Arrays
- Java collections has methods that **return Stream.**
  - These Methods are **added as default methods**.

Characteristics of Streams

- Not related to InputStreams, OutputStreams, etc.
- These are NOT data structures
  - Wrappers around Collection that carry values from a source through a pipeline of operations.
- More powerful, faster and more memory efficient than Lists
- Designed for lambdas
- Employ lazy evaluation
- Streams are parallelizable

Collections vs Streams

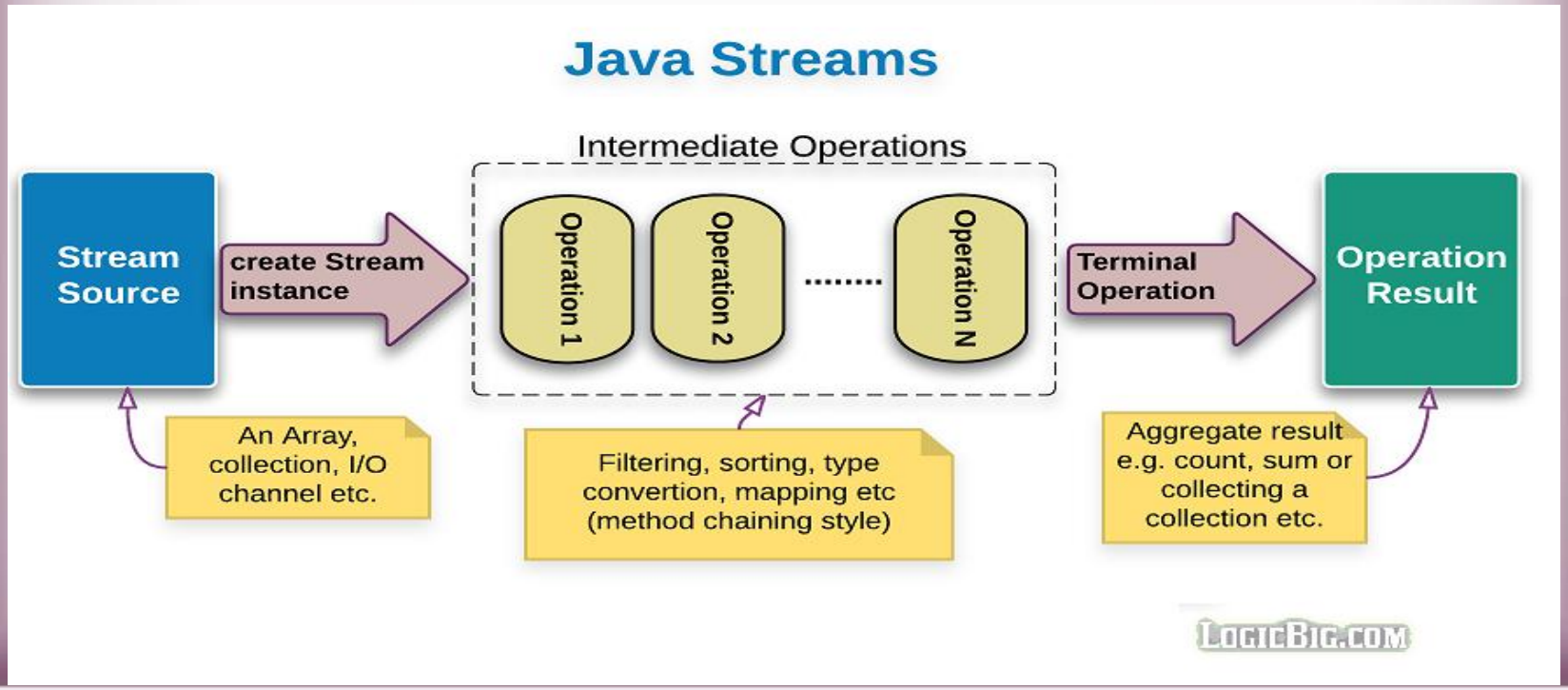
- Collections are in-memory data structures which hold elements within it.
  - Each element in the collection is computed before it actually becomes a part of that collection.
  - On the other hand Streams are data computed on-demand basis.
- Streams are lazily constructed Collections, where the values are computed when user demands for it.
- Collections are eagerly computed values

Streams

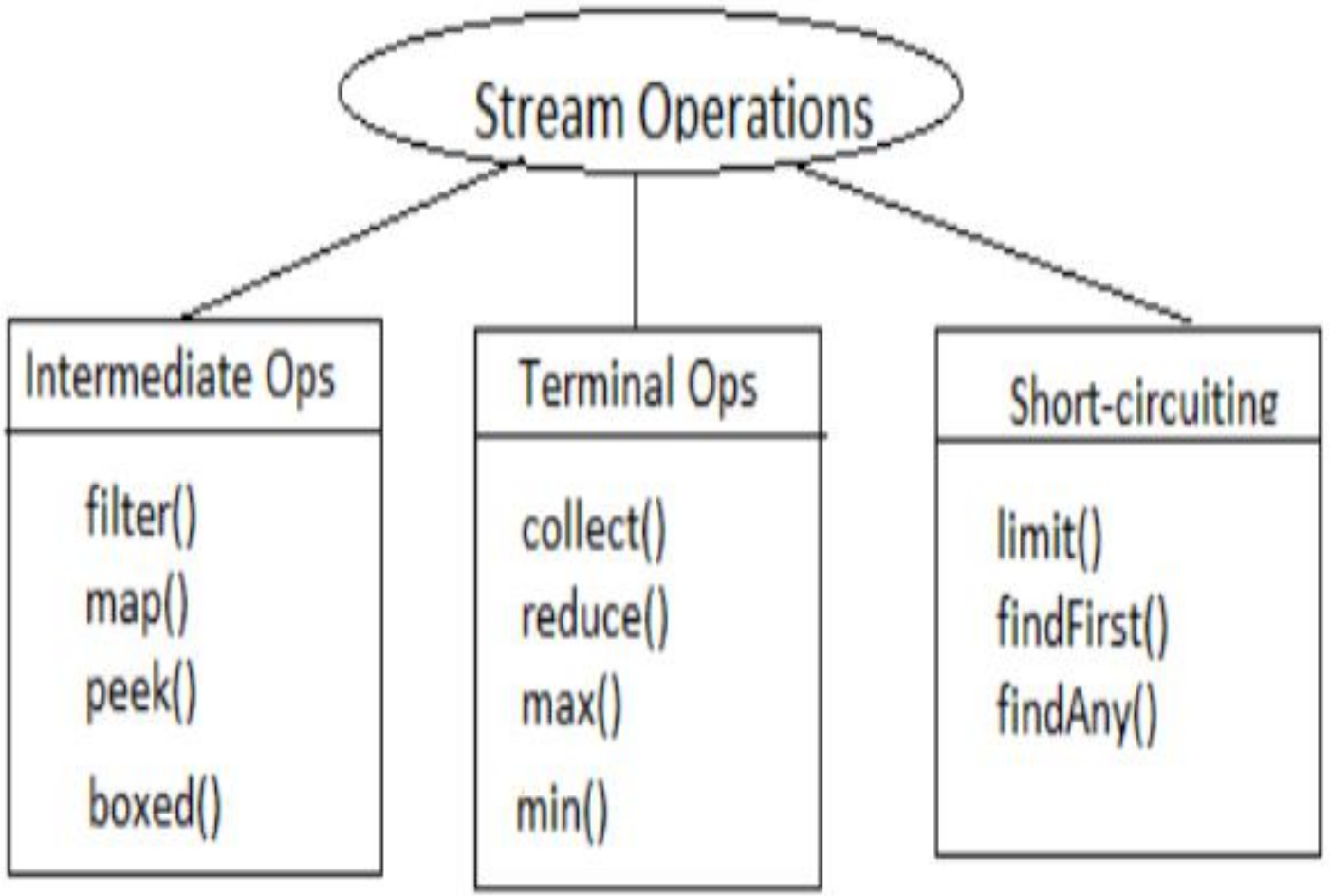
- The Streams also support Pipelining and Internal Iterations.
- Streams operations returns Streams only.
  - Helps creating chain of various stream operations.
  - **This is called as pipelining.**
- The pipelined operations looks similar to a sql query.
- In Java, we traditionally use *for* loops or *iterators* to iterate through the collections.
  - These iterations are called as *external iterations* and they are clearly visible in the code.
- **Java 8 Stream uses internal Iterations which are not visible in the code**

Structure of Java 8 Stream Operations

- The three important components to make it work.
  - A source
  - one or more intermediate operations
  - A terminal operation.
- The three are **pipelined** in a sequence to make a stream work.



Stream Operations



Operation Categories

- Two types of Operations
  - Intermediate operations
  - Terminal Operations.
- **Intermediate operations**
  - *intermediate operations* are *lazy*.
    - **They will not be invoked until the terminal operation is invoked.**
    - Streams become ***accessible*** **when intermediate operations** were called.
  - Return streams can pipeline of operations.
    - *map, filter,* and *limit*

Operation Categories

- **The terminal operations**
  - Used at the end of a pipeline
  - They close the stream in some meaningful way.
  - Eagerly executed
  - Collects the results of stream operations
  - Can be collected like lists, integers or simply nothing.

- Stream become *inaccessible* when a terminal operation is called
  - Will trigger the *IllegalStateException which is a RuntimeException*
- **Example : collect() ,count() ,forEach(), min() ,max()**

INTERMEDIATE OPERATIONS

Types of Intermediate Operations

- Intermediate operations can be classified into Stateful and Stateless

- **Stateful Operations**
  - Operations which maintain information from a previous invocation internally
  - It can be used again in a future invocation of the method
  - State storage can become huge for instances of infinite streams
  - Can potentially affect performance of the whole system.

  - **Example : distinct() , sorted()**

Stateless Intermediate Operations:

- Operations do not store any state across passes.
- Improves the performance of these operations
- Stream operation can be in parallel
  - No information to shared, no need for order
- **Example : filter(), map(), findAny()**

Filter()

- Takes predicate as an argument
- Applies to the whole stream and returns a filtered stream
  - Containing elements which match the Predicate.

```
List<Customer> filteredList= customerList.stream().filter((Customer
    cust ) -> cust.getCustomerId() >200).collect(toList());

System.out.println("After applying filter method");
filteredList.forEach(System.out::println);
```

Map()

- Takes an instance of Function <T, R> as parameter
- Converts the type of elements in the stream from the current type T to the new type R.
- Returns as output a stream of type R or Stream<R>

```
List<Customer> customerList = HandleCustomers.getCustomers();

List<Long> phoneBook = customerList.stream().
        map((eachCust)->eachCust.getPhoneNumber()).collect(toList());

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

Distinct()

- returns a stream instance which has all elements unique/distinct.
- **The *uniqueness* is determined by the equals & hashcode**

```
List<Long> custIds= customerList.stream().map(e ->
    e.getCustomerId()).distinct().collect(toList());
```

Sorted

- sorted()
  - Sorts the elements using natural ordering.
  - The element class must implement Comparable interface.

```
List<Customer> sortedList =
    customerList.stream().sorted().collect(toList());
```

- **sorted(Comparator<? super T> comparator)**
  - Sorted using an instance of Comparator using lambda expression.
- **Comparator.comparing**
  - A static function that accepts a sort key *Function*
  - Returns a *Comparator* for the type which contains the sort key:

Sorted

- **Comparator.comparing**

```
List<Customer> slist =
    customerList.stream().sorted(Comparator.comparing(Customer::
    getCustomerId)).collect(toList());
```

- **reversed()**
  - To reverse the natural ordering Comparator

```
List<Customer> revrseList =
    customerList.stream().sorted(Comparator.reverseOrder()).collect
    (toList());
```

- *thenComparing()*
  - To set up lexicographical ordering of values by provisioning multiple sort keys in a particular sequence.

TERMINAL OPERATIONS

Stream Reduction

- *count(), max(), min(), sum()*
  - Predefined Terminal Operation implementation.
  - Used to aggregate a stream to a type or to a primitive
- **Can create a custom Stream's reduction mechanism** by using two methods
- ***reduce()***
- ***collect()***

Max , Min ,Count

- ***count()***
  - returns the count of elements of a stream.
- ***max()***
  - returns the maximum element of a stream according to the provided Comparator.
- ***min()***
  - returns the minimum element of a stream according to the provided Comparator.

Min

```
List<Product> catalog = new ArrayList<>();

Comparator<Product> priceComparator =
    Comparator.comparing(Product::getRatePerUnit);

   String itemType="TV";

Optional<Product> minresult = catalog.stream().
                        filter((Product prod ) ->
    prod.getProductName().equalsIgnoreCase(itemType)).
                          min(priceComparator);
 double leastPriced=0;
 if(minresult.isPresent())
 {
  leastPriced= minresult.get().getRatePerUnit();
 }
```

Max

```
Optional<Product> maxresult= catalog.stream().
                    filter((Product prod ) ->
  prod.getProductName().equalsIgnoreCase(itemType)).
                        max(priceComparator);
```

```
double highPriced=0;
if(maxresult.isPresent())
{
highPriced= maxresult.get().getRatePerUnit();
}
```

```
System.out.println("Highest Priced :="+itemType +
   highPriced);
```

Count()

```
long  countresult= catalog.stream().
        filter((Product prod ) ->
  prod.getProductName().equalsIgnoreCase(itemType)).count();
```

System.***out.println(countresult);***

collect()

- A terminal operation
- It accepts an argument of the type *Collector*
  - Specifies the mechanism of reduction.
- Predefined collectors for most common operations.
- Accessed with the help of the *Collectors* type.

- ***Collectors.toList()***
  - Used for collecting all *Stream* elements into a *List* instance.
  - Can have more control with *toCollection* instead
- ***Collectors.toSet()***
  - Used for collecting all *Stream* elements into a *Set* instance.

collect()

- ***Collectors.toCollection()***
  - To use a custom implementation

```
List<String> result = givenList.stream()
  .collect(toCollection(LinkedList::new))
```

- ***Collectors.toMap()***
  - Used to collect *Stream* elements into a *Map* instance.

```
Map<Long, String> map = customerList.stream()
        .collect(Collectors.toMap(Customer::getPhoneNumber,
                                  Customer::getCustomerName));

map.forEach((x, y) -> System.out.println("Key: " + x +", value: "+ y));
```