Laboratory Sheet 9+10

The deadline for submission is 48 hours after the end of your scheduled laboratory session in week 11 *(1 –5 December 2014).*

Aims and objectives

- experience code reuse and refactoring
- interact with large complex library frameworks
- explore simple web-based development
- design and implement a testing strategy

Set up

Download `Laboratory9.zip` from moodle and unzip this file. You will obtain two folders `JarFiles` and `sparkcrates`. Inside Eclipse, create a new project called Lab9 (or similar). Create a new package called sparkcrates and add the Java source code files in the unzipped `sparkcrates` folder into this package. Also locate your Lab6 source code files and copy the package `fillingcrates` (with its Java source code files) into this new Lab9 project. Delete the `CratesProgram` class – you are going to define a main method in the `sparkcrates.WebCrates` class instead.

Add all the jar files in the `JarFiles` folder to the project via the Project > Properties > Java Build Path > Libraries > Add External Jars configuration. You should be able to select them all in a single step.

Instructions

Remember the crate filling exercise we did for lab 6? We are going to revisit it and transform it into an interactive web-based system. Since this is a more complex exercise, you will have two lab sessions (weeks 10 and 11) to work on this code. You do not have to submit your code until next week, 48 hours after your last lab of the semester.

We will be using the Spark micro-framework for Java web applications. There is helpful documentation at http://sparkjava.com/documentation.html which you may wish to refer to.

Task 1 (warmup)

Write a simple 'hello world' spark web application in the `sparkcrates.WebCrates` class. You will need write a main method that registers a single callback to print "hello" on a get operation. The source code below gives a simple template for you to adapt.

```
package sparkcrates;

import spark.Route;
import spark.Request;
import spark.Response;
import spark.Spark;

public class WebCrates {

   public static void main(String[] args) {
      Spark.get(“/hello”, new Route() {
         @Override
         public Object handle(Request request, Response response) {
            return "Hello World!";
         }
      });

   }

}
```

When you run this Java class, you should see some logging information print out in the console pane in Eclipse. The final lines should say something like:

```
== Spark has ignited ...
>> Listening on 0.0.0.0:4567
```

Now you can open a web browser and access the URL
http://localhost:4567/hello

You should see the “hello world” message in your browser window.

Note that the application carries on running after the main method returns, due to background threads continuing to run. To stop the web server, you need to click the red stop button in Eclipse, or quit the Eclipse IDE altogether. It is not possible to run more than one instance of this web application at once – you will get an error:

```
java.net.BindException: Address already in use.
```

If you are able to use Java 8 (as installed on the Boyd Orr lab machines) then you can rewrite the above Hello World code much more concisely using a lambda expression:

```
package sparkcrates;
import static spark.Spark.*;
public class HelloWorld {
    public static void main(String[] args) {
        get("/hello", (req, res) -> "Hello World");
    }
}
```

Now you should investigate how to change the output behaviour. Instead of returning a simple string, you might try to generate more complex HTML code. Extra helper methods might be required here, possibly using StringBuilders.

Task 2 (Lab 9 Submission)

Given a static `List<FillableContainer> crates` belonging to class `WebCrates`, you need to add a web-based callback routine to display the current list of crates. http://localhost:4567/showcrates should trigger this action.

The simplest approach to visualizing the crates list is using the `ArrayList toString()` method and defining an appropriate `toString` for `Crate` objects. More complex approaches involving HTML tables or CSS elements are possible, requiring a for-each loop over `Crate` instances in the crates list.

Then you should define another callback to add a new empty crate with a specified capacity to the crates list. http://localhost:4567/add/50 should add a new empty `Crate` of size 50 to the crates list. Use a Spark Route pattern (see http://sparkjava.com/documentation.html) to do this, accessing the size via a named parameter using the `Request.params()` method.

(Apology to web programming experts: I am abusing CRUD / RESTful conventions here, since state updates should use a PUT or POST method. But these are more difficult to set up from simple web browser window, so we will restrict our implementation to GET requests for now.)

Task 3 (Lab 9 Submission part 2)

Now we want to add support for the three bin packing (crate-filling) algorithms we explored in lab 6. Ideally, we want to issue requests like
http://localhost:4567/fill/first/50 or http://localhost:4567/fill/best/99 or
http://localhost:4567/fill/worst/20
which will invoke the appropriate crate-filling algorithm to store the specified amount in the static `crates` list belonging to the WebCrates class.

You may need to modify your `AbstractFit` class (or subclasses) so that they can operate on pre-existing lists of crates, rather than creating a new `ArrayList` when the `AbstractFit` constructor is called.

Then you will need to register a new get callback for the fill URL, that takes two parameters – one specifying the fitting algorithm, the other specifying the amount – as shown in the above example URLs.

Task 3 (Lab 9 Submission part 3)

There is a skeleton JUnit test file supplied, called `TestWebCrates`. As usual you will need to add the JUnit library to the project. This JUnit test file shows how to instantiate the Spark web server framework and how to test simple GET requests.

You need to write test cases to verify the behaviour of your web crates system – following the example test case given. Also look back at previous JUnit test classes from earlier labs for more inspiration.

Submission details

You will want to submit files: `Crate.java`, `AbstractFit.java`, `FirstFit.java`, `BestFit.java`, `WorstFit.java` – these are all in package `fillingcrates`.
You will want to submit files: `WebCrates.java`, `TestWebCrates.java` plus any other Java files you create for your system. These are all in package `sparkcrates`.
You are allowed to submit a maximum of 10 source code files in the Moodle submission slot for JOOSE2 lab 9.

Outline Mark Scheme

Below is an illustrative mark scheme. This is a large and challenging project so tutors will use their discretion. One mark will apply to both labs 9 and 10 (i.e. this is a double-submission worth 5% of final course grade).

A: Correct code, with good coverage test suite. Supports all three *-fit algorithms. Attractive graphical output in web browser. Code has sensible structure and is well commented.

B: Mostly correct code, with fairly good coverage test suite. Supports all three *-fit algorithms. Sensible graphical or text-based output in web browser. Code has reasonable structure and some comments.

C: Code compiles and behaves fairly reasonably. Some attempt at tests. Supports at least one fit algorithm. Basic output in web browser. Code has fair structure.

D: Code may not compile. Minimal attempt at tests. Attempted to support at least one fit algorithm.

E: Code does not compile. No tests. Attempted to support at least one fit algorithm.