

```
//Stream.iterate(initial value, next value)
Stream.iterate(0, n -> n + 1)
    .limit(10)
    .forEach(x -> System.out.println(x));
```

Output

```
0
1
2
3
4
5
6
7
8
9
```

1.2 Stream of odd numbers only.

```
Stream.iterate(0, n -> n + 1)
    .filter(x -> x % 2 != 0) //odd
    .limit(10)
    .forEach(x -> System.out.println(x));
```

```
1
3
5
7
9
11
13
15
17
19
```

1.3 A classic Fibonacci example.

```
Stream.iterate(new int[]{0, 1}, n -> new int[]{n[1], n[0] + n[1]})
    .limit(20)
    .map(n -> n[0])
    .forEach(x -> System.out.println(x));
```

Output

```
0
```

1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181

Sum all the Fibonacci values.

```
int sum = Stream.iterate(new int[]{0, 1}, n -> new int[]{n[1], n[0]
+ n[1]})
    .limit(10)
    .map(n -> n[0]) // Stream<Integer>
    .mapToInt(n -> n)
    .sum();

System.out.println("Fibonacci 10 sum : " + sum);
```

Output

Fibonacci 10 sum : 88

In Java 8, the `Stream.reduce()` combine elements of a stream and produces a single value.

A simple sum operation using a for loop.

```
int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int sum = 0;
for (int i : numbers) {
    sum += i;
}

System.out.println("sum : " + sum); // 55
```

The equivalent in `Stream.reduce()`

```
int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

// 1st argument, init value = 0
int sum = Arrays.stream(numbers).reduce(0, (a, b) -> a + b);

System.out.println("sum : " + sum); // 55
```

Java 8 Stream

Java provides a new additional package in Java 8 called `java.util.stream`. This package consists of classes, interfaces and enum to allows functional-style operations on the elements. You can use stream by importing `java.util.stream` package.

Stream provides following features:

- Stream does not store elements.
- Stream is functional in nature. Operations performed on a stream does not modify it's source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.
- Stream is lazy and evaluates code only when required.
- The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.

You can use stream to filter, collect, print, and convert from one data structure to other etc.

[Java Stream Interface Methods](#)

Methods	Description
boolean allMatch(Predicate<? super T> predicate)	It returns all elements of this stream which match the provided predicate. If the stream is empty then true is returned and the predicate is not evaluated.
boolean anyMatch(Predicate<? super T> predicate)	It returns any element of this stream that matches the provided predicate. If the stream is empty then false is returned and the predicate is not evaluated.
<R,A> R collect(Collector<? super T,A,R> collector)	It performs a mutable reduction operation on the elements of this stream using a Collector. A Collector encapsulates the functions used as arguments to collect(Supplier, BiConsumer, BiConsumer), allowing for reuse of collection strategies and composition of collect operations such as multiple-level grouping or partitioning.
<R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)	It performs a mutable reduction operation on the elements of this stream. A mutable reduction is one in which the reduced value is a mutable result container, such as an ArrayList, and elements are incorporated by updating the state of the result rather than by replacing the result.
static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)	It creates a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream. The resulting stream is ordered if both of the input streams are ordered, and parallel if either of the input streams is parallel. When the resulting stream is closed, the close handlers for both input streams are invoked.
long count()	It returns the count of elements in this stream. This is a special case of a reduction.
Stream<T> distinct()	It returns a stream consisting of the distinct elements (according to Object.equals(Object)) of this stream.
static <T> Stream<T> empty()	It returns an empty sequential Stream.
Stream<T> filter(Predicate<? super T> predicate)	It returns a stream consisting of the elements of this stream that match the given predicate.
Optional<T> findAny()	It returns an Optional describing some element of the stream, or an empty Optional if the stream is empty.
Optional<T> findFirst()	It returns an Optional describing the first element of this stream, or an empty Optional if the stream is empty. If

	the stream has no encounter order, then any element may be returned.
<code><R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)</code>	It 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. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.)
<code>DoubleStream flatMapToDouble(Function<? super T,? extends DoubleStream> mapper)</code>	It returns a 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. Each mapped stream is closed after its contents have placed been into this stream. (If a mapped stream is null an empty stream is used, instead.)
<code>IntStream flatMapToInt(Function<? super T,? extends IntStream> mapper)</code>	It 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. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.)
<code>LongStream flatMapToLong(Function<? super T,? extends LongStream> mapper)</code>	It returns a 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. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.)
<code>void forEach(Consumer<? super T> action)</code>	It performs an action for each element of this stream.
<code>void forEachOrdered(Consumer<? super T> action)</code>	It performs an action for each element of this stream, in the encounter order of the stream if the stream has a defined encounter order.
<code>static <T> Stream<T> generate(Supplier<T> s)</code>	It returns an infinite sequential unordered stream where each element is generated by the provided Supplier. This is suitable for generating constant streams, streams of random elements, etc.

static <T> Stream<T> iterate(T seed,UnaryOperator<T> f)	It 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)	It 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)	It returns a stream consisting of the results of applying the given function to the elements of this stream.
DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)	It returns a DoubleStream consisting of the results of applying the given function to the elements of this stream.
IntStream mapToInt(ToIntFunction<? super T> mapper)	It returns an IntStream consisting of the results of applying the given function to the elements of this stream.
LongStream mapToLong(ToLongFunction<? super T> mapper)	It 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)	It returns the maximum element of this stream according to the provided Comparator. This is a special case of a reduction.
Optional<T> min(Comparator<? super T> comparator)	It returns the minimum element of this stream according to the provided Comparator. This is a special case of a reduction.
boolean noneMatch(Predicate<? super T> predicate)	It returns elements of this stream match the provided predicate. If the stream is empty then true is returned and the predicate is not evaluated.
@SafeVarargs static <T> Stream<T> of(T... values)	It returns a sequential ordered stream whose elements are the specified values.
static <T> Stream<T> of(T t)	It returns a sequential Stream containing a single element.
Stream<T> peek(Consumer<? super T> action)	It 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)	It 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)	It 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)	It performs a reduction on the elements of this stream, using the provided identity, accumulation and combining functions.
Stream<T> skip(long n)	It returns a stream consisting of the remaining elements of this stream after discarding the first n elements of the stream. If this stream contains fewer than n elements then an empty stream will be returned.
Stream<T> sorted()	It returns a stream consisting of the elements of this stream, sorted according to natural order. If the elements of this stream are not Comparable, a java.lang.ClassCastException may be thrown when the terminal operation is executed.
Stream<T> sorted(Comparator<? super T> comparator)	It returns a stream consisting of the elements of this stream, sorted according to the provided Comparator.
Object[] toArray()	It returns an array containing the elements of this stream.
<A> A[] toArray(IntFunction<A[]> generator)	It 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

In the following example, we are filtering data without using stream. This approach we are used before the stream package was released.

```

1. import java.util.*;
2. class Product{
3.     int id;
4.     String name;
5.     float price;
6.     public Product(int id, String name, float price) {
7.         this.id = id;
8.         this.name = name;
9.         this.price = price;
10.    }
11. }
12. public class JavaStreamExample {
13.     public static void main(String[] args) {
14.         List<Product> productsList = new ArrayList<Product>();
15.         //Adding Products
16.         productsList.add(new Product(1,"HP Laptop",25000f));
17.         productsList.add(new Product(2,"Dell Laptop",30000f));
18.         productsList.add(new Product(3,"Lenevo Laptop",28000f));
19.         productsList.add(new Product(4,"Sony Laptop",28000f));
20.         productsList.add(new Product(5,"Apple Laptop",90000f));
21.         List<Float> productPriceList = new ArrayList<Float>();
22.         for(Product product: productsList){
23.
24.             // filtering data of list
25.             if(product.price<30000){
26.                 productPriceList.add(product.price); // adding price to a productPriceList
27.             }
28.         }
29.         System.out.println(productPriceList); // displaying data
30.     }
31. }

```

Output:

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

Java Stream 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.

```

1. import java.util.*;
2. import java.util.stream.Collectors;

```



```

3. class Product{
4.     int id;
5.     String name;
6.     float price;
7.     public Product(int id, String name, float price) {
8.         this.id = id;
9.         this.name = name;
10.        this.price = price;
11.    }
12. }
13. public class JavaStreamExample {
14.     public static void main(String[] args) {
15.         List<Product> productsList = new ArrayList<Product>();
16.         //Adding Products
17.         productsList.add(new Product(1,"HP Laptop",25000f));
18.         productsList.add(new Product(2,"Dell Laptop",30000f));
19.         productsList.add(new Product(3,"Lenevo Laptop",28000f));
20.         productsList.add(new Product(4,"Sony Laptop",28000f));
21.         productsList.add(new Product(5,"Apple Laptop",90000f));
22.         List<Float> productPriceList2 =productsList.stream()
23.             .filter(p -> p.price > 30000)// filtering data
24.             .map(p->p.price)    // fetching price
25.             .collect(Collectors.toList()); // collecting as list
26.         System.out.println(productPriceList2);
27.     }
28. }

```

Output:

```
[90000.0]
```

Java Stream Iterating Example

You can use stream to iterate any number of times. Stream provides predefined methods to deal with the logic you implement. In the following example, we are iterating, filtering and passed a limit to fix the iteration.

```

1. import java.util.stream.*;
2. public class JavaStreamExample {
3.     public static void main(String[] args){
4.         Stream.iterate(1, element->element+1)
5.         .filter(element->element%5==0)

```

```
6.     .limit(5)
7.     .forEach(System.out::println);
8. }
9. }
```

Output:

```
5
10
15
20
25
```

Java Stream Example: Filtering and Iterating Collection

In the following example, we are using filter() method. Here, you can see code is optimized and very concise.

```
1. import java.util.*;
2. class Product{
3.     int id;
4.     String name;
5.     float price;
6.     public Product(int id, String name, float price) {
7.         this.id = id;
8.         this.name = name;
9.         this.price = price;
10.    }
11. }
12. public class JavaStreamExample {
13.     public static void main(String[] args) {
14.         List<Product> productsList = new ArrayList<Product>();
15.         //Adding Products
16.         productsList.add(new Product(1,"HP Laptop",25000f));
17.         productsList.add(new Product(2,"Dell Laptop",30000f));
18.         productsList.add(new Product(3,"Lenevo Laptop",28000f));
19.         productsList.add(new Product(4,"Sony Laptop",28000f));
20.         productsList.add(new Product(5,"Apple Laptop",90000f));
21.         // This is more compact approach for filtering data
22.         productsList.stream()
23.             .filter(product -> product.price == 30000)
24.             .forEach(product -> System.out.println(product.name));
25.     }
26. }
```

Output:

Dell Laptop

Java Stream Example : reduce() Method in Collection

This method takes a sequence of input elements and combines them into a single summary result by repeated operation. For example, finding the sum of numbers, or accumulating elements into a list.

In the following example, we are using reduce() method, which is used to sum of all the product prices.

```
1. import java.util.*;
2. class Product{
3.     int id;
4.     String name;
5.     float price;
6.     public Product(int id, String name, float price) {
7.         this.id = id;
8.         this.name = name;
9.         this.price = price;
10.    }
11. }
12. public class JavaStreamExample {
13.     public static void main(String[] args) {
14.         List<Product> productsList = new ArrayList<Product>();
15.         //Adding Products
16.         productsList.add(new Product(1,"HP Laptop",25000f));
17.         productsList.add(new Product(2,"Dell Laptop",30000f));
18.         productsList.add(new Product(3,"Lenevo Laptop",28000f));
19.         productsList.add(new Product(4,"Sony Laptop",28000f));
20.         productsList.add(new Product(5,"Apple Laptop",90000f));
21.         // This is more compact approach for filtering data
22.         Float totalPrice = productsList.stream()
23.             .map(product->product.price)
24.             .reduce(0.0f,(sum, price)->sum+price); // accumulating price
25.         System.out.println(totalPrice);
26.         // More precise code
27.         float totalPrice2 = productsList.stream()
28.             .map(product->product.price)
29.             .reduce(0.0f,Float::sum); // accumulating price, by referring method of Float class
```

```
30.    System.out.println(totalPrice2);
31.
32. }
33. }
```

Output:

```
201000.0
201000.0
```

Java Stream Example: Sum by using Collectors Methods

We can also use collectors to compute sum of numeric values. In the following example, we are using Collectors class and its specified methods to compute sum of all the product prices.

```
1. import java.util.*;
2. import java.util.stream.Collectors;
3. class Product{
4.     int id;
5.     String name;
6.     float price;
7.     public Product(int id, String name, float price) {
8.         this.id = id;
9.         this.name = name;
10.        this.price = price;
11.    }
12. }
13. public class JavaStreamExample {
14.     public static void main(String[] args) {
15.         List<Product> productList = new ArrayList<Product>();
16.         //Adding Products
17.         productList.add(new Product(1,"HP Laptop",25000f));
18.         productList.add(new Product(2,"Dell Laptop",30000f));
19.         productList.add(new Product(3,"Lenevo Laptop",28000f));
20.         productList.add(new Product(4,"Sony Laptop",28000f));
21.         productList.add(new Product(5,"Apple Laptop",90000f));
22.         // Using Collectors's method to sum the prices.
23.         double totalPrice3 = productList.stream()
24.             .collect(Collectors.summingDouble(product->product.price));
25.         System.out.println(totalPrice3);
26.
27.     }
28. }
```

Output:

201000.0

Java Stream Example: Find Max and Min Product Price

Following example finds min and max product price by using stream. It provides convenient way to find values without using imperative approach.

```
1. import java.util.*;
2. class Product{
3.     int id;
4.     String name;
5.     float price;
6.     public Product(int id, String name, float price) {
7.         this.id = id;
8.         this.name = name;
9.         this.price = price;
10.    }
11. }
12. public class JavaStreamExample {
13.     public static void main(String[] args) {
14.         List<Product> productsList = new ArrayList<Product>();
15.         //Adding Products
16.         productsList.add(new Product(1,"HP Laptop",25000f));
17.         productsList.add(new Product(2,"Dell Laptop",30000f));
18.         productsList.add(new Product(3,"Lenevo Laptop",28000f));
19.         productsList.add(new Product(4,"Sony Laptop",28000f));
20.         productsList.add(new Product(5,"Apple Laptop",90000f));
21.         // max() method to get max Product price
22.         Product productA = productsList.stream().max((product1, product2)-
>product1.price > product2.price ? 1: -1).get();
23.         System.out.println(productA.price);
24.         // min() method to get min Product price
25.         Product productB = productsList.stream().min((product1, product2)-
>product1.price > product2.price ? 1: -1).get();
26.         System.out.println(productB.price);
27.
28.     }
29. }
```

Output:

90000.0
25000.0

Java Stream Example: count() Method in Collection

```
1. import java.util.*;
2. class Product{
3.     int id;
4.     String name;
5.     float price;
6.     public Product(int id, String name, float price) {
7.         this.id = id;
8.         this.name = name;
9.         this.price = price;
10.    }
11. }
12. public class JavaStreamExample {
13.     public static void main(String[] args) {
14.         List<Product> productsList = new ArrayList<Product>();
15.         //Adding Products
16.         productsList.add(new Product(1,"HP Laptop",25000f));
17.         productsList.add(new Product(2,"Dell Laptop",30000f));
18.         productsList.add(new Product(3,"Lenevo Laptop",28000f));
19.         productsList.add(new Product(4,"Sony Laptop",28000f));
20.         productsList.add(new Product(5,"Apple Laptop",90000f));
21.         // count number of products based on the filter
22.         long count = productsList.stream()
23.             .filter(product->product.price<30000)
24.             .count();
25.         System.out.println(count);
26.     }
27. }
```

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

```

1. import java.util.*;
2. import java.util.stream.Collectors;
3. class Product{
4.     int id;
5.     String name;
6.     float price;
7.     public Product(int id, String name, float price) {
8.         this.id = id;
9.         this.name = name;
10.        this.price = price;
11.    }
12. }
13.
14. public class JavaStreamExample {
15.     public static void main(String[] args) {
16.         List<Product> productsList = new ArrayList<Product>();
17.
18.         //Adding Products
19.         productsList.add(new Product(1,"HP Laptop",25000f));
20.         productsList.add(new Product(2,"Dell Laptop",30000f));
21.         productsList.add(new Product(3,"Lenevo Laptop",28000f));
22.         productsList.add(new Product(4,"Sony Laptop",28000f));
23.         productsList.add(new Product(5,"Apple Laptop",90000f));
24.
25.         // Converting product List into Set
26.         Set<Float> productPriceList =
27.             productsList.stream()
28.                 .filter(product->product.price < 30000) // filter product on the base of price
29.                 .map(product->product.price)
30.                 .collect(Collectors.toSet()); // collect it as Set(remove duplicate elements)
31.         System.out.println(productPriceList);
32.     }
33. }

```

Output:

```
[25000.0, 28000.0]
```

Java Stream Example : Convert List into Map

```

1. import java.util.*;
2. import java.util.stream.Collectors;
3. class Product{
4.     int id;
5.     String name;
6.     float price;
7.     public Product(int id, String name, float price) {
8.         this.id = id;
9.         this.name = name;
10.        this.price = price;
11.    }
12. }
13.
14. public class JavaStreamExample {
15.     public static void main(String[] args) {
16.         List<Product> productsList = new ArrayList<Product>();
17.
18.         //Adding Products
19.         productsList.add(new Product(1,"HP Laptop",25000f));
20.         productsList.add(new Product(2,"Dell Laptop",30000f));
21.         productsList.add(new Product(3,"Lenevo Laptop",28000f));
22.         productsList.add(new Product(4,"Sony Laptop",28000f));
23.         productsList.add(new Product(5,"Apple Laptop",90000f));
24.
25.         // Converting Product List into a Map
26.         Map<Integer,String> productPriceMap =
27.             productsList.stream()
28.                 .collect(Collectors.toMap(p->p.id, p->p.name));
29.
30.         System.out.println(productPriceMap);
31.     }
32. }

```

Output:

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

Method Reference in stream

```

1. import java.util.*;
2. import java.util.stream.Collectors;
3.
4. class Product{
5.     int id;

```



```

6.   String name;
7.   float price;
8.
9.   public Product(int id, String name, float price) {
10.      this.id = id;
11.      this.name = name;
12.      this.price = price;
13.  }
14.
15.  public int getId() {
16.      return id;
17.  }
18.  public String getName() {
19.      return name;
20.  }
21.  public float getPrice() {
22.      return price;
23.  }
24. }
25.
26. public class JavaStreamExample {
27.
28.     public static void main(String[] args) {
29.
30.         List<Product> productsList = new ArrayList<Product>();
31.
32.         //Adding Products
33.         productsList.add(new Product(1,"HP Laptop",25000f));
34.         productsList.add(new Product(2,"Dell Laptop",30000f));
35.         productsList.add(new Product(3,"Lenevo Laptop",28000f));
36.         productsList.add(new Product(4,"Sony Laptop",28000f));
37.         productsList.add(new Product(5,"Apple Laptop",90000f));
38.
39.         List<Float> productPriceList =
40.             productsList.stream()
41.                 .filter(p -> p.price > 30000) // filtering data
42.                 .map(Product::getPrice)      // fetching price by referring getPrice method
43.                 .collect(Collectors.toList()); // collecting as list
44.         System.out.println(productPriceList);
45.     }
46. }

```

Output:

```
[90000.0]
```

Java Stream Filter

Java stream provides a method filter() to filter stream elements on the basis of given predicate. Suppose you want to get only even elements of your list then you can do this easily with the help of filter method.

This method takes predicate as an argument and returns a stream of consisting of resulted elements.

Signature

The signature of Stream filter() method is given below:

1. Stream<T> filter(Predicate<? super T> predicate)

Parameter

predicate: It takes Predicate reference as an argument. Predicate is a functional interface. So, you can also pass lambda expression here.

Return

It returns a new stream.

Java Stream filter() example

In the following example, we are fetching and iterating filtered data.

```
1. import java.util.*;
2. class Product{
3.     int id;
4.     String name;
5.     float price;
6.     public Product(int id, String name, float price) {
7.         this.id = id;
8.         this.name = name;
9.         this.price = price;
10.    }
11. }
12. public class JavaStreamExample {
13.     public static void main(String[] args) {
14.         List<Product> productsList = new ArrayList<Product>();
15.         //Adding Products
16.         productsList.add(new Product(1,"HP Laptop",25000f));
17.         productsList.add(new Product(2,"Dell Laptop",30000f));
18.         productsList.add(new Product(3,"Lenevo Laptop",28000f));
19.         productsList.add(new Product(4,"Sony Laptop",28000f));
20.         productsList.add(new Product(5,"Apple Laptop",90000f));
21.         productsList.stream()
```

```

22.         .filter(p ->p.price> 30000) // filtering price
23.         .map(pm ->pm.price)        // fetching price
24.         .forEach(System.out::println); // iterating price
25.     }
26. }

```

Output:

90000.0

Java Stream filter() example 2

In the following example, we are fetching filtered data as a list.

```

1. import java.util.*;
2. import java.util.stream.Collectors;
3. class Product{
4.     int id;
5.     String name;
6.     float price;
7.     public Product(int id, String name, float price) {
8.         this.id = id;
9.         this.name = name;
10.        this.price = price;
11.    }
12. }
13. public class JavaStreamExample {
14.     public static void main(String[] args) {
15.         List<Product> productsList = new ArrayList<Product>();
16.         //Adding Products
17.         productsList.add(new Product(1,"HP Laptop",25000f));
18.         productsList.add(new Product(2,"Dell Laptop",30000f));
19.         productsList.add(new Product(3,"Lenevo Laptop",28000f));
20.         productsList.add(new Product(4,"Sony Laptop",28000f));
21.         productsList.add(new Product(5,"Apple Laptop",90000f));
22.         List<Float> pricesList = productsList.stream()
23.             .filter(p ->p.price> 30000) // filtering price
24.             .map(pm ->pm.price)        // fetching price
25.             .collect(Collectors.toList());
26.         System.out.println(pricesList);
27.     }
28. }

```

Output:

[90000.0]

reduction operations OR Stream.reduce() in Java with examples

A *reduction* is a terminal operation that aggregates a stream into a type or a primitive. The Java 8 Stream API contains a set of predefined reduction operations, such as `average`, `sum`, `min`, `max`, and `count`, which return one value by combining the elements of a stream.

Java Stream reduce method

`Stream.reduce` is a general-purpose method for generating our custom reduction operations.

This method performs a reduction on the elements of this stream, using an associative accumulation function. It returns an `Optional` describing the reduced value, if any.

This method takes two parameters: the identity and the accumulator. The identity element is both the initial value of the reduction and the default result if there are no elements in the stream. The accumulator function takes two parameters: a partial result of the reduction and the next element of the stream. It returns a new partial result. The `Stream.reduce` method returns the result of the reduction.

Stream.reduce() in Java with examples

- Difficulty Level : [Medium](#)
- Last Updated : 16 Oct, 2019

Many times, we need to perform operations where a stream reduces to single resultant value, for example, maximum, minimum, sum, product, etc. Reducing is the repeated process of combining all elements.

reduce operation applies a binary operator to each element in the stream where the first argument to the operator is the return value of the previous application and second argument is the current stream element.

Syntax :

```
T reduce(T identity, BinaryOperator<T> accumulator);
```

Where, **identity** is initial value of type **T** and **accumulator** is a function for combining two values.

sum(), min(), max(), count() etc. are some examples of reduce operations. reduce() explicitly asks you to specify how to reduce the data that made it through the stream.

Let us see some examples to understand the reduce() function in a better way :

Example 1 :

```
// Implementation of reduce method
// to get the longest String
import java.util.*;

class GFG {

    // Driver code
    public static void main(String[] args)
    {
        // creating a list of Strings
        List<String> words = Arrays.asList("GFG", "Geeks", "for",
                                           "GeeksQuiz", "GeeksforGeeks");

        // The lambda expression passed to
        // reduce() method takes two Strings
        // and returns the longer String.
        // The result of the reduce() method is
        // an Optional because the list on which
        // reduce() is called may be empty.
        Optional<String> longestString = words.stream()
                                              .reduce((word1, word2)
                                              -> word1.length() > word2.length()
                                              ? word1 : word2);

        // Displaying the longest String
        longestString.ifPresent(System.out::println);
    }
}
```

```
    }  
}
```

Output :

GeeksforGeeks

Example 2 :

```
// Implementation of reduce method  
// to get the combined String  
import java.util.*;  
  
class GFG {  
  
    // Driver code  
    public static void main(String[] args)  
    {  
  
        // String array  
        String[] array = { "Geeks", "for", "Geeks" };  
  
        // The result of the reduce() method is  
        // an Optional because the list on which  
        // reduce() is called may be empty.  
        Optional<String> String_combine = Arrays.stream(array)  
                                                    .reduce((str1, str2)  
                                                        -> str1 + "-" + str2);  
  
        // Displaying the combined String  
        if (String_combine.isPresent()) {  
            System.out.println(String_combine.get());  
        }  
    }  
}
```

Output :

Geeks-for-Geeks

Example 3 :

```
// Implementation of reduce method
// to get the sum of all elements
import java.util.*;

class GFG {

    // Driver code
    public static void main(String[] args)
    {

        // Creating list of integers
        List<Integer> array = Arrays.asList(-2, 0, 4, 6, 8);

        // Finding sum of all elements
        int sum = array.stream().reduce(0,
            (element1, element2) -> element1 + element2);

        // Displaying sum of all elements
        System.out.println("The sum of all elements is " + sum);
    }
}
```

Output :

The sum of all elements is 16

Java Interface **Splitterator**

Java Splitterator is one of the four iterators – Enumeration, [Iterator](#), [ListIterator](#) and Splitterator.

Java Splitterator

Like Iterator and ListIterator, Splitterator is a Java Iterator, which is used to iterate elements one-by-one from a [List](#) implemented object.

Some important points about Java Splitterator are:

1. Java Splitterator is an interface in [Java Collection](#) API.
2. Splitterator is introduced in [Java 8](#) release in `java.util` package.
3. It supports Parallel Programming functionality.
4. We can use it for both Collection API and [Stream](#) API classes.
5. It provides characteristics about Collection or API objects.
6. We can NOT use this Iterator for [Map](#) implemented classes.
7. It uses `tryAdvance()` method to iterate elements individually in multiple [Threads](#) to support Parallel Processing.
8. It uses `forEachRemaining()` method to iterate elements sequentially in a single Thread.
9. It uses `trySplit()` method to divide itself into Sub-Splitterators to support Parallel Processing.
10. Splitterator supports both Sequential and Parallel processing of data.

Splitterator itself does not provide the parallel programming behavior. However, it provides some methods to support it. Developers should utilize Splitterator interface methods and implement parallel programming by using Fork/Join Framework (one good approach).

Main Functionalities of Splitterator

- Splitting the source data.
- Processing the source data.

Java Splititerator Methods

In this section, we will list out all Java Splititerator methods one by one with some useful description.

1. `int characteristics()`: Returns a set of characteristics of this Splititerator and its elements.
2. `long estimateSize()`: Returns an estimate of the number of elements that would be encountered by a `forEachRemaining()` traversal, or returns `Long.MAX_VALUE` if infinite, unknown, or too expensive to compute.
3. `default void forEachRemaining(Consumer action)`: Performs the given action for each remaining element, sequentially in the current thread, until all elements have been processed or the action throws an exception.
4. `default Comparator getComparator()`: If this Splititerator's source is SORTED by a Comparator, returns that Comparator.
5. `default long getExactSizeIfKnown()`: Convenience method that returns `estimateSize()` if this Splititerator is SIZED, else -1.
6. `default boolean hasCharacteristics(int characteristics)`: Returns true if this Splititerator's characteristics() contain all of the given characteristics.
7. `boolean tryAdvance(Consumer action)`: If a remaining element exists, performs the given action on it, returning true; else returns false.
8. `Splititerator trySplit()`: If this splititerator can be partitioned, returns a Splititerator covering elements, that will, upon return from this method, not be covered by this Splititerator.

Java Splititerator Example

In this section, we will discuss about how to create Java Splititerator object using `splititerator()` and will develop simple example.

```
import java.util.Splititerator;
import java.util.ArrayList;
import java.util.List;

public class SplititeratorSequentialIteration
{
    public static void main(String[] args)
    {
        List<String> names = new ArrayList<>();
        names.add("Rams");
        names.add("Posa");
        names.add("Chinni");

        // Getting Splititerator
        Splititerator<String> namesSplititerator = names.splititerator();

        // Traversing elements
        namesSplititerator.forEachRemaining(System.out::println);
    }
}
```

Output:-

Rams
Posa
Chinni

If we observe the above program and output, we can easily understand that this `Splitterator.forEachRemaining()` method works in the same way as [ArrayList.forEach\(\)](#). Yes, both works in similar way.

Advantages of Spliterator

1. Unlike Iterator and ListIterator, it supports Parallel Programming functionality.
2. Unlike Iterator and ListIterator, it supports both Sequential and Parallel Processing of data.
3. Compare to other Iterators, it provides better performance.

Iterator vs Spliterator

Iterator	Spliterator
Introduced in Java 1.2.	Introduced in Java 1.8.
It is an Iterator for whole Collection API.	It is an Iterator for both Collection and Stream API, except Map implemented classes.
It is an Universal Iterator.	It is NOT an Universal Iterator.
It does NOT support Parallel Programming.	It supports Parallel Programming.

That's all about Spliterator in Java.

```
1. import java.util.*;
2. import java.util.stream.Stream;
3. public class InterfaceSpliteratorExample1 {
4.     public static void main(String args[]){
5.         //Create an object of array list
6.         ArrayList<Integer> list = new ArrayList<>();
7.         //Add values to the array list.
8.         list.add(101);
9.         list.add(201);
10.        list.add(301);
11.        list.add(401);
12.        list.add(501);
```

```

13.    Stream<Integer> str = list.stream();
14.    //Get Spliterator object on list
15.    Spliterator<Integer> splitr = str.spliterator();
16.    //Get estimateSize method
17.    System.out.println("Estimate size: " + splitr.estimateSize());
18.    //Print getExactSizeKnown method
19.    System.out.println("Exact size: " + splitr.getExactSizeKnown());
20.    //Check for hasCharacteristics and characteristics method
21.    System.out.println("Boolean Result: "+splitr.hasCharacteristics(splitr.characteristics()));

22.    System.out.println("Elements of ArrayList :");
23.    //Obtain result forEachRemaining method
24.    splitr.forEachRemaining((n) -> System.out.println(n));
25.    //Obtaining another Stream to the array list.
26.    Stream<Integer> str1 = list.stream();
27.    splitr = str1.spliterator();
28.    //Obtain result from trySplit() method
29.    Spliterator<Integer> splitr2 = splitr.trySplit();
30.    //If splitr could be split, use splitr2 first.
31.    if(splitr2 != null) {
32.        System.out.println("Output from splitr2: ");
33.        splitr2.forEachRemaining((n) -> System.out.println(n));
34.    }
35.    //Now, use the splitr
36.    System.out.println("Output from splitr1: ");
37.    splitr.forEachRemaining((n) -> System.out.println(n));
38.    }
39. }

```

[Test it Now](#)

Output:

```

Estimate size: 5
Exact size: 5
Boolean Result: true
Elements of ArrayList :
101
201
301
401
501
Output from splitr2:
101
201
Output from splitr1:
301
401
501

```

[Example 2](#)

```

1. import java.util.*;
2. public class InterfaceSpliteratorExample2 {
3.     public static void main(String args[]){
4.         List<String> fruitList = Arrays.asList("Mango", "Banana", "Apple");
5.         Spliterator<String> splitr = fruitList.spliterator();
6.         System.out.println("List of Fruit name-");
7.         while(splitr.tryAdvance((num) -> System.out.println("" +num)));
8.     }
9. }

```

[Test it Now](#)

Output:

```

List of Fruit name-
Mango
Banana
Apple

```

Example 3

```

1. import java.util.*;
2. public class InterfaceSpliteratorExample3 {
3.     public static void main (String[] args) {
4.         List<String> progList = Arrays.asList("Java","Android","Python","C++");
5.         Spliterator<String> splitr = progList.spliterator();
6.         Spliterator<String> st = splitr.trySplit();
7.         splitr.forEachRemaining(System.out::println);
8.         System.out.println("Traversing the next half of the spliterator-");
9.         st.forEachRemaining(System.out::println);
10.    }
11. }

```

[Test it Now](#)

Output:

```

Python
C++
Traversing the next half of the spliterator-
Java

```

Stream mapping in Java

Stream map(Function mapper) returns a stream consisting of the results of applying the given function to the elements of this stream.

Stream map(Function mapper) is an **intermediate operation**. These operations are always lazy. Intermediate operations are invoked on a Stream instance and after they finish their processing, they give a Stream instance as output.

Syntax :

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper)
```

where, R is the element type of the new stream.
Stream is an interface and T is the type of stream elements. mapper is a stateless function which is applied to each element and the function returns the new stream.

Example 1 : Stream map() function with operation of number * 3 on each element of stream.

```
// Java code for Stream map(Function mapper)
// to get a stream by applying the
// given function to this stream.
import java.util.*;

class GFG {

    // Driver code
    public static void main(String[] args)
    {

        System.out.println("The stream after applying "
                           + "the function is : ");

        // Creating a list of Integers
        List<Integer> list = Arrays.asList(3, 6, 9, 12, 15);

        // Using Stream map(Function mapper) and
        // displaying the corresponding new stream
        list.stream().map(number -> number *
3).forEach(System.out::println);
    }
}
```

Output :

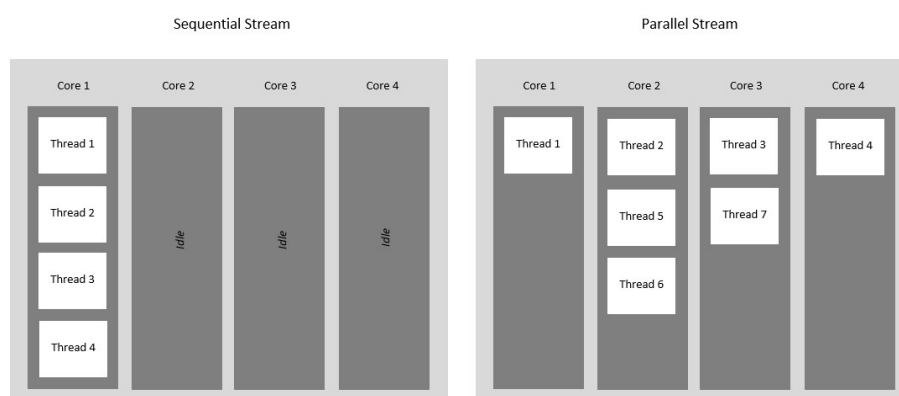
The stream after applying the function is :

9
18
27
36
45

What is Java Parallel Streams?

Java Parallel Streams is a feature of Java 8 and higher, meant for utilizing multiple cores of the processor. Normally any java code has one stream of processing, where it is executed sequentially. Whereas by using parallel streams, we can divide the code into multiple streams that are executed in parallel on separate cores and the final result is the combination of the individual outcomes. The order of execution, however, is not under our control.

Therefore, it is advisable to use parallel streams in cases where no matter what is the order of execution, the result is unaffected and the state of one element does not affect the other as well as the source of the data also remains unaffected.



Why Parallel Streams?

Parallel Streams were introduced to increase the performance of a program,

Ways to Create Stream

There are two ways we can create which are listed below and described later as follows:

1. Using `parallel()` method on a stream
2. Using `parallelStream()` on a Collection

Method 1: Using `parallel()` method on a stream

The **parallel() method** of the **BaseStream interface** returns an equivalent parallel stream. Let us explain how it would work with the help of an example.

In the code given below, we create a file object which points to a pre-existent 'txt' file in the system. Then we create a Stream that reads from the text file one line at a time. Then we use the **parallel() method** to print the read file on the console. The order of execution is different for each run, you can observe this in the output. The two outputs given below have different orders of execution.

Example

```
// Java Program to Illustrate Parallel Streams
// Using parallel() method on a Stream

// Importing required classes
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.util.stream.Stream;

// Main class
// ParallelStreamTest
public class GFG {

    // Main driver method
    public static void main(String[] args) throws IOException {

        // Creating a File object
        File fileName = new File("M:\\Documents\\Textfile.txt");

        // Create a Stream of string type
        // using the lines() method to
        // read one line at a time from the text file
        Stream<String> text = Files.lines(fileName.toPath());

        // Creating parallel streams using parallel() method
        // later using forEach() to print on console
```

```

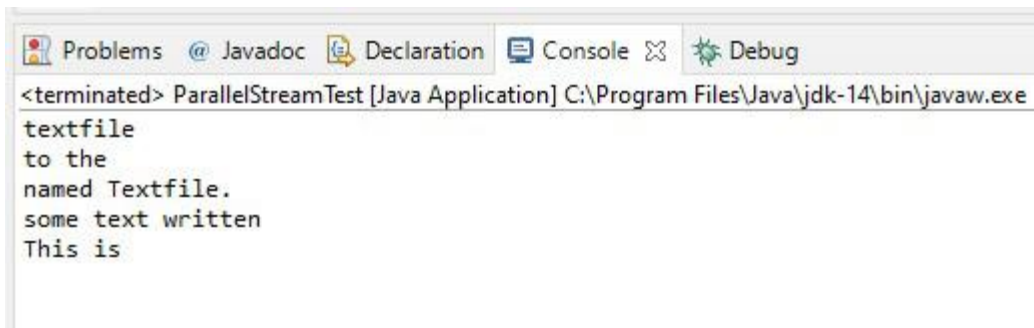
        text.parallel().forEach(System.out::println);

        // Closing the Stream
        // using close() method
        text.close();
    }
}

```

Output:

1A



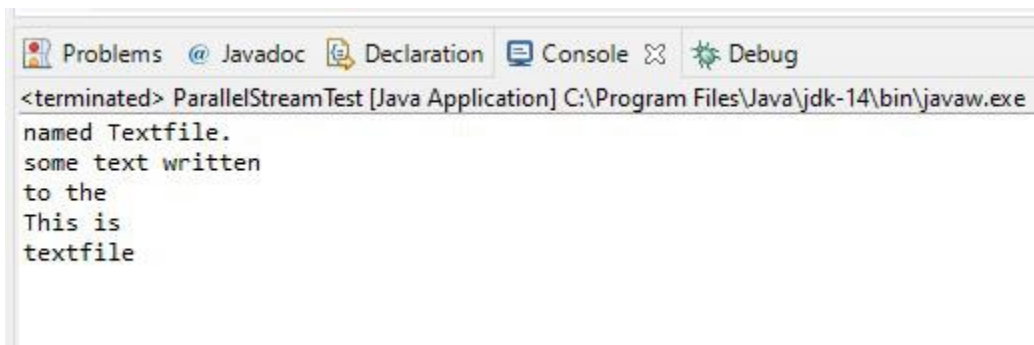
```

<terminated> ParallelStreamTest [Java Application] C:\Program Files\Java\jdk-14\bin\javaw.exe
textfile
to the
named Textfile.
some text written
This is

```

Output 1

1B



```

<terminated> ParallelStreamTest [Java Application] C:\Program Files\Java\jdk-14\bin\javaw.exe
named Textfile.
some text written
to the
This is
textfile

```

Output 2

Method 2: Using parallelStream() on a Collection

The **parallelStream()** method of the [Collection interface](#) returns a possible parallel stream with the collection as the source. Let us explain the working with the help of an example.

Implementation:

In the code given below, we are again using parallel streams but here we are using a List to read from the text file. Therefore, we need the *parallelStream()* method.

Example

```
// Java Program to Illustrate Parallel Streams
// using parallelStream() method on a Stream

// Importing required classes
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.util.*;

// Main class
// ParallelStreamsTest
public class GFG {

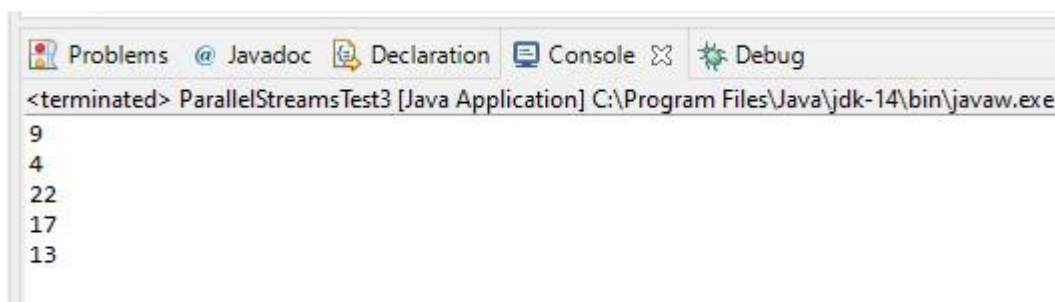
    // Main driver method
    public static void main(String[] args)
        throws IOException
    {

        // Creating a File object
        File fileName
            = new File("M:\\Documents\\List_Textfile.txt");

        // Reading the lines of the text file by
        // create a List using readAllLines() method
        List<String> text
            = Files.readAllLines(fileName.toPath());

        // Creating parallel streams by creating a List
        // using readAllLines() method
        text.parallelStream().forEach(System.out::println);
    }
}
```

Output:



```
<terminated> ParallelStreamsTest3 [Java Application] C:\Program Files\Java\jdk-14\bin\javaw.exe
9
4
22
17
13
```

Output