Parsing the request

Parsing out GET parameters
Part 1

Prerequisites

You should preferably have seen lectures and read about

- Servlet technology - introduction to servlets
- HTTP - in particular GET parameters
- Java enum types
- Java Set interface
- Java Map interface
- The ternary operator (String type = num % 2 == 0 ? "even" : "odd";)
- For-each-loop
- Java arrays
- SQL SELECT

Re-cap What are GET parameters?

Given the following URL:

http://localhost:8080/search/products/all?min_alcohol=5&max_price=20&type=Cider&nisse=apa&max_alcohol=60

The GET parameters occur after the question mark:

min_alcohol=5&max_price=20&type=Cider&nisse=apa&max_alcohol=60

They can be accessed via methods on the HttpServletRequest object parameter of the doGet() method of a servlet.

Their use is to give arguments to a web application (like a servlet)

We could use the parameters to filter products

A request with parameters

```
type=Cider&min_alcohol=6&max_alcohol=8
```

could be used to produce an SQL statement (for instance) for fetching all products of type Cider whose alcohol is between 6 and 8 percent:

```
SELECT * FROM product
  WHERE alcohol >= 6 AND alcohol <= 8 AND type = 'Cider' ;
```

In order to do so, we need to “parse” the parameters and understand what they mean.

Example GET parameters

min_alcohol=5&max_price=20&type=Cider&nisse=apa&max_alcohol=60

represent the following map values:

```
min_alcohol => 5
max_price   => 20
type        => Cider
nisse       => apa
max_alcohol => 60
```

You decide what parameters are legal (allowed)

The `nisse => apa` parameter is clearly not a legal parameter in our example. So how do we check the parameters?

Let's say that the following parameter keys are legal:

```
alcohol
min_alcohol
min_price
max_alcohol
max_price
price
name
type
```

How can we then check all the parameters to see if they are legal?

Looping through the parameters

We can iterate over the parameters like this:

```
for (Object key : request.getParameterMap().keySet()) {
  // Check if key.equals() some legal keys
}
```

The `request.getParameterMap()` method returns a java.util.Map<java.lang.String,java.lang.String[]>. A map is a data structure with key,value pairs. So each parameter key has a value which is a String[].

The keySet() method of a Map returns a Set of all keys (Strings in this case).

Checking the parameters in a loop

Given the following parameters:

```
min_alcohol => 5
max_price   => 20
type        => Cider
nisse       => apa
max_alcohol => 60
```

The loop could check that each “key” is one of the above allowed keys. We’ll skip validating the values here (i.e. checking that max_alcohol is a number between 0 and 100 etc).

Parse loop version 1

Parsing means analyzing and understanding some input. The first attempt to validate the keys of the request could be the loop and if statements:

```
for (String key : (Set<String>)map.keySet()) {
  if (key.equals("alcohol") ||
      key.equals("min_alcohol") ||
      key.equals("min_price") ||
      key.equals("max_alcohol") ||
      key.equals("max_price") ||
      key.equals("price") ||
      key.equals("name") ||
      key.equals("type")) {
    // it's a valid key, so let's store the
    // key-value pair somehow
    value = map.get(key)[0]; // first element
  } else {
    // here we've found an invalid key
  }
}
```

Problems with IF statements

What to do when the rules for valid keys changes? Should we add, change or remove if-statements?

An alternative to if-statements could be to use a list (or set), and check if the current key of the loop is a member of that list (or set):

```
// validFields is a String[] with the valid fields

if( Arrays.asList(validFields).contains(key) ) {
  // valid key
} else {
  //invalid key
}
```

If we don’t like if

We could create a class representing a key-value pair and let the instances of the class store:

- key : String
- value : String
- type : <<enum>> {MIN, MAX, EQUALS, INVALID}

Then, we don’t need the if-statement in the loop, every key-value can be represented as an object whose type is either INVALID or some other type.

Eliminating the IFs

```
// map is a Map<String, String[]> with the parameters
// Constraint is the class representing a key-value pair with a type

public void parse() {
  for (String key : (Set<String>)map.keySet()) {
    String value = map.get(key)[0]; // the map is String key, String[]
values
    // Remove the min_ or max_ part from the key "min_price" -> "price"
    String field = keyToField(key);
    // Is it MAX, MIN, EQUALS or INVALID?
    Constraint.Type type = keyToConstraintType(key);
    // Store this constraint in the constraints list
    constraints.add(new Constraint(field, type, decode(value)));
  }
}

// constraints is a List<Constraint>
```

Eliminating the IFs

```
// A declarative style is to say what you want, not how you want it.
// We use helper methods to achieve that and eliminate if statements:

public void parse() {
  for (String key : (Set<String>)map.keySet()) {
    String value = map.get(key)[0]; // the map is String key, String[]
values
    // Remove the min_ or max_ part from the key "min_price" -> "price"
    String field = keyToField(key);
    // Is it MAX, MIN, EQUALS or INVALID?
    Constraint.Type type = keyToConstraintType(key);
    // Store this constraint in the constraints list
    constraints.add(new Constraint(field, type,  decode(value)));
  }
}
// constraints is a List<Constraint>
```

keyToField(key)

```
// Should return the name of a field so that prefixes
// max_ and min_ are removed, if they are there and valid

private String keyToField(String key) {
  if( ! Arrays.asList(validFields).contains(key) ) {
    return key;
  }
  return key.matches("^min_.+") || key.matches("^max_.+") ?
         key.split("_")[1] : key;
}
// The “Ternary operator” works like this:
// (test_evaluates_to_true) ? value : value_if_test_fails
```

```
public class Constraint {
  private String field;
  private Type type;
  private String value;
  enum Type {
    MAX,
    MIN,
    EQUALS,
    INVALID;
  }
  public Constraint(String field, Type type, String value) {
    this.field = field;
    this.type = type;
    this.value = value;
  }
  public String field() { return field; }
  public Type type() { return type; }
  public String value() { return value; }
  public String toString() { return field + " " + type + " " + value; }
}
```

keyToConstraintType(key)

```
// A legal/valid key which starts with max_ has type MAX,
// min_ has type MIN and all other valid keys has type EQUALS
// Invalid keys have type INVALID

private Constraint.Type keyToConstraintType(String key) {
  if( ! Arrays.asList(validFields).contains(key) ) {
    return Constraint.Type.INVALID;
  }
  return key.matches("^min_.+") ?
           Constraint.Type.MIN :
         key.matches("^max_.+") ? Constraint.Type.MAX :
           Constraint.Type.EQUALS;
}
```

How would we create a SELECT statement?

```
private String sql(List<Constraint> conds) {
  String condition = "";
  for (Constraint con : conds) {
    switch (con.type()) {
      case MAX: condition += con.field() + " <= " + con.value();
        break;
      case MIN: condition += con.field() + " >= " + con.value();
        break;
      case EQUALS: condition += con.field() + " = '" + con.value() + "' ";
        break;
    }
    condition += " AND ";
  }
  if (! condition.equals("")) { // remove last AND, prepend with WHERE
    condition = condition.substring(0, condition.length()-6);
    condition = " WHERE " + condition;
  }
  return "SELECT * FROM product" + condition + ";";
}
```

Example runs

?apa=groda&min_bad=33
SQL: SELECT * FROM product;
Constraints: [apa INVALID groda, min_bad INVALID 33]
Valid constraints: []

?apa=groda&min_bad=33&max_price=44
SQL: SELECT * FROM product WHERE price <= 4;
Constraints: [apa INVALID groda, min_bad INVALID 33, price MAX 44]
Valid constraints: [price MAX 44]

?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]