Stream API

Introduced in Java 8, the Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.

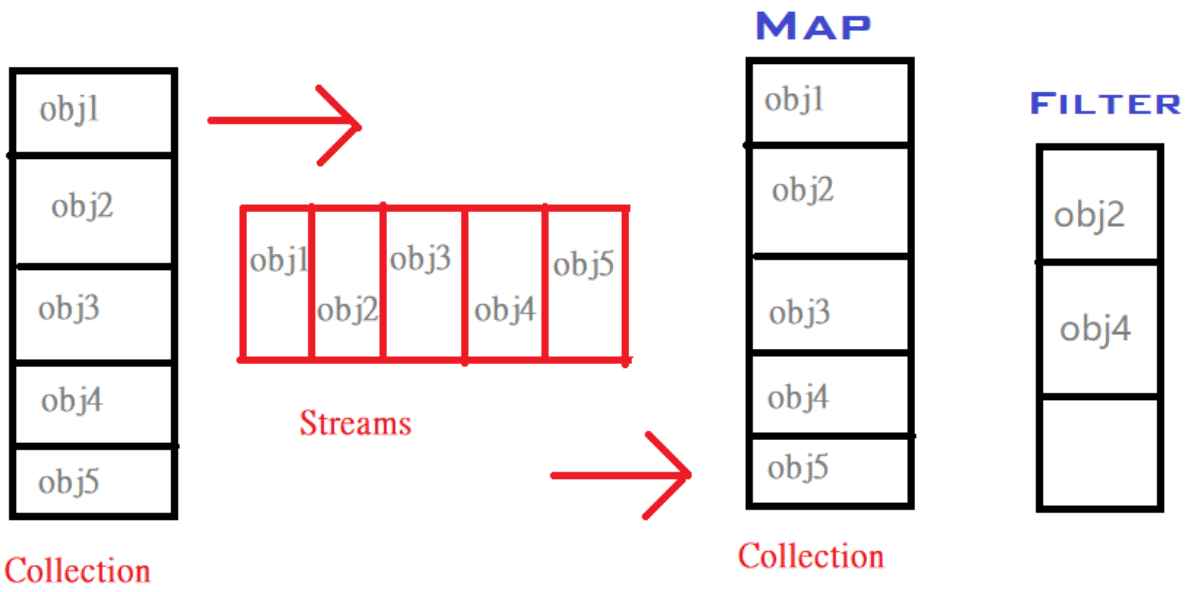
Collection : group of object in single in entity

Stream : process of collections object.

Package: java.util.stream*;

The features of Java stream are –

- A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.
- Streams don't change the original data structure; they only provide the result as per the pipelined methods.
- **Each intermediate operation is lazily executed and returns a stream as a result, hence various *intermediate operations* can be pipelined. Terminal operations mark the end of the stream and return the result.**



Different Operations on Streams-

Intermediate Operations:

1. **map**: The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.

   List number = Arrays.asList(2,3,4,5);

   List square = number.stream().map(x->x*x).collect(Collectors.toList());

2. **filter**: The filter method is used to select elements as per the Predicate passed as argument.

   List names = Arrays.asList("Reflection","Collection","Stream");

   List result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());

3. **sorted**: The sorted method is used to sort the stream.

   ```
   List names = Arrays.asList("Reflection","Collection","Stream");

   List result = names.stream().sorted().collect(Collectors.toList());
   ```

Terminal Operations:

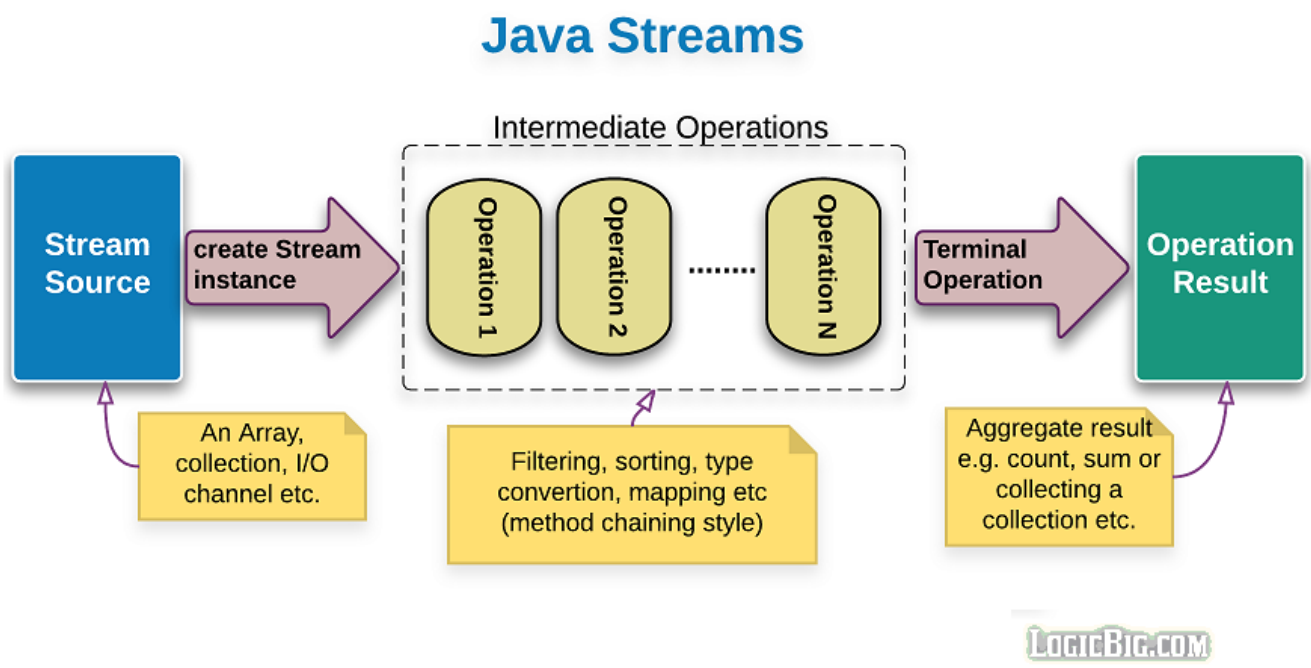
4. **collect**: The collect method is used to return the result of the intermediate operations performed on the stream.

   ```
   List number = Arrays.asList(2,3,4,5,3);

   Set square = number.stream().map(x->x*x).collect(Collectors.toSet());
   ```

5. **forEach**: The forEach method is used to iterate through every element of the stream.

   ```
   List number = Arrays.asList(2,3,4,5);

   number.stream().map(x->x*x).forEach(y->System.out.println(y));
   ```

6. **reduce**: The reduce method is used to reduce the elements of a stream to a single value. The reduce method takes a BinaryOperator as a parameter.
7. **Count()**:



Java Stream Interface Methods

Modifier and Type	Method and Description
boolean	allMatch(Predicate<? super T> predicate) Returns whether all elements of this stream match the provided predicate.
boolean	anyMatch(Predicate<? super T> predicate) Returns whether any elements of this stream match the provided predicate.
static <T> Stream.Builder<T>	builder() Returns a builder for a Stream.
<R,A> R	collect(Collector<? super T,A,R> collector) Performs a mutable reduction operation on the elements of this stream using a Collector.
<R> R	collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner) Performs a mutable reduction operation on the elements of this stream.
static <T> Stream<T>	concat(Stream<? extends T> a, Stream<? extends T> b) Creates a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream.
long	count()

	Returns the count of elements in this stream.
Stream<T>	**distinct**() Returns a stream consisting of the distinct elements (according to **Object.equals(Object)**) of this stream.
static <T> **Stream**<T>	**empty**() Returns an empty sequential Stream.
Stream<T>	**filter**(**Predicate**<? super T> predicate) Returns a stream consisting of the elements of this stream that match the given predicate.
Optional<T>	**findAny**() Returns an **Optional** describing some element of the stream, or an empty Optional if the stream is empty.
Optional<T>	**findFirst**() Returns an **Optional** describing the first element of this stream, or an empty Optional if the stream is empty.
<R> **Stream**<R>	**flatMap**(**Function**<? super T,? extends **Stream**<? extends R>> mapper) Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
DoubleStream	**flatMapToDouble**(**Function**<? super T,? extends **DoubleStream**> mapper) Returns an DoubleStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
IntStream	**flatMapToInt**(**Function**<? super T,? extends **IntStream**> mapper) Returns an IntStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
LongStream	**flatMapToLong**(**Function**<? super T,? extends **LongStream**> mapper) Returns an LongStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
void	**forEach**(**Consumer**<? super T> action) Performs an action for each element of this stream.
void	**forEachOrdered**(**Consumer**<? super T> action) Performs an action for each element of this stream, in the encounter order of the stream if the stream has a defined encounter order.
static <T> **Stream**<T>	**generate**(**Supplier**<T> s) Returns an infinite sequential unordered stream where each element is generated by the provided Supplier.
static <T> **Stream**<T>	**iterate**(T seed, **UnaryOperator**<T> f) Returns an infinite sequential ordered Stream produced by iterative application of a function f to an initial element seed, producing a Stream consisting of seed, f(seed), f(f(seed)), etc.
Stream<T>	**limit**(long maxSize) Returns a stream consisting of the elements of this stream, truncated to be no longer than maxSize in length.

<R> Stream<R>	map(Function<? super T,? extends R> mapper) Returns a stream consisting of the results of applying the given function to the elements of this stream.
DoubleStream	mapToDouble(ToDoubleFunction<? super T> mapper) Returns a DoubleStream consisting of the results of applying the given function to the elements of this stream.
IntStream	mapToInt(ToIntFunction<? super T> mapper) Returns an IntStream consisting of the results of applying the given function to the elements of this stream.
LongStream	mapToLong(ToLongFunction<? super T> mapper) Returns a LongStream consisting of the results of applying the given function to the elements of this stream.
Optional<T>	max(Comparator<? super T> comparator) Returns the maximum element of this stream according to the provided Comparator.
Optional<T>	min(Comparator<? super T> comparator) Returns the minimum element of this stream according to the provided Comparator.
boolean	noneMatch(Predicate<? super T> predicate) Returns whether no elements of this stream match the provided predicate.
static <T> Stream<T>	of(T... values) Returns a sequential ordered stream whose elements are the specified values.
static <T> Stream<T>	of(T t) Returns a sequential Stream containing a single element.
Stream<T>	peek(Consumer<? super T> action) Returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream.
Optional<T>	reduce(BinaryOperator<T> accumulator) Performs a **reduction** on the elements of this stream, using an **associative** accumulation function, and returns an Optional describing the reduced value, if any.
T	reduce(T identity, BinaryOperator<T> accumulator) Performs a **reduction** on the elements of this stream, using the provided identity value and an **associative** accumulation function, and returns the reduced value.
<U> U	reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner) Performs a **reduction** on the elements of this stream, using the provided identity, accumulation and combining functions.
Stream<T>	skip(long n) Returns a stream consisting of the remaining elements of this stream after discarding the first n elements of the stream.
Stream<T>	sorted() Returns a stream consisting of the elements of this stream, sorted according to natural order.
Stream<T>	sorted(Comparator<? super T> comparator)

	Returns a stream consisting of the elements of this stream, sorted according to the provided `Comparator`.
`Object[]`	`toArray()` Returns an array containing the elements of this stream.
`<A> A[]`	`toArray(IntFunction<A[]> generator)` Returns an array containing the elements of this stream, using the provided `generator` function to allocate the returned array, as well as any additional arrays that might be required for a partitioned execution or for resizing.

Java Example: Filtering Collection without using Stream

```java
import java.util.*;
class Product{
    int id;
    String name;
    float price;
    public Product(int id, String name, float price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }
}
class JavaStreamExample {
    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();
        //Adding Products
        productsList.add(new Product( id: 1, name: "HP Laptop", price: 25000f));
        productsList.add(new Product( id: 2, name: "Dell Laptop", price: 30000f));
        productsList.add(new Product( id: 3, name: "Lenevo Laptop", price: 28000f));
        productsList.add(new Product( id: 4, name: "Sony Laptop", price: 28000f));
        productsList.add(new Product( id: 5, name: "Apple Laptop", price: 90000f));
        List<Float> productPriceList = new ArrayList<Float>();
        for(Product product: productsList) {
            // filtering data of list
            if (product.price < 30000) {
                productPriceList.add(product.price);    // adding price to a productPriceList
            }
        }
        System.out.println(productPriceList);   // displaying data
    }
}
```

```
[25000.0, 28000.0, 28000.0]
```

Java Example: Filtering Collection by using Stream

Here, we are filtering data by using stream. You can see that code is optimized and maintained. Stream provides fast execution.

```
package com.stream.api;
import java.util.*;
import java.util.stream.Collectors;
class Product{
    int id;
    String name;
    float price;
    public Product(int id, String name, float price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }
}
 class JavaStreamExample {
    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();
        //Adding Products
        productsList.add(new Product( id: 1, name: "HP Laptop", price: 25000f));
        productsList.add(new Product( id: 2, name: "Dell Laptop", price: 30000f));
        productsList.add(new Product( id: 3, name: "Lenevo Laptop", price: 28000f));
        productsList.add(new Product( id: 4, name: "Sony Laptop", price: 28000f));
        productsList.add(new Product( id: 5, name: "Apple Laptop", price: 90000f));
        List<Float> productPriceList2 =productsList.stream()
                .filter(p -> p.price > 30000)// filtering data
                .map(p->p.price)        // fetching price
                .collect(Collectors.toList()); // collecting as list
        System.out.println(productPriceList2);
    }
}
```

simple program to demonstrate the use of stream in java

```java
package com.stream.api;
import java.util.*;
import java.util.stream.*;
class Demo
{
    public static void main(String args[])
    {
        // create a list of integers
        List<Integer> number = Arrays.asList(2,3,4,5);
        // demonstration of map method
        List<Integer> square = number.stream()
                .map(x -> x*x)
                .collect(Collectors.toList());
        System.out.println(square);
        // create a list of String
        List<String> names = Arrays.asList("Reflection","Collection","Stream");
        // demonstration of filter method
        List<String> result = names.stream()
                .filter(s->s.startsWith("S"))
                .collect(Collectors.toList());
        System.out.println(result);
        // demonstration of sorted method
        List<String> show = names.stream()
                .sorted()
                .collect(Collectors.toList());
        System.out.println(show);
        // create a list of integers
        List<Integer> numbers = Arrays.asList(2,3,4,5,2);
        // collect method returns a set
        Set<Integer> squareSet = numbers.stream()
```

```java
        List<Integer> numbers = Arrays.asList(2,3,4,5,2);
        // collect method returns a set
        Set<Integer> squareSet = numbers.stream()
                        .map(x->x*x)
                        .collect(Collectors.toSet());
        System.out.println(squareSet);
        // demonstration of forEach method
        number.stream()
                .map(x->x*x)
                .forEach(y->System.out.println(y));

        // demonstration of reduce method
        int even = number.stream().filter(x->x%2==0).reduce( identity: 0,(ans,i)-> ans+i);

        System.out.println(even);
    }
}
```

Java Stream Example: count() Method in Collection

```
    }
}
```

Output:

```
3
```

stream allows you to collect your result in any various forms. You can get you result as set, list or map and can perform manipulation on the elements.

Java Stream Example : Convert List into Set

```
}
```

Output:

```
[25000.0, 28000.0]
```

Java Stream Example: Convert List into Map

```
}
```

Output:

```
{1=HP Laptop, 2=Dell Laptop, 3=Lenevo Laptop, 4=Sony Laptop, 5=Apple Laptop}
```

Java code for Stream flatMap(Function mapper)