System Design – Reactive Programming [06]

Hochschule für Technik Stuttgart

Marcus Deininger

WS 2021/22

Overview

- Lambdas / Closures

  → Function Pointers
- Streams

  → Functional Programming
- Publishers / Subscribers

  → Reactive Programming

Lambdas / Closure – Basic Idea

- Functions can be considered as Data too

→ Functions can be

  - assigned to variables
  - passed as parameters (→ “Callbacks”)
  - evaluated anytime later

Finding `min` and `max`

```
public class MinMax1 {

    public static int min(int[] a) {
        int min = a[0];
        for(int i = 0; i < a.length; i++)
            if(a[i] < min)
                min = a[i];
        return min;
    }

    public static int max(int[] a) {
        int max = a[0];
        for(int i = 0; i < a.length; i++)
            if(a[i] > max)
                max = a[i];
        return max;
    }

    public static void main(String[] args) {
        int[] a = {2, 3, 1, 5, 0, 8, 4};

        System.out.println(min(a));
        System.out.println(max(a));
    }
}
```

This is basically the same logic/algorithm.

This is the only difference.

Using a Strategy …

```
public interface BiIntPredicate {

    boolean test(int n, int m);
}

public class MinMax2 {

        public static int select(int[] a, BiIntPredicate predicate) {
        int selection = a[0];
        for(int i = 0; i < a.length; i++)
            if(predicate.test(a[i], selection))
                selection = a[i];
        return selection;
    }

…

}
```

This is the strategy

Passing of the strategy.

Usage of the strategy.

Supplying a Strategy …

```
public class MinMax2 {

    static class Min implements BiIntPredicate{

        @Override
        public boolean test(int n, int m) {
            return n < m;
        }

    }

    public static void main(String[] args) {
        int[] a = {2, 3, 1, 5, 0, 8, 4};

        System.out.println(select(a, new Min()));

        System.out.println(select(a, new BiIntPredicate() {

            @Override
            public boolean test(int n, int m) {
                return n > m;
            }
        }));
```

Option 1: Creating an implementation

Option 1: Supplying an instance

Option 2: Supplying an instance of an anonymous inner class.

This is a lot of boiler-plate code for mainly supplying `n<m` or `n>m`

Enter Lambdas …

```
public class MinMax3 {

    public static int select(int[] a, BiIntPredicate predicate) {
        …
    }

    public static void main(String[] args) {
        int[] a = {2, 3, 1, 5, 0, 8, 4};


        System.out.println(select(a, new BiIntPredicate() {

            @Override
            public boolean test(int n, int m) {
                return n > m;
            }
        }));


        System.out.println(select(a, (n, m) -> n > m));

    }
}
```

again (to illustrate the contrast)
Option 2: Supplying an instance of an anonymous inner class

Option 3: A lambda-expression – effectively the same as Option 2

Enter Lambdas – Type Deduction …

```
public class MinMax3 {

    public static int select(int[] a, BiIntPredicate predicate) {
        …
    }

    public static void main(String[] args) {
        int[] a = {2, 3, 1, 5, 0, 8, 4};


        System.out.println(select(a, new BiIntPredicate() {

            @Override
            public boolean test(int n, int m) {
                return n > m;
            }
        }));


        System.out.println(select(a, (n, m) -> n > m));

    }
}
```

Prerequisite: The Interface is a SAM-Type (Single Abstract Method)

only one abstract method → method name is **`test`**

method has 2 parameters (of **`int`**)

method has **`return`** type of **`boolean`**

`(n, m)` is a shortcut for **`boolean`** `test(int n, int m)`

`n > m` is a shortcut for **`return`** `n > m;`

Lambdas / Closures

- Lambdas are effectively
  - shortcuts for interface-implementations
  - of interfaces with a single abstract method (“SAM”)
  - without states (i.e. private fields)
  - without checked exceptions
  - aka as “Closures”

→ Lambdas allow a less wordy implementation

→ Lambdas allow passing code to methods

- Closures are a more efficient than creating anonymous classes / instances
- Lambdas are typically used for ActionListeners, Runnables, Callables, Comparators and Streams (see later)

Lambdas: Rules / Restrictions – 1

Lambda (focus Parameters)	Comment
`(int x) -> x+1`	Single parameter with type and brackets
`int x -> x+1`	Wrong: missing brackets
`(x) -> x+1`	Single parameter with brackets.
`x -> x+1`	Single parameter without brackets.
`(int x, int y) -> x + y`	Two parameters with types and brackets
`int x,int y -> x+y`	Wrong: missing brackets
`(x,y) -> x+y`	Two parameters without types
`x,y -> x+y`	Wrong: two parameters need brackets
`(x, int y) -> x+y`	Wrong: mixture of type / no-type
`() -> 42`	Empty parameter list

see: http://www.angelikalanger.com/Articles/EffectiveJava/71.Java8.Lambdas/71.Java8.Lambdas.html

Lambdas: Rules / Restrictions – 2

Lambda (focus Body)	Comment
`() -> System.out.println(42)`	Body with single expression
`(String[] args)` `  -> (args != null) ? args.length : 0`	Ternary operator is still a single expression

`(String[] args) -> { if(args != null)`
`  return  args.length;`
`  else`
`  return  0;`
`  }`
	Body with if-statement (which is not an expression) → needs curly brackets

`(int x) ->  x+1`	Again, body with single expression
`(int x) -> return x+1`	Wrong: **`return`** is a statement
`(int x) ->  {  return x+1 ; }`	Body with return-statement in curly brackets and semicolon

see: http://www.angelikalanger.com/Articles/EffectiveJava/71.Java8.Lambdas/71.Java8.Lambdas.html

Lambdas: Rules / Restrictions – 3

Lambda Usage	Comment

`int k = 5;`
`m(() -> System.out.println(k));`
`k = 10;`
	Wrong: `k` is *not* effective final

`int k = 5;`
`m(() -> System.out.println(k));`
`// no further assignment to k`
	`k` is effective final

`final int k = 5;` `m(() -> System.out.println(k));`	`k` is final
`for(int i = 0; i < 10; i++)` `  m(() -> System.out.println(i));`	Wrong: `i` is (of course) *not* effective final

`for(int i = 0; i < 10; i++) {`
`  int n = i;`
`  m(() -> System.out.println(n));`
`}`
	`i` is *not* effective final, but `n` is

Lambdas – `java.util.function`

Interface	Method	Sample Expression	Remark
`Consumer<T>`	`accept(T x)`	`x -> m()`	m is a **`void`**-method
`Supplier<T>`	`get():T`	`() -> 5`	no parameters
`Predicate<T>`	`test(T x):boolean`	`x -> x < 5`	returns a boolean
`Function<T,R>`	`apply(T x):R`	`x -> x.toString()`	one parameter, returns a value
`BiFunction<T,U,R>`	`apply(T x, U y):R`	`(x,y) -> x+y*y`	two parameters, returns a value
`UnaryOperator<T>`	`apply(T x):T`	`x -> x*x-1`	one parameter, returns a value of the same type
`BinaryOperator<T>`	`apply(T x, T y):T`	`(x,y) -> x-y`	two parameters, returns a value of the same type

- Some specific interfaces for `int`, `double` and `long`, e.g. `IntConsumer` with method `accept(int x)`
- If you need more (e.g. `IntBiPredicate`), you may define it for yourself

Lambdas – Method References

- Sometimes an already existing method fits your needs, e.g.

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

- Instead, you may use a method reference

```
Consumer<String> c = System.out::println;
```

The implementer of a method – either an object or a class.

The method – if there are several, the correct one will be deducted

Functional Programming – Basic Idea

- The program is considered as a stream of data
- The data is processed by functions
- Functions have an input and an output but no side-effects

Streams – 1

- A Stream is a sequence of elements supporting sequential and parallel aggregate operations – it is not a collection
- The type is `Stream<T>` in the package `java.util`
- Streams are typically used like this

```
create().op1(f1).op2(f2). … .lastOp(fn);
```

A method which creates a new stream.

Intermediate operations converting a `Stream<T>` to `Stream<U>` with f being a `Function<T, U>`

A terminal operation which does not return a Stream with f being a Function or Consumer

Streams – 2

- Lazy evaluation: the terminal operation triggers the stream
- The stream hands over one element after the other to its operations.
- Streams cannot preview or undo or restart.
- Streams can be considered as Pipes, intermediate operations are Filters

Streams – Example 1

```
String[] strings =
  {"This", "is", "a", "sequence",
    "of", "strings", "."};

Arrays.stream(strings)
                .map(s -> s.toUpperCase())
                .forEach(s -> System.out.println(s));
```

Creates a `Stream<String>` from the String array.

Intermediate operation `map` converting a `Stream<String>` to `Stream<String>` by applying its function on each element.

Terminal operation `forEach` applying a `Consumer` on each element

Streams – Example 2

```
String[] strings =
  {"This", "is", "a", "sequence",
    "of", "strings", "."};

int sum = Arrays.stream(strings)
              .map(s -> s.toUpperCase())
              .filter(s -> s.matches("\\w*"))
              .map(s -> s.length())
              .peek(l -> System.out.println(l))
              .reduce(0, (s, l) -> s + l);
```

Creates a `Stream<String>` from the String array.

Intermediate operation `filter` filtering all elements fulfilling Predicate's test.

Intermediate operation `peek` applying its Consumer otherwise passing its elements unaltered.

Terminal operation `reduce` applying a `BiFunction` on each element, starting with a seed and returning the final result.

Stream-Creation

Operation	Result
`Stream.of(T…values)`	Stream of the given values
`Arrays.stream(T[] values)`	Stream of the array values
`List<String> list = …` `list.stream()`	Stream of the list values
`Stream.generate(Supplier<T> s)`	Endless stream, values by the supplier
`Stream.iterate` `(T seed, UnaryOperator<T> op)`	Endless stream, values from the previous value by the operator – seed is the first value.

- Endless streams may be limited with the `limit(long)`-filter
- For primitive values there are (more efficient) `IntStream`, `DoubleStream`, `LongStream`

Intermediate Operations

Operation	Result
`.map(Function<T, U> f)`	Transforms each value → returns a new stream of the target type U
`.filter(Predicate<T> p)`	Lets only pass values fulfilling the predicate
`.limit(long n)`	Limits the number of elements to n (useful for endless streams)
`.peek(Consumer<T> a)`	Performs the consumer-operation on each element – otherwise the elements are passed (useful for debugging)
`.parallel()`	runs the stream in parallel
`.sequential()`	runs the stream sequentially (default)

Intermediate Operations – `flatMap`

- `flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)`
- Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
- Example

```
Stream<String> lines = Stream.of("This is a sequence
of strings.", "This is the next line.");

lines
 .flatMap(line -> Stream.of(line.split(" +")))
 .forEach(System.out::println);
```

Terminal Operations

Operation	Result
`T reduce(T seed, BinaryOperator<T> f)`	Reduces (accumulates) all values to one value, using seed as a starting point. E.g. `reduce(0, (n, value) -> n + value)`
`void forEach( Consumer<T> action))`	Performs the action on each value.
`long count()`	Counts the number of stream elements
`<R,A> R collect(Collector<T,A,R> c)`	Creates a collection according to the Collector which contains all stream elements.

- Collectors are usually imported statically, i.e. `import static java.util.stream.Collectors.*`
- predefined Collectors are: `toList()`,`toSet()`, `toMap(Function, Function)`, `groupingBy(Function, Collector)`
- the last producing a Map with a collection as value (again by a collector)

Reactive Programming – Basic Idea

- The program is considered as a (live-)stream of messages
- The messages are emitted by a publisher
- A client may subscribe on the publisher
- Such a subscriber receives ("observes") the messages asynchronously
- Publisher and subscribers may be chained with intermediate processors (which are publishers and subscribers)
- The stream is back-pressured
  - A subscriber gets only as much data as he can handle
  - Undelivered data may be buffered or dropped by the publisher

Components

- ```
  public interface Publisher<T> {
    public void subscribe(Subscriber<? super T> s);
  }
  ```
- ```
  public interface Subscriber<T> {
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
  }
  ```
- ```
  public interface Subscription {
    public void request(long n);
    public void cancel();
  }
  ```
- ```
  public interface Processor<T, R>
    extends Subscriber<T>, Publisher<R> { }
  ```

Components – Behavior

- A Subscriber may `subscribe` to a Publisher
- The Subscriber
  - receives the Subscription through `onSubscription`
  - signals a `request` through his subscription
  - may `cancel` his subscription
  - receives `onNext` with next item, if requested
  - receives `onError`, if an error occurred – no further messages are available
  - receives `onComplete`, when the stream is finished
- Java 9 already provides a predefined `SubmissionPublisher` (with its Subscription)
- You have to come up with your own Subscriber

A Sample Subscriber

Callback after subscribe: save the subscription and request the first item

Callback after request: get a new item and request the next item

Callback after an error: no more requests possible.

Callback after regular termination: no more requests possible.

```
public class SampleSubscriber<T> implements Subscriber<T> {

    protected Subscription subscription;
    private String name;

    public SampleSubscriber() {
        this.name = "Subscriber";
    }

    public SampleSubscriber(String name) {
        this.name = "Subscriber " + name;
    }

    public void onSubscribe(Subscription subscription) {
        this.subscription = subscription;
        System.out.println(name + ": subscribed");
        subscription.request(1);
    }

    public void onNext(T item) {
        System.out.println(name + ": got " + item);
        subscription.request(1);
    }

    public void onError(Throwable t) {
        System.out.println(name + ": error " + t.toString());
    }

    public void onComplete() {
        System.out.println(name + ": done");
    }

    public String toString() {
        return name;
    }
}
```

based on: https://github.com/politrons/reactive/blob/master/src/test/java/java9/FlowFeatures.java

A Sample Application

Create Publisher and Subscriber

Register the Subscriber

Items to be published

Wait for everything to be done and close.

```
public class SampleFlowApp1 {

    public static void main(String[] args) throws InterruptedException {

        SubmissionPublisher<String>publisher = new SubmissionPublisher<>();
        SampleSubscriber<String>   subscriber = new SampleSubscriber<>();

        publisher.subscribe(subscriber);
        Thread.sleep(500);

        List<String> items = List.of("A", "B", "C", "D");
        System.out.println("Publishing Items...");
        items.stream().forEach(publisher::submit);
        System.out.println("Publishing finished");

        Thread.sleep(500);
        publisher.close();
        System.out.println("Application finished");
    }
```

The Publisher emits the item.

```
Subscriber: subscribed
Publishing Items...
Publishing finished
Subscriber: got A
Subscriber: got B
Subscriber: got C
Subscriber: got D
Application finished
Subscriber: done
```

Output

A Lazy Subscriber

Callback after request: get a new item and sleep for some time and request the next item

```
public class LazySubscriber<T> extends SampleSubscriber<T> {

    private int sleepIime = 0;

    public LazySubscriber(int sleepIime) {
        this.sleepIime = sleepIime;
    }

    public LazySubscriber(String name, int sleepIime) {
        super(name);
        this.sleepIime = sleepIime;
    }

    @Override
    public void onNext(T item) {
        System.out.println(this + ": got " + item);
        try {
            Thread.sleep(sleepIime);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        subscription.request(1);
    }
}
```


based on: https://github.com/politrons/reactive/blob/master/src/test/java/java9/FlowFeatures.java

A Sample Application 2 – Initialization

Count the number of dropped items

Sleep for some time and return retry – to be used below

Parallel management of the subscribers

Create several subscribers; each becoming more slower

```
public class SampleFlowApp6 {

    private static int drops;

    public static boolean sleepAndRetry(int ms, boolean retry) {
        try {
            Thread.sleep(ms);
        } catch (InterruptedException e) { e.printStackTrace(); }
        return retry;
    }

    public static void main(String[] args) throws Inte

        int maxBufferCapacity = 2;
        SubmissionPublisher<String> publisher =
            new SubmissionPublisher<>(ForkJoinPool.commonPool(),
                                      maxBufferCapacity);

        for (int i = 0; i < 5; i++)
            publisher.subscribe(new LazySubscriber<>((i + 1) + "",
                                                     i * 500));
        Thread.sleep(500);

        …
    }
```

The capacity of the publisher's buffer

A Sample Application 2 – Offering

Items to be published

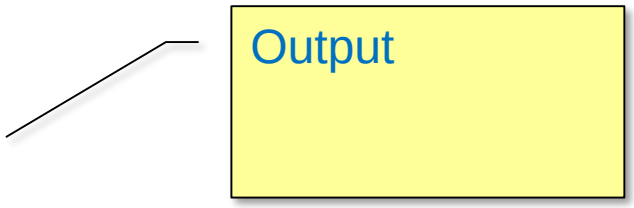
Offer item to the subscriber – do some action if dropped and possibly retry.

Negative Result gives the number of dropped items.

```
public class SampleFlowApp6 {
    …
    public static void main(String[] args) throws InterruptedException {
        …
        System.out.println("Publishing Items...");
        drops = 0;
        Stream.iterate(1, n -> n + 1)
              .limit(3)
              .map(n -> "Item " + n)
              .forEach(item -> {

                  int result = publisher.offer(item,
                      (subscriber, value) ->  sleepAndRetry(500, false));

                  // Negative result gives the number of dropped items
                  if (result < 0) {
                      System.out.println("dropped: " + -result);
                      drops = drops - result;
                  }

              });
        System.out.println("Publishing finished; dropped: " + drops);

        Thread.sleep(500);
        publisher.close();
        System.out.println("Application finished");
    }
}
```

A Sample Application 2 – Output

```
Subscriber 3: subscribed
Subscriber 2: subscribed
Subscriber 1: subscribed
Subscriber 4: subscribed
Subscriber 5: subscribed
Publishing Items...
Subscriber 1: got Item 1
Subscriber 2: got Item 1
Subscriber 1: got Item 2
Subscriber 3: got Item 1
Subscriber 4: got Item 1
Subscriber 2: got Item 2
dropped: 2
Subscriber 3: got Item 2
Subscriber 2: got Item 3
Publishing finished; dropped: 2
Subscriber 4: got Item 2
Application finished
Subscriber 2: done
Subscriber 5: got Item 1
```



A Processor

A Processor is a Publisher and a Subscriber.

The Processor shall transform its values with this function.

Callback after request: get a new item, submit it to the next Subscriber and request the next item from the Publisher

```
public class TransformerProcessor<T, R> extends
        SubmissionPublisher<R> implements Processor<T, R> {

    private Function<? super T, ? extends R> transform;
    private Subscription subscription;

    public TransformerProcessor(
        Function<? super T, ? extends R> transform) {
        super();
        this.transform = transform;
    }

    public void onSubscribe(Subscription subscription) {
        this.subscription = subscription;
        subscription.request(1);
    }

    public void onNext(T item) {
        submit((R) transform.apply(item));
        subscription.request(1);
    }

    public void onError(Throwable t) {
        System.out.println("Transformer: error "
                                        + t.toString());
    }

    public void onComplete() {
        close();
    }
}
```

based on: https://github.com/politrons/reactive/blob/master/src/test/java/java9/FlowFeatures.java

A Sample Application 3

Create Publisher, Processor and Subscriber

Register the Processor to the Publisher

Register the Subscriber to the Processor

Wait for everything to be done and close.

```
public class SampleFlowApp8 {

    public static void main(String[] args) throws InterruptedException {

        SubmissionPublisher<String> publisher = new SubmissionPublisher<>();
        TransformerProcessor<String, Integer>
                processor = new TransformerProcessor<>(Integer::parseInt);
        SampleSubscriber<Integer> subscriber = new SampleSubscriber<>();

        publisher.subscribe(processor);
        processor.subscribe(subscriber);
        Thread.sleep(500);

        List<String> items = List.of("1", "2", "three", "4"
        System.out.println("Publishing Items...");
        items.stream().forEach(publisher::submit);
        System.out.println("Publishing finished");

        Thread.sleep(500);
        publisher.close();
        System.out.println("Application finished");
    }
```

Items to be published – "three" should force an exception

The Publisher emits the item.

Output

```
Subscriber: subscribed
Publishing Items...
Publishing finished
Subscriber: got 1
Subscriber: got 2
Transformer: error java.lang.NumberFormatException: For input string: "three"
Application finished
```

Hochschule für Technik Stuttgart

Creating Subscribers and Publishers

Creating Subscribers

- Rather simple, see samples before

Creating Publishers

- From the scratch, rather complicated (see: https://dzone.com/articles/mastering-own-reactive-streams-implementation-part)
- Java provides one predefined SubmissionPublisher → can be used to realize your own easily

A Sample Publisher

The StringPublisher should emit an endless stream of Strings

```
public class StringPublisher implements Publisher<String> {

    private SubmissionPublisher<String> publisher;

    public StringPublisher() {
        System.out.println("Created ...");
        this.publisher = new SubmissionPublisher<>();
        new Thread(() -> emit()).start();
    }

    private void emit() {
        System.out.println("Publishing Items...");
        int n = 1; char c = 'A';
        while(true) {
            String str = c + "_" + n;
            n++; c++;
            if(c == 'Z') c = 'A';

            publisher.submit(str);

            try { Thread.sleep(100); } catch (InterruptedException e) { }
        }
    }

    @Override
    public void subscribe(Subscriber<? super String> subscriber) {
        publisher.subscribe(subscriber);
    }
}
```

Use the pre-defined Submission Publisher

Assign the publisher

Start a thread, which creates and emits the strings

This is just a sample of an endless loop creating some strings

Delegate the emittance

Delegate the subscription

A Sample Application 4

Create Publisher and Subscriber

Register the Subscriber

```
public class SampleFlowAppWithStringPublisher {

    public static void main(String[] args) throws InterruptedException {

        StringPublisher   publisher = new StringPublisher();
        SampleSubscriber<String> subscriber = new SampleSubscriber<>();

        publisher.subscribe(subscriber);
        Thread.sleep(1000);
        System.out.println("Application finished");
    }
}
```

```
Created ...
Publishing Items...
Subscriber: subscribed
Subscriber: got B_2
Subscriber: got C_3
Subscriber: got D_4
Subscriber: got E_5
Subscriber: got F_6
Subscriber: got G_7
Subscriber: got H_8
Subscriber: got I_9
Subscriber: got J_10
Application finished
Subscriber: got K_11
Subscriber: got L_12
...
```

Output

The app is finished but the publisher continues

References

- Baeldung: Java 9 Reactive Streams, 2018, https://www.baeldung.com/java-9-reactive-streams
- Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson: The Reactive Manifesto, 2014, https://www.reactivemanifesto.org/
- Oleh Dokuka: Mastering Reactive Streams (Part 1): Publisher, 2017, https://dzone.com/articles/mastering-own-reactive-streams-implementation-part
- Moisés Macero: The Reactive Magazine Developer, 2018, https://thepracticaldeveloper.com/2018/01/31/reactive-programming-java-9-flow/
- reactive-streams.org: Reactive Streams, 2017, https://github.com/reactive-streams/reactive-streams-jvm.
- Venkat Subramaniam: Reactive Programming in Java, 2017, video, https://www.youtube.com/watch?v=f3acAsSZPhU