Java 8 Workshop

Webstep 2014

Bjørn Hamre (@javaguruen)
Kristian Berg (@rasmantutu)

Innhold

Del 1:

- @FunctionalInterface/Single Abstract Method
- Lambda

Del 2:

- Optional
- Stream (filter, map, reduce, Collectors)

Del 3:

- Parallel stream
- More Collectors
- Generators

This workshop *is not*

A course in functional programming

Part 1

Lambda expression

A parameter list

An arrow

A block of code

```
(final String name) -> {
    return “Hello, “ + name;
}
```

Lambda expression

Types are inferred

No parenteses for one parameter

```
name -> {
    return “Hello, “ + name;
}
```

Lambda expression

No braces for one-liners

```
name ->
    return “Hello, “ + name;
```

Lambda expression

Return types inferred

No “return” for one-liners

name -> “Hello, “ + name;

Single Abstract Method (SAM)

Applies to interfaces and abstract classes

Known examples:

Comparator, Runnable

New: java.util.function

Consumer, Predicate, Function

@FunctionalInterface

Lambda expression

Can be assigned to functional interfaces:

```
Predicate<Integer> isEven = (Integer number) -> {
  return number % 2 == 0;
};

Predicate<Integer> isEvenShorter = (n) -> n % 2 == 0;
```

External -> internal iteration

External iteration

- Outside collections

Imperative

- How and what

```
final List<String> names = Arrays.asList("Bjørn", "Kristian", "Erlend");

for( String name : names ){
  System.out.println( name );
}
```

External -> internal iteration

Internal iteration

- Inside collections

Declarative

- Only what

```
final List<String> names = Arrays.asList("Bjørn", "Kristian", "Erlend");

names.forEach( name -> System.out.println(name) );

names.forEach( System.out::println );
```

Internal iteration

New method on Iterable

```
default void forEach(Consumer<? super T> action) {
    Objects.requireNonNull(action);
    for (T t : this) {
        action.accept(t);
    }
}
```

Internal iteration

java.util.function.Consumer:

```
public interface Consumer<T> {

  void accept(T t);

  default Consumer<T> andThen(Consumer<? super T> after) {
    Objects.requireNonNull(after);
    return (T t) -> { accept(t); after.accept(t); };
  }
}
```

Lambdas in action

Internal vs external iteration

Which one would you parallelize?

```
final List<String> names = Arrays.asList("Bjørn", "Kristian", "Erlend");

for( String name : names ){
  System.out.println( name );
}

names.forEach( name -> System.out.println(name) );

names.forEach( System.out::println );
```

Method reference

Class::staticMethod

objectReference::instanceMethod

Class::new

```
public class ForEachGreeter {

  public static void main(String[] args) {
    final List<String> names = Arrays.asList("Bjørn", "Kristian", "Erlend");
    names.forEach( ForEachGreeter::printGreeting );
  }

  public static void printGreeting(String name){
    System.out.println("Hello, " + name);
  }
}
```

Sorting with lamda

Sortering < Java 8

```
List<Person> persons = Arrays.asList(
    new Person("Bjørn", 41), new Person("Kristian", 47), new Person("Erlend", 110));

persons.sort( new Comparator<Person>() {
  @Override
  public int compare(Person o1, Person o2) {
    return o1.name.compareTo(o2.name);
  }
});
```

Sortering Java 8 style

```
List<Person> persons = Arrays.asList(
    new Person("Bjørn", 41), new Person("Kristian", 47), new Person("Erlend", 110));

persons.sort( (Person p1, Person p2) -> p1.name.compareTo(p2.name) );
```

Functions as first class citizen

```
public class HigherOrderFunctions {

 public static void main(String[] args) {

   List<Person> persons = Arrays.asList(
       new Person("Bjørn", 41), new Person("Kristian", 47), new Person("Erlend", 110));

   System.out.println("original : " + persons);
   Comparator<Person> byName = getSorter( );
   persons.sort( byName );
   System.out.println("sorted :   " + persons);
 }

 private static Comparator<Person> getSorter() {
     return (Person p1, Person p2) -> p1.name.compareTo(p2.name);
 }

}
```

```
original : [{ name: Bjørn, age: 41 }, { name: Kristian, age: 47 }, { name: Erlend, age: 110 }]
sorted :   [{ name: Bjørn, age: 41 }, { name: Erlend, age: 110 }, { name: Kristian, age: 47 }]
```

Some useful interfaces

Consumer

void accept(T t);

Accepts a single input, returns no result

Predicate

boolean test(T t);

Returns boolean value for given object

Comparator

int compare(T o1, T o2);

Negative, 0, positive for less, equal, greater

Exercise 1

Hints:

new methods in List

forEach, removeIf, sort

See the javadoc

Part 2

Optional

Optional

Java < 8 : no language construct for

“no data” or “nothing”

null, empty collection, empty string

Java 8:

```
public Optional<String> getAddress();
```

Creating an Optional value

```
Optional.empty();

Optional<Person> person = Optional.of(new Person("Bjørn", 41));

Optional.ofNullable(new Person("Bjørn", 41));

Optional.ofNullable(null);

Optional.ofNullable(obj.mayReturnNull());
```

Checking an Optional

```
if (person.isPresent()) {
  System.out.println(person.get());
}

person.ifPresent(System.out::println);

person.orElse(new Person("Anonymous", 35));

person.orElseGet(() -> new Person("Anonymous", 35));

person.orElseGet( Person::new );

person.orElseThrow(() -> new IllegalArgumentException("No such Person"));

person.get();
```

Stream

Stream

Layer on top of Collection

“Collection on speed”

New way of working with collections

- Lazy
- Doesn’t mutate collection
- Easy parallelization
- Powerful Collectors
- Functional programming style

Stream - sorting

Original list not changed!

“Perform operation(s) and collect”

```
List<Person> persons = Arrays.asList(
    new Person("Bjørn", 41),
    new Person("Kristian", 47),
    new Person("Erlend", 35));


List<Person> personsByName = persons.stream()
    .sorted( (p1,p2) -> p1.name.compareTo( p2.name )  )
    .collect(Collectors.toList());
```

Stream - sorting

Method references for readability

```
List<Person> personsByName = persons.stream()
    .sorted( (p1,p2) -> p1.name.compareTo( p2.name )  )
    .collect(Collectors.toList());

List<Person> personsByName2 = persons.stream()
    .sorted( Person::compareByName )
    .collect(Collectors.toList());
```

Stream - sorting

Lambdas for more for readability

```
Comparator<Person> byName = (p1, p2) -> p1.name.compareTo(p2.name);

List<Person> personsByName = persons.stream()
    .sorted( byName  )
    .collect(Collectors.toList());
```

Introduction to

functional programming

consepts

Functional programming

Filter

- Remove/keep certain elements

Map

- Transform elements to another type
- Nothing about Google Maps

FlatMap

- Map to lists, then flatten

Functional programming

Reduce (a.k.a folding)

- Reduce a collection to one (or zero) element
- With or without initial/identity value
- E. g. sum(), count(), min(), max()
- Can reduce to a different type

Chain several of these in a stream

Does not mutate original collection

FP - Filter

Pass each element to method (predicate)

Keep element if true is returned

Result can be zero to n elements

FP - Filter

Java < 8

```
List<Person> persons = Arrays.asList(
    new Person("Bjørn", 41),
    new Person("Kristian", 47),
    new Person("Erlend", 35));

List<Person> olderThan40 = new ArrayList<>();
for( Person p : persons){
  if( p.age > 40 ){
    olderThan40.add( p );
  }
}
```

FP - Filter

Java 8

```
List<Person> persons = Arrays.asList(
    new Person("Bjørn", 41),
    new Person("Kristian", 47),
    new Person("Erlend", 35));

List<Person> olderThan40 = persons.stream()
    .filter( p -> p.age > 40 )
    .collect(Collectors.toList());
```

FP - Map

Change the type

Returns the same number of elements

```
List<Person> persons = Arrays.asList(
    new Person("Bjørn", 41),
    new Person("Kristian", 47),
    new Person("Erlend", 35));

List<String> names = persons.stream()
    .map( p -> p.name )
    .collect( Collectors.toList() );
```

FP - Reduce

To zero or one element

returns Optional<Person>

```
List<Person> persons = Arrays.asList(
    new Person("Bjørn", 41),
    new Person("Kristian", 47),
    new Person("Erlend", 35));

Optional<Person> youngest = persons.stream()
    .reduce( (p1, p2) -> p1.age < p2.age ? p1 : p2 );
youngest.ifPresent( p -> System.out.println("Youngest: " + p.name) );
```

Specialized streams

Built-in mapping

- mapToInt -> IntStream
- mapToDouble -> DoubleStream
- mapToLong -> LongStream

Built-in “reducers”

- sum, count, average, min, max

Also sorting

- sorted

Stream/FP examples

```
List<Person> persons = Arrays.asList(
    new Person("Bjørn", 41),
    new Person("Kristian", 47),
    new Person("Erlend", 35));

persons.stream()
    .filter( p -> p.age > 40 )
    .sorted( Person::compareByName )
    .collect(Collectors.toList() );
```

```
persons.stream()
    .mapToInt( p -> p.age )
    .average();

persons.stream()
    .mapToInt( p -> p.age )
    .max();

persons.stream()
    .filter( p -> p.age > 40 )
    .reduce( Person::longestName );
```

Exercise 2

Part 3

Parallel stream

Parallel for loop

- A collection with elder people (>40)
  - … in parallel?

```
List<Person> persons = Arrays.asList(
    new Person("Bjørn", 41),
    new Person("Kristian", 47),
    new Person("Erlend", 35));

List<Person> olderThan40 = new ArrayList<>();
for( Person p : persons ){
  if( p.age > 40){
    olderThan40.add( p );
  }
}
```

Stream

```
List<Person> olderThan40 = persons.stream()
    .filter( p -> p.age > 40 )
    .collect(Collectors.toList());
```

Parallel Stream

```
List<Person> olderThan40 = persons.stream()
    .filter( p -> p.age > 40 )
    .collect(Collectors.toList());

List<Person> olderThan40 = persons.parallelStream()
    .filter( p -> p.age > 40 )
    .collect(Collectors.toList());
```

Collectors

Partitioning

Split collection in two

- Older or younger than 40
- Predicate (true/false) becomes key

```
Map<Boolean, List<Person>> partitioned =
    persons.stream()
    .collect(Collectors.partitioningBy(p -> p.age > 40));
```

Grouping

Group by same value

SQL group by

Each unique value (age) becomes key

```
Map<Integer, List<Person>> grouped = persons.stream()
    .collect(Collectors.groupingBy(p -> p.age));
```

Useful methods on stream

```
.limit( 10 )
.skip(10)
.findFirst()
.findAny()
.allMatch( p -> p.age > 40)
.anyMatch( p -> p.age > 40)
.distinct()
.min( Person::compareByName )
.max( Person::compareByName )
.count()
```

Generators

Builder

Fluent builder for generating a Stream

```
Stream s = Stream.builder().add(1).add(2).add(3).build();
```

Stream.iterate

for producing infinite ordered Stream

```
Stream multiplesOfThree = Stream.iterate(3, s -> s+3);
```

Random

Contains random generators for numbers

```
IntStream below100 = new Random(seed).ints(1, 100);
```

Exercise 3

Please give feedback

Slides

Content

Exercises

Presenters

Other

Bjørn: bjorn.hamre@gmail.com

Kristian: rasmantuta@gmail.com