Parsing the request

Part 2 - Creating a filter

Last example about parsing created SQL

We showed you in the last lecture (Parsing the request) how to write a small parser which created an SQL SELECT statement string based on the GET parameters.

It produced for instance the following result:

```
GET parameters:      ?type=Cider&min_alcohol=6&max_alcohol=8&max_price=15
SQL:                 SELECT * FROM product WHERE alcohol >= 6 AND alcohol <= 8
                         AND price <= 15 AND type = 'Cider';
Constraints:         [alcohol MIN 6, alcohol MAX 8, price MAX 15, type EQUALS Cider]
Valid constraints:   [alcohol MIN 6, alcohol MAX 8, price MAX 15, type EQUALS Cider]
```

But maybe the servlet shouldn’t bother with SQL?

- The Servlet should accept requests and send appropriate responses
- The Servlet should in this case send some Json data according to the request specification (the GET parameters)
- The Servlet shouldn’t talk to the database directly
- Let’s let the Servlet parse the request parameters and fetch data according to the parse result

Servlet responsibility

1. Read request e.g.: type=Cider&min_alcohol=6&max_alcohol=8&max_price=15
2. parse (understand) the request (create a filter/predicate)
3. send back Json with products which satisfy the criteria

We can let the servlet produce a predicate for acceptable products.

It can then use the predicate to ask some object or service for the products matching/satisfying the predicate.

It can then translate the products to some format (e.g. Json).

Building a Predicate from the GET params

1. Keep only the valid parameters (possibly report the invalid params too
2. Loop through the parameters (key=value pairs)
   a. Create and save a predicate for each valid parameter

```
p -> p.price() <= maxPrice
```

3. Combine the predicates to one big predicate
   a. `reduce(p -> true, Predicate::and)`

Introduction to Predicate

- Functional interface in `java.util.function`
- Often used as a lambda expression
  - `n -> n % 2 == 0     // lambda predicate for even number`
- Example:

```
List<String> strings = new ArrayList<>();
strings.add("ABBA");
strings.add("Bowie");
strings.add("Credence");
strings.stream()
       .filter(s -> s.startsWith("B")) // lambda Predicate
       .forEach(System.out::println);
Bowie
```

java.util.function.Predicate

Basically declares one abstract method

```
boolean test(T t);
```

So we could do either:

```
Predicate<String> stringPredicate = s -> s.startsWith("B");
// or:
Predicate<String> stringPredicate = new Predicate<>() {
  public boolean test(String s) {
    return s.startsWith("B");
  }
}
```

Reducing a List

If we have a list of elements and we want to reduce it to one single element using some operation, we can use the Stream class' reduce method.

Example using a list of integers:

```
jshell> System.out.println(ints);
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

// Reduce the list to the sum of all elements:
jshell> ints.stream().reduce(0, (a,b) -> a + b);
$5 ==> 55
```

A closer look at reduce

```
// Reduce the list to the sum of all elements:
jshell> ints.stream().reduce(0, (a,b) -> a + b);
$5 ==> 55
```

Reduce takes two arguments, the identity (start value) and the operation on two elements. For addition, identity is 0. For multiplication it is one:

```
// Reduce the list to the product of all elements:
jshell> ints.stream().reduce(1, (a,b) -> a * b);
$6 ==> 3628800
```

Reducing kind of works like this:

Add the identity at the start of the list, then put the operator between each element:

```
jshell> System.out.println(ints);
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

// Reduce the list to the sum of all elements:
jshell> ints.stream().reduce(0, (a,b) -> a + b);
// kind of: 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55
```

By the way!

How can you calculate the sum of ints between 0 and 10 in your head?

It must be 55! Because

```
0+10 +   // 10
1+9 +    // 10
2+8 +    // 10
3+7 +    // 10
4+6 +    // 10
5 = 55
```

:-)

Reducing a list of predicates

If we have a list of predicates (which all can result in true/false) and we want to combine them, we can reduce the list and use true as the identity and “AND” as the operation between all elements:

```
List<Predicate<Integer>> filters = new ArrayList<>();
filters.add(n -> n % 2 == 0); // A number is divisible by 2
filters.add(n -> n % 3 == 0); // A number is divisible by 3
Predicate<Integer> divisibleByTwoAndThree = filters
   .stream()
   .reduce(p -> true, Predicate::and);
```

Remember: reduce puts the identity first and an operator between. Think:

true && p1 && p2....

If we don’t want to use streams?

We could do something like this, to combine all predicates in a list:

```
Predicate combinedPredicate = p -> true;
for (Predicate p : predicates) {
  combinedPredicate = combinedPredicate && p.test();
}
return combinedPredicate;
```

Tying it all together

```
// We'll use a String[] with key=value pairs for this example

  List<Predicate<Product>> predicates = new ArrayList<>();
  List<String> validArgs = new ArrayList<>(Arrays.asList(args));
  // 1. Take away invalid parameters
  validArgs.removeIf(s->!isValidKey(s) || !isDouble(s.split("=")[1]));
```

See:
https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html#removeIf-java.util.function.Predicate-

Tying it all together, continued

```
// 2. Loop through the key=value pairs and create predicates
for (String arg : validArgs) {
  double value = Double.parseDouble(arg.split("=")[1]);
  switch(arg.split("=")[0]) { // Check what filter it is
    case "max_price": predicates.add(p -> p.price() <= value);
      break;
    case "min_price": predicates.add(p -> p.price() >= value);
      break;
    case "min_alcohol": predicates.add(p -> p.alcohol() >= value);
      break;
    case "max_alcohol": predicates.add(p -> p.alcohol() <= value);
      break;
    default:
      continue;
  }
}
return predicates;
```

Tying it all together, continued

```
    // 3. Combine the predicates to one big predicate
    predicates.stream().reduce(p -> true, Predicate::and);
```

```
// After that, we can call a method like:
private static List<Product> productsFilteredBy(Predicate<Product> filter) {
  return fetchProducts()   // all products
           .stream()       // as a Stream<Product>
           .filter(filter) // filter it!
           .collect(Collectors.toList());
}
```

Tying it all together - the call logic now becomes...

```
Predicate<Product> filter = filter(args); // filter using args
List<Product> products = productsFilteredBy(filter));


// Perhaps we can then use an object to convert the products to e.g. Json?

Formatter formatter = FormatterFactory.getFormatter();
String formattedData = formatter.format(products);

// Then, the servlet can write the formattedData as the response
```

Example result

GET: min_alcohol=10&max_alcohol=40&max_price=200

```
[
  {
    name:"Renat",
    price:100.0,
    alcohol:38.0
  },
  {
    name:"Val de Loire",
    price:69.0,
    alcohol:11.0
  },
  {
    name:"Baily's",
    price:169.0,
    alcohol:30.0
  }
]
```