**JAVA-8 Features**

**Functional Interface (SAM/Single Abstract Method Interface)**

* **An interface that contains exactly one abstract method is known as Functional Interface.**
* Java provides predefined functional interfaces to deal with functional programming by using lambda and method references.
* It can have any number of default, static methods but can contain only one abstract method.
* It can also declare methods of object class.(Example-2)
* It helps to achieve Functional Programming Approach.
* A functional interface can extends to other interface only when that does not have any abstract method. (Example-3)
* If a Functional Interface extends another interface having an abstract method, then compiler will throw a compile time error. (Example-4)

**Example1**

**package** FunctionalInterface;

@FunctionalInterface

**interface** message{

**void** alert (String msg);

}

**public** **class** FunctionalInterfaceTest1 **implements** message{

**public** **void** alert(String message){

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

}

**public** **static** **void** main(String[] args) {

FunctionalInterfaceTest1 fit = **new** FunctionalInterfaceTest1();

fit.alert("Hello World");

}

}

**Example2**

**package** FunctionalInterface;

@FunctionalInterface

**interface** sayable{

**void** say(String msg); // abstract method

// It can contain any number of methods of Object class.

**int** hashCode();

String toString();

**boolean** equals(Object obj);

}

**public** **class** FunctionalInterfacewithObjectClassMethods **implements** sayable{

**public** **void** say(String msg){

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

}

**public** **static** **void** main(String[] args) {

FunctionalInterfacewithObjectClassMethods fie = **new** FunctionalInterfacewithObjectClassMethods();

fie.say ("Hello World");

}

}

**Example3**

**package** FunctionalInterface;

//Non-Functional Interface

**interface** doable{

**default** **void** doIt(){

System.***out***.println("Do it now");

}

}

@FunctionalInterface

**interface** sayable1 **extends** doable{

**void** say(String msg); // abstract method

}

**public** **class** FunctionalInterfaceExtendingNonFunctionalInterface **implements** sayable1{

**public** **void** say (String msg){

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

}

**public** **static** **void** main (String[] args) {

FunctionalInterfaceExtendingNonFunctionalInterface fie = **new** FunctionalInterfaceExtendingNonFunctionalInterface();

fie.say("Hello World");

fie.doIt();

}

}

**Example4**

**package** FunctionalInterface;

**interface** sayable2{

**void** say(String msg); // abstract method

}

@FunctionalInterface

**interface** doable1 **extends** sayable{

// Invalid '@FunctionalInterface' annotation; doable1 is not a functional interface

**void** doIt();

}

**public** **class** FunctionalInterfaceImplementingNonFuctionaInterface {

**public** **static** **void** main(String[] args) {

}

}

**Output: Compiletime Error**

**Predefined Functional Interface**

* Java provides predefined functional interfaces to deal with functional programming by using lambda and method references.
* We can also define our own custom functional interface.
* Following is the list of functional interface which are placed in java.util.function package.

|  |  |
| --- | --- |
| **Interface** | **Description** |
| [BiConsumer<T,U>](https://www.javatpoint.com/java-biconsumer-interface) | It represents an operation that accepts two input arguments and returns no result. |
| [Consumer<T>](https://www.javatpoint.com/java-consumer-interface) | It represents an operation that accepts a single argument and returns no result. |
| [Function<T,R>](https://www.javatpoint.com/java-function-interface) | It represents a function that accepts one argument and returns a result. |
| [Predicate<T>](https://www.javatpoint.com/java-predicate-interface) | It represents a predicate (boolean-valued function) of one argument and returns boolean value. |
| BiFunction<T,U,R> | It represents a function that accepts two arguments and returns a result. |
| BinaryOperator<T> | It represents an operation upon two operands of the same data type. It returns a result of the same type as the operands. |
| BiPredicate<T,U> | It represents a predicate (boolean-valued function) of two arguments. |
| BooleanSupplier | It represents a supplier of boolean-valued results. |
| DoubleBinaryOperator | It represents an operation upon two double type operands and returns a double type value. |
| DoubleConsumer | It represents an operation that accepts a single double type argument and returns no result. |
| DoubleFunction<R> | It represents a function that accepts a double type argument and produces a result. |
| DoublePredicate | It represents a predicate (boolean-valued function) of one double type argument. |
| DoubleSupplier | It represents a supplier of double type results. |
| DoubleToIntFunction | It represents a function that accepts a double type argument and produces an int type result. |
| DoubleToLongFunction | It represents a function that accepts a double type argument and produces a long type result. |
| DoubleUnaryOperator | It represents an operation on a single double type operand that produces a double type result. |
| IntBinaryOperator | It represents an operation upon two int type operands and returns an int type result. |
| IntConsumer | It represents an operation that accepts a single integer argument and returns no result. |
| IntFunction<R> | It represents a function that accepts an integer argument and returns a result. |
| IntPredicate | It represents a predicate (boolean-valued function) of one integer argument. |
| IntSupplier | It represents a supplier of integer type. |
| IntToDoubleFunction | It represents a function that accepts an integer argument and returns a double. |
| IntToLongFunction | It represents a function that accepts an integer argument and returns a long. |
| IntUnaryOperator | It represents an operation on a single integer operand that produces an integer result. |
| LongBinaryOperator | It represents an operation upon two long type operands and returns a long type result. |
| LongConsumer | It represents an operation that accepts a single long type argument and returns no result. |
| LongFunction<R> | It represents a function that accepts a long type argument and returns a result. |
| LongPredicate | It represents a predicate (boolean-valued function) of one long type argument. |
| LongSupplier | It represents a supplier of long type results. |
| LongToDoubleFunction | It represents a function that accepts a long type argument and returns a result of double type. |
| LongToIntFunction | It represents a function that accepts a long type argument and returns an integer result. |
| LongUnaryOperator | It represents an operation on a single long type operand that returns a long type result. |
| ObjDoubleConsumer<T> | It represents an operation that accepts an object and a double argument, and returns no result. |
| ObjIntConsumer<T> | It represents an operation that accepts an object and an integer argument. It does not return result. |
| ObjLongConsumer<T> | It represents an operation that accepts an object and a long argument, it returns no result. |
| Supplier<T> | It represents a supplier of results. |
| ToDoubleBiFunction<T,U> | It represents a function that accepts two arguments and produces a double type result. |
| ToDoubleFunction<T> | It represents a function that returns a double type result. |
| ToIntBiFunction<T,U> | It represents a function that accepts two arguments and returns an integer. |
| ToIntFunction<T> | It represents a function that returns an integer. |
| ToLongBiFunction<T,U> | It represents a function that accepts two arguments and returns a result of long type. |
| ToLongFunction<T> | It represents a function that returns a result of long type. |
| UnaryOperator<T> | It represents an operation on a single operand that returnsa a result of the same type as its operand. |

**Consumer Interface** **(**[**Consumer<T>**](https://www.javatpoint.com/java-consumer-interface)**)**

* The Consumer Interface accepts a single argument and does not return any result.
* It is a functional interface defined in java.util.function package.
* It contains an abstract accept() and a default andThen() method.
* It can be used as the assignment target for a lambda expression or method reference.

**Methods**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void accept(T t) | It performs this operation on the given argument. |
| default Consumer<T> andThen(Consumer<? super T> after) | It returns a composed Consumer that performs, in sequence, this operation followed by the after operation. If performing either operation throws an exception, it is relayed to the caller of the composed operation. If performing this operation throws an exception, the after operation will not be performed. |

Any input type

Consumer<T>

**Example1**

**package** Consumer;

**import** java.util.function.Consumer;

**public** **class** ConsumerTest {

**public** **static** **void** main(String[] args) {

Consumer<String> c=s->System.***out***.println(s);

Consumer<Integer> c1=s->System.***out***.println(s);

c.accept("Hello");

c.accept("World");

c1.accept(10);

}

}

**Output:** Hello

World

10

**Example2**

**package** Consumer;

**import** java.util.function.Consumer;

**public** **class** ConsumerInterfaceTest2 {

**static** **void** printMessage(String name)

{

System.***out***.println("Hello "+name);

}

**static** **void** printValue(**int** val)

{

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

}

**public** **static** **void** main(String[] args) {

// Referring method to String type Consumer interface

Consumer<String> consumer1 = ConsumerInterfaceTest2::*printMessage*;

consumer1.accept("Smruti"); // Calling Consumer method

// Referring method to Integer type Consumer interface

Consumer<Integer> consumer2 = ConsumerInterfaceTest2::*printValue*;

consumer2.accept(12); // Calling Consumer method

}

}

**Output:** Hello Smruti

12

**BiConsumer Interface** **(**[**BiConsumer<T,U>**](https://www.javatpoint.com/java-biconsumer-interface)**)**

* It represents an operation that accepts two input arguments and does not return any result.
* This is the two-arity specialization of Consumer interface.
* It provides a functional method accept (Object, Object) to perform custom operations.

**Lambda Expression**

* It provides a clear & concise way to represent one method interface using an epression.
* It is very useful in collection library.
* It helps to iterate, filter & extract data from collection.
* It is a replacement of java inner anonymous class.
* Java lambda expression is treated as a function , so compiler does not create .class file.
* Lambda expression provides implementation of functional interface.
* **An interface which has only one abstract method is called functional interface**.
* Java provides an annotation **@FunctionalInterface**, which is used to declare an interface as functional Interface.
* Less Coding.

**Syntax for Lambda Expression**

**(argument-list)->{body};**

* Java lambda expression consists of 3 components :-

1. **Argument-list**: It can be empty or non-empty as well.
2. **Arrow-token:** Used to link argument list & body of expression.
3. **Body**: It consists expressions & statements of lambda expression.

**WithoutLambdaExample1**

**package** lambdaExpression;

**interface** Drawable1{

**public** **void** draw();

}

**public** **class** WithoutLambdaExample1 {

**public** **static** **void** main(String[] args) {

**int** width=30;

Drawable1 d=**new** Drawable1() {

@Override

**public** **void** draw() {

System.***out***.println("Width is:"+width);

}

};

d.draw();

}

}

**LambdaExpressionExample1**

**package** lambdaExpression;

@FunctionalInterface

**interface** Drawable{

**public** **void** draw();

}

**public** **class** LambdaExpressionExample1 {

**public** **static** **void** main(String[] args) {

**int** width=23;

Drawable d=()->{

System.***out***.println("Width is:"+width);

};

d.draw();

}

}

**LambdaExpressionSingleParameter**

**package** lambdaExpression;

@FunctionalInterface

**interface** Sayable{

**public** String say(String name);

}

**public** **class** LambdaExpressionSingleParameter {

**public** **static** **void** main(String[] args) {

Sayable s=(name)->{

**return** "Hello "+name;

};

System.***out***.println(s.say("smruti"));

}

}

**LambdaExpessionMultiParameter**

**package** lambdaExpression;

**interface** Addable{

**int** add(**int** a, **int** b);

}

**public** **class** LambdaExpessionMultiParameter {

**public** **static** **void** main(String[] args) {

Addable add1=(a,b)->(a+b);

System.***out***.println(add1.add(19, 23));

Addable add2=(**int** a, **int** b)->(a+b);

System.***out***.println(add2.add(4, 6));

}

}

**LambdaExpressionForEach**

**package** lambdaExpression;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** LambdaExpressionForEach {

**public** **static** **void** main(String[] args) {

List<String> list=**new** ArrayList<String>();

list.add("Smruti");

list.add("Ranjan");

list.add("Gouda");

list.forEach((n)->System.***out***.println(n));

}

}

**LambdaExpressionThread**

**package** lambdaExpression;

**public** **class** LambdaExpressionThread {

**public** **static** **void** main(String[] args) {

//Without lambda

Runnable r1=**new** Runnable() {

@Override

**public** **void** run() {

System.***out***.println("Thread withuot lambda is running........");

}

};

Thread t1=**new** Thread(r1);

t1.start();

//With Lambda

Runnable r2=()->{

System.***out***.println("Thread with lambda is running........");

};

Thread t2=**new** Thread(r2);

t2.start();

}

}

**LambdaExpressionforComparator**

**package** lambdaExpression;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.List;

**class** Product{

**int** id;

String name;

**float** price;

**public** Product(**int** id, String name, **float** price) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.price = price;

}

}

**public** **class** LambdaExpressionforComparator{

**public** **static** **void** main(String[] args) {

List<Product> list=**new** ArrayList<Product>();

//Adding Products

list.add(**new** Product(1,"HP Laptop",25000f));

list.add(**new** Product(3,"Keyboard",300f));

list.add(**new** Product(2,"Dell Mouse",150f));

System.***out***.println("Sorting on the basis of name...");

// implementing lambda expression

Collections.*sort*(list,(p1,p2)->{ **return** p1.name.compareTo(p2.name);

});

**for**(Product p:list){

System.***out***.println(p.id+" "+p.name+" "+p.price);

}

}

}

**LambdaExpressionforCollection**

**package** lambdaExpression;

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.stream.Stream;

**class** Product1{

**int** id;

String name;

**float** price;

**public** Product1(**int** id, String name, **float** price) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.price = price;

}

}

**public** **class** LambdaExpressionCollection{

**public** **static** **void** main(String[] args) {

List<Product1> list=**new** ArrayList<Product1>();

list.add(**new** Product1(1,"Samsung A5",17000f));

list.add(**new** Product1(3,"Iphone 6S",65000f));

list.add(**new** Product1(2,"Sony Xperia",25000f));

list.add(**new** Product1(4,"Nokia Lumia",15000f));

list.add(**new** Product1(5,"Redmi4 ",26000f));

list.add(**new** Product1(6,"Lenevo Vibe",19000f));

// using lambda to filter data

Stream<Product1> filtered\_data = list.stream().filter(p -> p.price > 20000);

// using lambda to iterate through collection

filtered\_data.forEach(

product1 -> System.***out***.println(product1.name+": "+product1.price)

);

}

}

**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 is a new abstract layer introduced in Java 8. Using stream, you can process data in a declarative way similar to SQL statements. For example, consider the following SQL statement −

SELECT max(salary), employee\_id, employee\_name FROM Employee

The above SQL expression automatically returns the maximum salaried employee's details, without doing any computation on the developer's end. Using collections framework in Java, a developer has to use loops and make repeated checks. Another concern is efficiency; as multi-core processors are available at ease, a Java developer has to write parallel code processing that can be pretty error-prone.

To resolve such issues, Java 8 introduced the concept of stream that lets the developer to process data declaratively and leverage multicore architecture without the need to write any specific code for it.

## What is Stream?

Stream represents a sequence of objects from a source, which supports aggregate operations. Following are the characteristics of a Stream −

* **Sequence of elements** − A stream provides a set of elements of specific type in a sequential manner. A stream gets/computes elements on demand. It never stores the elements.
* **Source** − Stream takes Collections, Arrays, or I/O resources as input source.
* **Aggregate operations** − Stream supports aggregate operations like filter, map, limit, reduce, find, match, and so on.
* **Pipelining** − Most of the stream operations return stream itself so that their result can be pipelined. These operations are called intermediate operations and their function is to take input, process them, and return output to the target. collect() method is a terminal operation which is normally present at the end of the pipelining operation to mark the end of the stream.
* **Automatic iterations** − Stream operations do the iterations internally over the source elements provided, in contrast to Collections where explicit iteration is required.

**Stream provides 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 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. In the following examples, we have apply various operations with the help of stream.

## Generating Streams

With Java 8, Collection interface has two methods to generate a Stream −

* **stream()** − Returns a sequential stream considering collection as its source.
* **parallelStream()** − Returns a parallel Stream considering collection as its source.

List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collec

## 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. |
| static <T> Stream.Builder<T> builder() | It returns a builder for a Stream. |
| <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. |
| <R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper) | 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.) |
| DoubleStream flatMapToDouble(Function<? super T,? extends DoubleStream> mapper) | 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.) |
| IntStream flatMapToInt(Function<? super T,? extends IntStream> mapper) | 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.) |
| LongStream flatMapToLong(Function<? super T,? extends LongStream> mapper) | 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.) |
| void forEach(Consumer<? super T> action) | It performs an action for each element of this stream. |
| void forEachOrdered(Consumer<? super T> action) | 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. |
| static <T> Stream<T> generate(Supplier<T> s) | 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){
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]

## forEach

Stream has provided a new method ‘forEach’ to iterate each element of the stream. The following code segment shows how to print 10 random numbers using forEach.

Random random = new Random();

random.ints().limit(10).forEach(System.out::println);

## map

The ‘map’ method is used to map each element to its corresponding result. The following code segment prints unique squares of numbers using map.

List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

//get list of unique squares

List<Integer> squaresList = numbers.stream().map( i -> i\*i).distinct().collect(Collectors.toList());

## filter

The ‘filter’ method is used to eliminate elements based on a criteria. The following code segment prints a count of empty strings using filter.

List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

//get count of empty string

int count = strings.stream().filter(string -> string.isEmpty()).count();

## limit

The ‘limit’ method is used to reduce the size of the stream. The following code segment shows how to print 10 random numbers using limit.

Random random = new Random();

random.ints().limit(10).forEach(System.out::println);

## sorted

The ‘sorted’ method is used to sort the stream. The following code segment shows how to print 10 random numbers in a sorted order.

Random random = new Random();

random.ints().limit(10).sorted().forEach(System.out::println);

## Parallel Processing

parallelStream is the alternative of stream for parallel processing. Take a look at the following code segment that prints a count of empty strings using parallelStream.

List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

//get count of empty string

int count = strings.parallelStream().filter(string -> string.isEmpty()).count();

It is very easy to switch between sequential and parallel streams.

## Collectors

Collectors are used to combine the result of processing on the elements of a stream. Collectors can be used to return a list or a string.

List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.toList());

System.out.println("Filtered List: " + filtered);

String mergedString = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.joining(", "));

System.out.println("Merged String: " + mergedString);

## Statistics

With Java 8, statistics collectors are introduced to calculate all statistics when stream processing is being done.

List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

IntSummaryStatistics stats = integers.stream().mapToInt((x) -> x).summaryStatistics();

System.out.println("Highest number in List : " + stats.getMax());

System.out.println("Lowest number in List : " + stats.getMin());

System.out.println("Sum of all numbers : " + stats.getSum());

System.out.println("Average of all numbers : " + stats.getAverage());

## Stream Example

Create the following Java program using any editor of your choice in, say, C:\> JAVA.

### Java8Tester.java

import java.util.ArrayList;

import java.util.Arrays;

import java.util.IntSummaryStatistics;

import java.util.List;

import java.util.Random;

import java.util.stream.Collectors;

import java.util.Map;

public class Java8Tester {

public static void main(String args[]){

System.out.println("Using Java 7: ");

// Count empty strings

List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

System.out.println("List: " +strings);

long count = getCountEmptyStringUsingJava7(strings);

System.out.println("Empty Strings: " + count);

count = getCountLength3UsingJava7(strings);

System.out.println("Strings of length 3: " + count);

//Eliminate empty string

List<String> filtered = deleteEmptyStringsUsingJava7(strings);

System.out.println("Filtered List: " + filtered);

//Eliminate empty string and join using comma.

String mergedString = getMergedStringUsingJava7(strings,", ");

System.out.println("Merged String: " + mergedString);

List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

//get list of square of distinct numbers

List<Integer> squaresList = getSquares(numbers);

System.out.println("Squares List: " + squaresList);

List<Integer> integers = Arrays.asList(1,2,13,4,15,6,17,8,19);

System.out.println("List: " +integers);

System.out.println("Highest number in List : " + getMax(integers));

System.out.println("Lowest number in List : " + getMin(integers));

System.out.println("Sum of all numbers : " + getSum(integers));

System.out.println("Average of all numbers : " + getAverage(integers));

System.out.println("Random Numbers: ");

//print ten random numbers

Random random = new Random();

for(int i=0; i < 10; i++){

System.out.println(random.nextInt());

}

System.out.println("Using Java 8: ");

System.out.println("List: " +strings);

count = strings.stream().filter(string->string.isEmpty()).count();

System.out.println("Empty Strings: " + count);

count = strings.stream().filter(string -> string.length() == 3).count();

System.out.println("Strings of length 3: " + count);

filtered = strings.stream().filter(string ->!string.isEmpty()).collect(Collectors.toList());

System.out.println("Filtered List: " + filtered);

mergedString = strings.stream().filter(string ->!string.isEmpty()).collect(Collectors.joining(", "));

System.out.println("Merged String: " + mergedString);

squaresList = numbers.stream().map( i ->i\*i).distinct().collect(Collectors.toList());

System.out.println("Squares List: " + squaresList);

System.out.println("List: " +integers);

IntSummaryStatistics stats = integers.stream().mapToInt((x) ->x).summaryStatistics();

System.out.println("Highest number in List : " + stats.getMax());

System.out.println("Lowest number in List : " + stats.getMin());

System.out.println("Sum of all numbers : " + stats.getSum());

System.out.println("Average of all numbers : " + stats.getAverage());

System.out.println("Random Numbers: ");

random.ints().limit(10).sorted().forEach(System.out::println);

//parallel processing

count = strings.parallelStream().filter(string -> string.isEmpty()).count();

System.out.println("Empty Strings: " + count);

}

private static int getCountEmptyStringUsingJava7(List<String> strings){

int count = 0;

for(String string: strings){

if(string.isEmpty()){

count++;

}

}

return count;

}

private static int getCountLength3UsingJava7(List<String> strings){

int count = 0;

for(String string: strings){

if(string.length() == 3){

count++;

}

}

return count;

}

private static List<String> deleteEmptyStringsUsingJava7(List<String> strings){

List<String> filteredList = new ArrayList<String>();

for(String string: strings){

if(!string.isEmpty()){

filteredList.add(string);

}

}

return filteredList;

}

private static String getMergedStringUsingJava7(List<String> strings, String separator){

StringBuilder stringBuilder = new StringBuilder();

for(String string: strings){

if(!string.isEmpty()){

stringBuilder.append(string);

stringBuilder.append(separator);

}

}

String mergedString = stringBuilder.toString();

return mergedString.substring(0, mergedString.length()-2);

}

private static List<Integer> getSquares(List<Integer> numbers){

List<Integer> squaresList = new ArrayList<Integer>();

for(Integer number: numbers){

Integer square = new Integer(number.intValue() \* number.intValue());

if(!squaresList.contains(square)){

squaresList.add(square);

}

}

return squaresList;

}

private static int getMax(List<Integer> numbers){

int max = numbers.get(0);

for(int i=1;i < numbers.size();i++){

Integer number = numbers.get(i);

if(number.intValue() > max){

max = number.intValue();

}

}

return max;

}

private static int getMin(List<Integer> numbers){

int min = numbers.get(0);

for(int i=1;i < numbers.size();i++){

Integer number = numbers.get(i);

if(number.intValue() < min){

min = number.intValue();

}

}

return min;

}

private static int getSum(List numbers){

int sum = (int)(numbers.get(0));

for(int i=1;i < numbers.size();i++){

sum += (int)numbers.get(i);

}

return sum;

}

private static int getAverage(List<Integer> numbers){

return getSum(numbers) / numbers.size();

}

}

### Verify the Result

Compile the class using **javac** compiler as follows −

$javac Java8Tester.java

Now run the Java8Testeras follows −

$java Java8Tester

It should produce the following result −

Using Java 7:

List: [abc, , bc, efg, abcd, , jkl]

Empty Strings: 2

Strings of length 3: 3

Filtered List: [abc, bc, efg, abcd, jkl]

Merged String: abc, bc, efg, abcd, jkl

Squares List: [9, 4, 49, 25]

List: [1, 2, 13, 4, 15, 6, 17, 8, 19]

Highest number in List : 19

Lowest number in List : 1

Sum of all numbers : 85

Average of all numbers : 9

Random Numbers:

-1279735475

903418352

-1133928044

-1571118911

628530462

18407523

-881538250

-718932165

270259229

421676854

Using Java 8:

List: [abc, , bc, efg, abcd, , jkl]

Empty Strings: 2

Strings of length 3: 3

Filtered List: [abc, bc, efg, abcd, jkl]

Merged String: abc, bc, efg, abcd, jkl

Squares List: [9, 4, 49, 25]

List: [1, 2, 13, 4, 15, 6, 17, 8, 19]

Highest number in List : 19

Lowest number in List : 1

Sum of all numbers : 85

Average of all numbers : 9.444444444444445

Random Numbers:

-1009474951

-551240647

-2484714

181614550

933444268

1227850416

1579250773

1627454872

1683033687

1798939493

Empty Strings: 2

### 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);
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 it?s 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> 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. // Using Collectors's method to sum the prices.
23. **double** totalPrice3 = productsList.stream()
24. .collect(Collectors.summingDouble(product->product.price));
25. System.out.println(totalPrice3);
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()
23. .max((product1, product2)->
24. product1.price > product2.price ? 1: -1).get();
26. System.out.println(productA.price);
27. // min() method to get min Product price
28. Product productB = productsList.stream()
29. .max((product1, product2)->
30. product1.price < product2.price ? 1: -1).get();
31. System.out.println(productB.price);
33. }
34. }

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. }
14. **public** **class** JavaStreamExample {
15. **public** **static** **void** main(String[] args) {
16. List<Product> productsList = **new** ArrayList<Product>();
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));
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. }
14. **public** **class** JavaStreamExample {
15. **public** **static** **void** main(String[] args) {
16. List<Product> productsList = **new** ArrayList<Product>();
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));
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));
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;
4. **class** Product{
5. **int** id;
6. String name;
7. **float** price;
9. **public** Product(**int** id, String name, **float** price) {
10. **this**.id = id;
11. **this**.name = name;
12. **this**.price = price;
13. }
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. }
26. **public** **class** JavaStreamExample {
28. **public** **static** **void** main(String[] args) {
30. List<Product> productsList = **new** ArrayList<Product>();
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));
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]