Java 8 – Lambda Expressions, Streams & Collectors

Jorge Simão, Ph.D.

Java 8 - Functional Programming

Java 8 introduces several new features similar to what is found in *functional programming* languages. **Lambda expressions** are introduced as a way to simplify the implementation of **functional interfaces**. **Stream** objects are operated using a fluent-AI and a wide-variety of build-in functional interfaces.

Functional Interfaces

Functional interfaces are interfaces with a single method. (Not counting **static** and **default** methods, explained below).

» Example: Functional Interface – Event Handler

```
public interface EventHandler {
    void handle(Event event);
}
```

» Example: Functional Interface – Validator

```
public interface Validator<T> {
    boolean test(T obj);
}
```

» Example: Functional Interface – Generic Metric

```
public interface Metric<T> {
    double distance(T obj1, T obj2);
}
```

» Example: Functional Interface – Transform

```
public interface Transform<T,U> {
    U apply(T obj);
}
```

A functional interface can, optionally, be annotated with **@java.lang.FunctionalInterface** as a way to explicitly mark that the interface is functional. Adding further methods to such an interface will either produce compile-time or run-time errors.

» Example: Explicit Functional Interface

```
import java.lang.FunctionalInterface;

@FunctionalInterface
public interface <Validator<T> {
    boolean test(T obj);
}
```

Implementing Functional Interfaces

Anonymous inner-classes are the most convenient way to implement functional interfaces up-to Java 7. Named classes (inner or top-level), provide have the advantage of reuse.

» Example: Anonymous Class Implementation

```
filter(words, new Validator<String>() {
    public boolean test(String word) {
        return word.length>3;;
    }
});
```

» Example: Named Class Implementation

```
public class WordValidator implements
Validator<String>() {
    public boolean test(String word) {
        return word.length>3;;
    }
});
filter(words, new WordValidator());
```

While more less verbose than implementing as a named class (inner or top-level), it is arguably still a too verbose approaches – at least, compared with functional-like languages like JavaScript, Groovy, and other.

Lambda Expressions

Lambda expressions provide a light and convenient way to define functional interfaces, with syntax **(x0,x1,..) - > f(x0,x1,...).** Lambda expression can be used as argument of methods that expect an instance of a functional interface. Some examples as show below:

» Example: Lambda Expression with String

```
lines = transformAll(lines, s -> s.trim().toLowerCase());
```

» Example: Event Handler as Lambda Expression

```
widget.onClick( ev -> dialog.show());
```

» Example: Metrics for 2D Points

```
diameter(dataset, (p,q) -> (p.x-q.x)^2 + (p.y-q.y)^2);
diameter(dataset, (p,q) -> abs(p.x-q.x) + abs(p.y-q.y));
```

» Example: Lambda Expression for Validator

```
validate(about, s -> s.length>=12 and s.length<1024);
```

» Example: Lambda Expression with JDBC

```
String sql = "select id,username,email,password from Users";
List<Users> users = JdbcUtil.query(connection, sql,
    (ResultSet rs) -> new User(
        rs.getColumn("id"), rs.getColumn("username"),
        rs.getColumn("email"), rs.getColumn("username")));
```

Lambda expressions can be written with several styles or cases, summarizes below:

Syntax Style	Example
Explicit parameter type	(String s) → s.length>3
Implicit parameter type	(s) → s.length>3
No parameters	() -> Math.random()
Single Parameter	(s) -> s.length s -> s.length
Multiple Parameters	(s1,s2) -> s1.length-s2.length
Result Expression	(s) -> s.length>3

Statement Block	`(s) -> {`
	`  for (int i=0; i<s.length; i++) {`
	`    if (!isAlpha(s.charAt(i))) {`
	`      return false;`
	`  }`
	`  return true;`
	`}`

Default and Static Interface Methods

Default methods are methods whose implementation is define directly inside an interface definition. All classes implementing the interface automatically inherit the implementation of the method, although they can also overwrite it. Default methods have the modifier **default**.

Snippet below illustrates the use of a **default** method implementation. Notice that class **WordValidator** the implementation of **negate()** from the interface **Validator**.

» Example: Default Method in Functional Interface

```
public interface Validator<T> {
    boolean test(T obj);

    default Validator<T> negate() {
        return (obj) -> !test(obj);
    }
}
```

```
public class WordValidator implements Validator<String> {
    public boolean test(String s) {
        return s.length>3 && s.length<256;
    }
}
```

```
List<String> words = ...
WordValidator Validator = new WordValidator();

List<String> valid = filter(words, Validator);
List<String> invalid = filter(words, Validator.negate());
```

A functional interface can define **static** methods. A common use is for factory methods that create an instance of the interface out of other instance(s).

» Example: Static Method in Functional Interface

```
public interface Validator<T> {
    boolean test(T obj);
    ...
    static <T> ValidatorChecker<T>
    andN(Validator<? super T>... validators) {
        return (obj) -> {
            for (Validator<T> validator: validators) {
                if (!validator.test(obj)) {
                    return false;
                }
            }
            return true;
        }
    }
}
```

```
List<String> valid = filter(words, Validator.andN(
    new WordValidator(),
    Validator.negate(new InDicitionary()),
    s -> s.indexOf('-')<0);
```

Named Methods and Constructors

Named method are a convenient way to write a lambda expression whose behaviour is simply to call a method. The general syntax is **ClassName::methodName** or **object::methodName**. The following cases are supported:

Static method

- Instance method, of a specific object
- Instance method, of an arbitrary object
- Constructor

» Example: Named Static Method

```
filter(emails, EmailUtil::isValid);
```

» Example: Named Instance Method

```
filter(words, new WordValidator(3,7,"\d{16}")::isValid);
```

» Example: Named Instance Method – Arbitrary Object

```
max(portfolios, Portfolio::compareByValue);
```

» Example: Named Constructor

```
filter(stocks, s -> s.getDelta()>0, TreeSet::new);
```

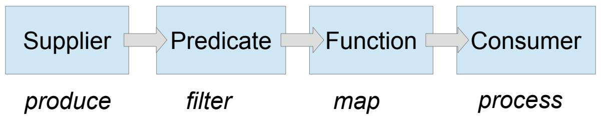
Built-in Functional Interfaces

Java 8 provides several useful Functional interfaces out-of-the-box, that can be used with **Stream**s and in other contexts.

Table below describes some of the most useful built-in functional interfaces.

Interface	Method	Description
Predicate<T>	**test(T)**	Condition on Object
Supplier<T>	**T get()**	Object source
Consumer<T>	**accept(T)**	Object sink
Function<T,U>	**U map(T)**	Object mapping

These built-in functional interfaces a often combined in processing pipelines:



Snippet below illustrates the use of several built-in functional interfaces.

» Example: Processing Data with Built-in Interfaces

```
<T> void process(Supplier<T> supplier,
          Predicate<T> predicate, Function<T,U> f,
          Consumer<T> consumer) {
     T obj = null;
     while ((obj=supplier.get()!=null)) {
          if (predicate.test(obj)) {
               consumer.accept(f.apply(obj));
          }
     }
}
final Iterator<String> it = dataset.iterator();
process(p -> it.hasNext() ? it.next() : null,
          p -> !isOutlier(p, dataset),
          p-> normalize(p)
          p -> out.println(p));
```

Table below describes further built-in functional interfaces to work with object pairs:

Interface	Method	Description
BiPredicate<T,U>	**test(T,U)**	Condition on pair of Objects
BiConsumer<T>	**accept(T,U)**	Object pair sink
Function<T,U,R>	**R map(T,R)**	Object pair mapping

Streams

A Stream is an abstraction for a consumable source of objects, with a fluent API to process and consume objects.

» Example: Stream Processing

```
words.stream()
     .filter(s -> s.length>3)
     .map(s → s.toLowerCase())
     .sort()
     .distinct()
     .sort((s1,s2) -> s1.length<s2.length)
     .forEach( s->System.out.println(s));
```

Input

```
"Now", "Hello", "the", "World", "Goodbye", "world"
```

Output

```
hello
world
goodbye
```

Table below summaries the API of **Stream**s:

Method & Example	Description
map(Function<?,?>)	Map elements
ds.map(s -> s.length)	
filter(Predicate<?>)	Filter elements
ds.filter(s -> s.length>3)	
forEach(Consumer<?>)	Iterate elements

forEachOrdered(Consumer<? >)	Sort and Iterate
limit(long maxSize)	Discard after **maxSize** elements
skip(long n)	Skip first **n** elements

Snippet below show some further examples of using the **Stream** API:

» Example: Stream Elements

```
words.words.stream()
     .map( TextUtil::transform)
     .filter( validator::isValid(s))
     .skip(2*pageSize)
     .limit(pageSize)
     .forEach( s->words.add(s));
```

Parallel Streams and Spliterators

Streams operations can be performed in parallel by splitting a stream in successive halves.

The interface Spliterator defines the API that Stream operations can use to split a Stream.

Reduce and Collect Operations

The **Stream.reduce()** operation allow all elements of a **Stream** to be processed and combined in a single result value. A accumulator function is used to the reduction of each element with the current intermediary result.

The **reduce()** method comes is several variations. In the simplest form, the accumulator is a **BinaryOperator** and the identity parameter is specified for the initial value to be accumulated. If the **Stream** as no elements the identity parameter is returned as a result.

» Example: Reduce Operation – With Identity

```
Stream<Integer> series =
    Stream.iterate(1, n -> n+1).limit(1000);
series.reduce(0, (n,m)->n+m);
List<String> words = ...
words.stream()
    .map(s -> s.length)
    .reduce(0, Integer::sum);
```

When the **identity** parameter is absent, another **reduce()** method returns an **Optional<T>**. This is a result descriptor which can be used to check if a result exist (i.e. the **Stream** is not empty), and to get the result. If the **Stream** as a single element the result is the that element.

» Example: Reduce Operation – Returning Optional

```
Optional<Integer> result = series.reduce(Integer::sum);
int sum = result.isPresent() ? result:get() : 0;
```

A third variant of reduce() can be used to combine a map and and a reduce operation in a single step.

Table below detail the signature of the three reduce methods.

Reduce Methods (Oveloaded)
T reduce(T identity, BinaryOperator<T> accumulator);
Optional<T> reduce(BinaryOperator<T> accumulator);
<U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner);

Notice that in the third reduce() variant the result value is different from the type of the Stream elements. A **BiFunction** is used as accumulator.

When Streams are parallel each split of the Stream is reduced in separately.

» Example: Reduce Operation – With Combiner

The **Stream.combine()** operation allows for the optimization of **reduce()** operations, by applying the accumulator to a mutable result container, such as a **Collection** or **StringBuilder**.

Collect Methods (Overloaded)
<R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)
<R,A> R collect(Collector<? super T,A,R> collector)

Snippets below show some further examples of using the **Stream** API:

» Example: Collect Operation

```
Stream<Integer> series = Stream.iterate(1, n ->
n+1).limit(1000);
series.reduce(0, (n,m)->n+m);
```

Collectors

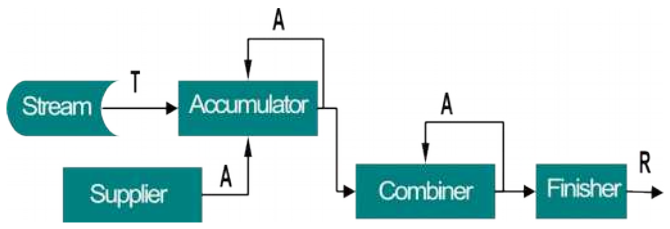
Collectors are descriptors that define all the strategies required to perform a **collect()** operation.

» Collector Interface

```
public interface Collector<T,A,R> {
    Supplier<A> supplier();
    BiConsumer<A,T> accumulator();
    BinaryOperator<A> combiner();
    Function<A,R> finisher();
    Set<Collector.Characteristics> characteristics();
}
```

» Collector.Characteristics

```
enum Collector.Characteristics {
    CONCURRENT,
    IDENTITY_FINISH,
    UNORDERED
}
```



Snippet illustrates the use of method **Collector.of()** to create a collector out of the individual strategies:

» Example: Creating a Collector

```
Collector<String,ArrayList> appender =
Collector.of(ArrayList::new, ArrayList::add, ArrayList::addAll);
Stream<String> words =/ …
List<String> wordList = words.collect(appender);
```

Built-in Collectors

Java provides several built-in collectors created by factory methods in utility class **Collectors**. Snippet below show how a built-in collectors can be used.

» Example: Summing with Built-in Collector

```
Collector<Employee, ?, Integer> sumLength
    = Collectors.summingInt(String::length);
int charCount = words.stream().collect(sumLength);
```

The built-in collector returned by method **Collectors.groupingBy()** groups all elements that have the same criteria value in a **Map**, where the key is the criteria value and the entry value is the **List** of all elements in the **Stream** matching the criteria – the group.

» Example: Grouping with Built-in Collector

```
words.stream().collect(
    Collectors.groupingBy(s->s.length);
```

Input "Now", "Hello", "the", "World", "Goodbye", "world"

Output {3={Now,the}, 5={Hello,World}, 7={Goodbye}}

A second collector can be provided as argument to **groupingBy()** to combine all element in each group.

» Example: Grouping followed by Aggregation

```
words.stream().collect(
    Collectors.groupingBy(User::getCountry,
        Collectors.counting());
```

Input User(username="Joe", country="UK", ...), ...

Output {UK=98, US=130, DE=76, PT=12, ES=22}

Method **Collectors.partitioning()** returns a **Map** with **boolean** entries, splitting the **Stream** elements in two sub-sets - the elements that match and don't match a boolean criteria:

» Example: Partitioning with Predicate

```
words.stream().collect(
    Collectors.groupingBy(s->s.length>3);
```

Input "Now", "Hello", "the", "World", "Goodbye", "world", ...

Input

Output {false={Now,the}, true={Hello,World,Goodbye}}

The **partitioning()** method can also be provided a second collector as argument to combine all elements in each partitioning.

» Example: Partitioning Followed by Aggregation

```
words.stream().collect(
    Collectors.groupingBy(s->s.length>3,
    Collectors.counting());
```

Input "Now", "Hello", "the", "World", "Goodbye", "world", ..., ...

Output {false=2, true=3}

Table below summaries the factory methods in utility class **Collectors**:

Method Collectors.*	Description
counting()	Count elements

joining()
joining(String separator)
	Joining Strings

toSet()	Build Set
toList()	Build List
toCollection()	Build Collection
toMap()	Build Map

summingInt()
summingLong()
summingDouble()
	Sum Numbers

averagingInt()
averagingLong()
averegingDouble()
	Averaging Numbers

summarizingInt()
summarizingLong()
summarizingDouble()
	Summary statitics

groupingBy()	Group elements
partitioningBy()	Partition elements

Resources

- JavaTM Tutorial on Lamba Expressions by OracleTM

EInnovator
Software Engineering School

About the Author

Jorge Simão is a software engineer and IT Trainer, with two decades long experience on education delivery both in academia and industry. Expert in a wide range of computing topics, he his an author, trainer, and director (Education & Consulting) at **EInnovator**. He holds a B.Sc., M.Sc., and Ph.D. in Computer Science and Engineering.

Java Programming Training

Master your Java with a trainer-lead 5 day course on **Java Programming**. Covers everything needed to become a professional Java developer, including: imperative and object-oriented programming with Java, basic and advanced APIs, Java 8 lambda expressions and streams, and enterprise Java topics, such as: Dependency-Injection+AOP, JPA, and Web development. Custom training with selected topics and duration also available on demand. Register for a public or online class or book an on-site class: **www.einnovator.org/course/java**

EInnovator – Software Engineering School

EInnovator.org offers the best Software Engineering resources and education, in partnership with the most innovative companies in the IT-world. Focusing on both foundational and cutting-edge technologies and topics, at EInnovator software engineers and data-scientists can get all the skills needed to become top-of-the-line on state-of-the-art IT professionals.

Contacts

Training – Bookings & Inquiries
training@einnovator.org

Consultancy – Partnerships & Inquiries
consulting@einnovator.org

General Info
info@einnovator.org

EInnovator