

Java 8 Stream API Guide

1. Overview

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

1.1 The Stream provides the following features:

- Stream* does not store elements. It simply conveys elements from a source such as a data structure, an array, or an I/O channel, through a pipeline of computational operations.
- Stream* is functional in nature. Operations performed on a stream does not modify its 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.

<div><<Java Interface>></div> <div><div><div>Stream<T></div><div>java.util.stream</div></div></div>
<div><div><div>filter(Predicate<? super T>):Stream<T></div><div>map(Function<? super T,? extends R>):Stream<R></div><div>mapToInt(ToIntFunction<? super T>):IntStream</div><div>mapToLong(ToLongFunction<? super T>):LongStream</div><div>mapToDouble(ToDoubleFunction<? super T>):DoubleStream</div><div>flatMap(Function<? super T,Stream<? extends R>>):Stream<R></div><div>flatMapToInt(Function<? super T,IntStream>):IntStream</div><div>flatMapToLong(Function<? super T,LongStream>):LongStream</div><div>flatMapToDouble(Function<? super T,DoubleStream>):DoubleStream</div><div>distinct():Stream<T></div><div>sorted():Stream<T></div><div>sorted(Comparator<? super T>):Stream<T></div><div>peek(Consumer<? super T>):Stream<T></div><div>limit(long):Stream<T></div><div>skip(long):Stream<T></div><div>forEach(Consumer<? super T>):void</div><div>forEachOrdered(Consumer<? super T>):void</div><div>toArray():Object[]</div><div>toArray(IntFunction<A[]>):A[]</div><div>reduce(T,BinaryOperator<T>):T</div><div>reduce(BinaryOperator<T>):Optional<T></div><div>reduce(U,BIfFunction<U,? super T,U>,BinaryOperator<U>):U</div><div>collect(Supplier<R>,BiConsumer<R,? super T>,BiConsumer<R,R>):R</div><div>collect(Collector<? super T,A,R>):R</div><div>min(Comparator<? super T>):Optional<T></div><div>max(Comparator<? super T>):Optional<T></div><div>count():long</div><div>anyMatch(Predicate<? super T>):boolean</div><div>allMatch(Predicate<? super T>):boolean</div><div>noneMatch(Predicate<? super T>):boolean</div><div>findFirst():Optional<T></div><div>findAny():Optional<T></div><div>builder():Builder<T></div><div>empty():Stream<T></div></div></div>

2. Stream API's with Examples

Let's explore Stream API's with examples

2.1 Java Example: Filtering Collection without using Stream

In the following example, we are filtering data without using a stream. This approach we are used before the stream package was released. First create a *Product* class, which is used in below examples :

```
public class Product {
    private int id;
    private String name;
    private float price;
    public Product(int id, String name, float price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
}
```

```

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public float getPrice() {
        return price;
    }
    public void setPrice(float price) {
        this.price = price;
    }
}

```

Let's first discuss without using *Stream API's* examples then we will create the same examples using *Stream API's*.

```

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class JavaWithoutStreamExample {
    private static List < Product > productsList = new ArrayList < Product > ();

    public static void main(String[] args) {

        // Adding Products
        productsList.add(new Product(1, "HP Laptop", 25000 f));
        productsList.add(new Product(2, "Dell Laptop", 30000 f));
        productsList.add(new Product(3, "Lenevo Laptop", 28000 f));
        productsList.add(new Product(4, "Sony Laptop", 28000 f));
        productsList.add(new Product(5, "Apple Laptop", 90000 f));
        // Without Java 8 Stream API'S
        withoutStreamAPI();
    }

    private static void withoutStreamAPI() {
        // without Stream API's
        List < Float > productPriceList = new ArrayList < Float > ();
        // filtering data of list
        for (Product product: productsList) {
            if (product.getPrice() > 25000) {
                // adding price to a productPriceList
                productPriceList.add(product.getPrice());
            }
        }

        // displaying data
        for (Float price: productPriceList) {
            System.out.println(price);
        }
    }
}

```

2.2 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. The stream provides fast execution.

```

import java.util.ArrayList;
import java.util.List;

```

```

import java.util.stream.Collectors;

public class JavaStreamExample {
    private static List<Product> productsList = new ArrayList<Product>();

    public static void main(String[] args) {

        // Adding Products
        productsList.add(new Product(1, "HP Laptop", 25000f));
        productsList.add(new Product(2, "Dell Laptop", 30000f));
        productsList.add(new Product(3, "Lenevo Laptop", 28000f));
        productsList.add(new Product(4, "Sony Laptop", 28000f));
        productsList.add(new Product(5, "Apple Laptop", 90000f));
        // With Java 8 Stream API'S
        withStreamAPI();
    }

    private static void withStreamAPI() {
        // filtering data of list
        List<Float> productPriceList = productsList.stream()
            .filter((product) -> product.getPrice() > 25000)
            .map((product) -> product.getPrice()).collect(Collectors.toList());
        // displaying data
        productPriceList.forEach((price) -> System.out.println(price));
    }
}

```

2.3 Java *Stream* Example: Filtering and Iterating *Collection*

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

```

public class FilteringAndIteratingCollection {
    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();
        // Adding Products
        productsList.add(new Product(1, "HP Laptop", 25000f));
        productsList.add(new Product(2, "Dell Laptop", 30000f));
        productsList.add(new Product(3, "Lenevo Laptop", 28000f));
        productsList.add(new Product(4, "Sony Laptop", 28000f));
        productsList.add(new Product(5, "Apple Laptop", 90000f));
        // This is more compact approach for filtering data
        productsList.stream().filter(product -> product.getPrice() == 30000)
            .forEach(product -> System.out.println(product.getPrice()));
    }
}

```

2.4 Java *Stream* Example: Sum by using *Collectors* Methods

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

```

public class SumByUsingCollectorsMethods {
    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();
        //Adding Products
        productsList.add(new Product(1,"HP Laptop",25000f));
        productsList.add(new Product(2,"Dell Laptop",30000f));
        productsList.add(new Product(3,"Lenevo Laptop",28000f));
        productsList.add(new Product(4,"Sony Laptop",28000f));
        productsList.add(new Product(5,"Apple Laptop",90000f));
    }
}

```

```

        // Using Collectors's method to sum the prices.
        double totalPrice3 = productList.stream()
            .collect(Collectors.summingDouble(product -> product.getPrice()));
        System.out.println(totalPrice3);
    }
}

```

2.5 Java *Stream* Example: Find Max and Min Product Price

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

```

public class FindMaxAndMinMethods {
    public static void main(String[] args) {
        List<Product> productList = new ArrayList<Product>();
        // Adding Products
        productList.add(new Product(1, "HP Laptop", 25000f));
        productList.add(new Product(2, "Dell Laptop", 30000f));
        productList.add(new Product(3, "Lenevo Laptop", 28000f));
        productList.add(new Product(4, "Sony Laptop", 28000f));
        productList.add(new Product(5, "Apple Laptop", 90000f));
        // max() method to get max Product price
        Product productA = productList
            .stream().max((product1,
                product2) -> product1.getPrice() > product2.getPrice() ? 1 : -1)
            .get();

        System.out.println(productA.getPrice());
        // min() method to get min Product price
        Product productB = productList
            .stream().max((product1,
                product2) -> product1.getPrice() < product2.getPrice() ? 1 : -1)
            .get();
        System.out.println(productB.getPrice());
    }
}

```

2.6 Java *Stream* Example: Convert *List* into *Set*

```

public class ConvertListToSet {
    public static void main(String[] args) {
        List<Product> productList = new ArrayList<Product>();

        // Adding Products
        productList.add(new Product(1, "HP Laptop", 25000f));
        productList.add(new Product(2, "Dell Laptop", 30000f));
        productList.add(new Product(3, "Lenevo Laptop", 28000f));
        productList.add(new Product(4, "Sony Laptop", 28000f));
        productList.add(new Product(5, "Apple Laptop", 90000f));

        // Converting product List into Set
        Set<Float> productPriceList = productList.stream()
            .filter(product -> product.getPrice() < 30000)
            .map(product -> product.getPrice())
            .collect(Collectors.toSet());

        System.out.println(productPriceList);
    }
}

```

```
}
```

2.7 Java *Stream* Example: Convert *List* into *Map*

```
public class ConvertListToMap {
    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();

        // Adding Products
        productsList.add(new Product(1, "HP Laptop", 25000f));
        productsList.add(new Product(2, "Dell Laptop", 30000f));
        productsList.add(new Product(3, "Lenevo Laptop", 28000f));
        productsList.add(new Product(4, "Sony Laptop", 28000f));
        productsList.add(new Product(5, "Apple Laptop", 90000f));

        // Converting Product List into a Map
        Map<Integer, String> productPriceMap = productsList.stream()
            .collect(Collectors.toMap(p -> p.getId(), p -> p.getName()));
        System.out.println(productPriceMap);
    }
}
```

2.8 Using Method References in *Stream* Examples

```
public class MethodReferenceInStream {
    public static void main(String[] args) {

        List<Product> productsList = new ArrayList<Product>();

        // Adding Products
        productsList.add(new Product(1, "HP Laptop", 25000f));
        productsList.add(new Product(2, "Dell Laptop", 30000f));
        productsList.add(new Product(3, "Lenevo Laptop", 28000f));
        productsList.add(new Product(4, "Sony Laptop", 28000f));
        productsList.add(new Product(5, "Apple Laptop", 90000f));

        List<Float> productPriceList = productsList.stream()
            .filter(p -> p.getPrice() > 30000)// filtering data
            .map(Product::getPrice) // fetching price by referring getPrice method
            .collect(Collectors.toList()); // collecting as list
        System.out.println(productPriceList);
    }
}
```

3. Reference

- [Java 8 Stream API JavaDoc](#)

4. Online Article of this Guide

- <http://www.javaguides.net/2018/07/java-8-stream-api.html>

5. Java 8 Tutorial

- [Top Java 8 Tutorial](#)