Lab 4 Part b

We continue our work on the assignment with the overall index.html structure, and then the Scorecard UI.

Overall structure

What we want to do now is set up a bit of our overall page layout. Here is our overall plan:

- The page contains a "Scorecard area" near the middle top, that displays the score and contains a "reset" button.
- Below the scorecard area is the "doors area" which contains the three large doors that the user gets to click.
- Below the doors area is a "messages" area that the system will use to give instructions to the user.

Before moving on, make a new issue in GitHub titled "overall page layout" and list the above three items in a comment. Add it to your milestone.

So the top level interface could be written in HTML as follows (add these inside the body tag in the index.html page, and before the script section there):

```
<div id="scorecard"></div>
<div id="doors"></div>
<div id="messages"></div>
```

We will obviously need to format them later once we have something more to show. For now let's leave it at that. Make a commit and link it to your issue about the UI. We will return to it later.

Scorecard UI

The ScorecardUI class is responsible for setting up the HTML for the scorecard, along with any needed jQuery methods to react to user actions, and methods to update the values when the score is updated. Our goal is for this class to not be directly related to the Score class itself. There will therefore be some amount of duplicated information.

This class will not be built in a test-driven-development style, as it involves the UI. What we will do is inspect the UI every few steps, to make sure things look OK. We will do this in a copy of the index.html file, which we will call testUI.html. The idea is that both files will be loading the same modules, but the UI file will have extra stuff that we can use to examine the behavior.

Create such a copy now, at the same level as the index.html, and open it in the browser.

Let's start by making an issue about building the scorecard UI (give it the *scorecard* label and add it to the milestone). And let's add some comments to it:

- The scorecard UI class will be given DOM element as a parameter, and it will store it.
- When we use this class in our application, this element will be the div with id "scorecard"
- The class needs to add labels and values for the four counters we wish to report, probably
- The class needs to have an 'update' method that is given some information about which value
- The class will be managing a "Reset" button, and when the button is pressed it triggers a "

We start by setting up the basics of the ScorecardUI class.

- Create a new js/scorecardui.js class star with an export default class ScorecardUI {} in it.
- Add a constructor to the class, that takes an argument called $el and stores it in a instance variable this.$el. The dollar signs here are just a reminder that this is to be a jQuery element and that jQuery methods will apply to it.
- On our testUI.html page, we want to create a little startup script that loads the ScorecardUI module and calls this constructor. As this will grow over time, we'll instead put it in a little "test script". So add the following before the closing </body> tag in testUI.html:

  html <script type="module" src="test/scorecarduiTest.js"></script>html

  and create this new file test/scorecarduiTest.js in the test folder. In it put

  ```
  import ScorecardUI from '../js/scorecardui.js';
  let scorecard = new ScorecardUI($("#scorecard"));
  ```

  Then save and reload the page and make sure you get no errors.
- To make sure that the constructor is getting called, add a "console.log" call in the constructor, then reload the page to see that a message is printing, then delete the console.log.
- Next we need to write the initialize function. It is given an object o that has properties switchWins, switchLoses, etc similar to the Score object. It is then using this object to create the view, and uses the $el element to place information onto the screen. Start by adding the following lines in the test/scorecarduiTest.js file:

  ```
  scorecard.initialize({
      switchWins: 5, switchLoses: 3,
      stayWins: 2, stayLosses: 4
  });
  ```

  Reload the page and you should see these lines fail because there is no function initialize.
- Now let's add this method to the ScorecardUI class in js/scorecardui.js:

  ```
  initialize(o) {
      this.$el.html(template(o));
  }
  ```

  This calls the template function on the object o. Place the template function at the bottom of the file, outside of the 'ScorecardUI class:

```
function template(o) {

}
```

This function uses a "template string", which is a new kind of string introduced in ES6. It is delineated by the single backticks instead of quotes, and it contains in it segments like '$o.switchWins$ that contain executable Javascript code. The template string executes that code, then replaces the whole segment with the code's result.

```
return `
<input type="button" id="resetButton" value="Reset"></input>
<div class="scoreSection"><h2>SWITCH</h2>
   <label for="switchWins" >Wins:
      <input type="text" id="switchWins" name="switchWins" value="${o.switchWins}"></inp
   </label>
   <label for="switchLosses" >Losses:
      <input type="text" id="switchLosses" name="switchLosses" value="${o.switchLosses}">
   </label>
</div>
<div class="scoreSection"><h2>STAY</h2>
   <label for="stayWins" >Wins:
      <input type="text" id="stayWins" name="stayWins" value="${o.stayWins}"></input>
   </label>
   <label for="switchLosses" >Losses:
      <input type="text" id="stayLosses" name="stayLosses" value="${o.stayLosses}"></inp
   </label>
</div>
`;
```

So what is going on here? The backtick next to the return marks the beginning of the string, while the backtick at the end marks the end of the string. And inbetween we can freely write the HTML code we want. Notice the parts that say `${o.switchWins}`, these are executable Javascript that will look at that property from the object `o` and replace that placeholder with the corresponding value.

Refresh your testUI page to make sure this worked. It doesn't look pretty yet, but you should be seeing the scores. Take a moment to study this HTML code:

- There is a button (`input` with a `type` of `button`) at the top that we will use for resetting everything. We'll obviously need to make it a lot bigger and make it span the width of the element.
- There are two `div` elements, one for the SWITCH case and one for the STAY case. They each contain an `h2` element with the case title.
- Then in those `div`s we have two input elements wrapped in labels, one for the wins and one for losses. Notice how the `for` attribute of the label matches the `name` attribute of the corresponding input, this links the two together.

This is a good time to make a commit, referring to initial UI for the ScorecardUI class. Make sure you link it to the corresponding issue number!

We will at some point worry about the look-and-feel of the whole thing, but for now let us focus on functionality. The class needs to have an `update` method that is given

simply the name of the updated field, for example "switchWins", and an updated value. It is then responsible for updating a corresponding value on the table. Let's work on creating this method.

- We start by writing some test code in our test/scorecarduiTest.js class. Right after the call we made to initialize, let's make a call to update:

  ```
  scorecard.update("switchLosses", 3 + 1);
  ```

  Reload the page and you should see it fail in the console log, because there is no update method. Now let's go into our ScorecardUI class, and add that method, right below the initialize:

  ```
  update(field, newValue) {
      let $inputEl = this.$el.find("#" + field);
      $inputEl.val(newValue);
      highlight($inputEl);
  }
  ```

  The first line finds the actual UI field, the second updates the value, and the third calls a helper function, that you should put further down in the file outside of the class:

  ```
  function highlight(el) {
      let currentColor = el.css('background-color');
      let targetColor = "#f47142";
      el.css('background-color', targetColor);
      setTimeout(() => el.css('background-color', currentColor), 1000);
  }
  ```

- Alright we have some basic update functionality working! Go ahead and make a commit, and refer to the appropriate issue number.

The reset button and observables One last thing we need to set up is what happens when the "Reset" button is clicked. Of course the values on our Score model will need to be reset, but our little UI doesn't know anything about the Score class, and we like it that way. What the UI will need to do when the Reset button is clicked, is send out an announcement, and hope someone is listening. Therefore we would like to use the observer pattern for that.

- Let's start with a little test code for that. In test/scorecarduiTest.js, we write the method that would add an observer to listen for the click:

  ```
  scorecard.on("resetRequested", () => console.log("Reset called!"));
  ```

  Reload your page and you should see an error, about the method "on" not existing.

- To fix that, we will turn our class into an "Observable". This is similar to the Event class we discussed in class. Take a look at the js/observable.js file to see how it is implemented, and the test/observable.spec.js file for some usage examples. We need to now use Observable in our class, by making it extend Observable. To do that, we need to change the first line to the following:

```
import Observable from './observable.js';

export default class ScorecardUI extends Observable {
   constructor($el) {
      super();
      this.$el = $el;
   }
```

The method super here calls the constructor of Observable, to make sure that it does all the work it needs to do.

Now we should be seeing no errors, but when we click the button nothing happens. We need to make sure that the UI triggers a message when the reset button is clicked. We can do this in the initialize method. After you set the html of the element, add the following:

```
this.$el.find("#resetButton")
    .on("click", () => this.trigger("resetRequested"));
```

So we find the button, and we register for a click event on that button, so that when the button is clicked we can trigger the "resetRequested" message. Click the button now and you should be seeing your console messages pop up.

And this concludes our ScorecardUI class! We'll work on the CSS later. For now, make a final commit and close the issue.

Notice how our ScorecardUI class had very little real logic in it. It is simply there to protect the rest of the application from knowing about specific UI details. But we don't want it to be too smart:

- It does not know what effect the resetRequested message may have on the system, or who would be responsible for acting on it.
- It does not know any of the rules based on which the four values it shows will be changing, nor does it care. We can test it by changing those values in some arbitrary way from the scorecarduiTest.js file.

Next up we will work on the ScorecardController class, which will coordinate between the Score model and the ScorecardUI view.

ScorecardController Now it is time to work on the Scorecard controller. Start by creating an issue:

```
Scorecard controller is the interface between the Score model and the ScorecardUI class. It k

- Its constructor would be taking as input two things: A UI instance and a score model. We wi
- It will register to listen for a change event on the Score model. We'll need to make the Sc
- It will register to listen to a reset change from the UI, then pass that reset instruction
```

Let's get started!

- We'll need a test/scoreController.spec.js file, and you'll need to also add to the tests.html file a <script> tag that loads it.

- Next we need to import Score and ScoreController. Add the following lines to the spec file, and run your tests to make sure they fail, as the second file hasn't been created yet.

```
import Score from '../js/score.js';
import ScoreController from '../js/scoreController.js';
```

- Now create the scoreController.js file, and add an empty starting class to it to make the tests pass:

```
export default class ScorecardController {

}
```

- Now let's start work on our first test. We'll need some boilerplate:

```
describe('ScoreController instances', () => {
   let score;
   let controller;
   beforeEach(function() {
      score = new Score();
      controller = new ScoreController(score);
   });
   it('register to listen to score changes', () => {

   });
});
```

  We use the beforeEach trick again, as we are likely to use the score and controller on pretty much every test. We have not done anything with the UI part yet; when we do we'll need to update the call to ScoreController(score) to also take our mock UI as input.

Our first controller test is in theory going to be relatively easy: We just need to make sure that the controller registers to listen to score changes. For this we'll need a few ingredients. First, we'll need to turn the Score class into an observable, and add some tests to that effect. And those test will take us down the rabbit hole of asynchronous testing.

Let's start by turning the Score class into an observable. Do this now, by following the same steps we did for ScorecardUI, namely importing the Observable class and then making the Score class definition extend Observable, and adding a call to super() in the constructor. Do this now and make sure your tests still pass.

We still haven't set up the class to trigger anything, but let's work through our first test for this. Go back to the Score class test file, test/score.spec.js.

Asynchronous events and testing Let's discuss the problem we will need to face with these tests. If you look at the Observable class, you will notice that it triggers its events **asynchronously**: It creates timeouts for the events to execute once the current function is done. This presents some challenges. For instance imagine that we add our next test as follows:

```
it('trigger messages when values change (switchWins)', () => {
   let h = (msg, value) => {
      expect(msg).to.equal('switchWins');
      expect(value).to.equal(2);
   };
   score.on('change', h);
   score.trigger('change', 'switchWins', 1);
});
```

So what we do here is prepare a function `h`, then we give it as a handler to the observable. Then we call `trigger`, and expect that the function `h` will be called and will do the two checks listed there. (actually in practice we would not call `trigger` ourselves, but we haven't hooked that up yet). Put that test in and run that code and see what happens. This code should be failing, but you'll see that the tests technically all pass (even though you see an uncaught exception pop up on the console).

Note that we purposefully wrote this test with the wrong value in the trigger call; this test should be failing, but it does not fail in a normal way, and we need to fix that.

So what is going on here? The problem is that when trigger happens, the observable class sets up a call to occur *in the future*, and in the meantime it returns. The flow of control will then go to the end of the `it`, which will finish normally and therefore the test will register as "done" and succeeding. So the problem is that our function `h` never gets a chance to run until after the `it` call has been completed.

In order to deal with these problems, the one solution is to use some mock observable structure that works synchronously rather than asynchronously.

The other approach, which we will take, is to accept that the operations will happen asynchronously, and to adapt the tests accordingly.

Mocha offers us an important tool in this process, via the `done` parameter. Our tests will look something like this:

```
it('...', (done) => {
   ... do stuff ...
   ... possibly asynchronously ...
   ... call done()  when the test is trully done ...
});
```

The idea of this is that the system does not consider the task completed until that function `done` is called. It wait wait for asynchronous events to complete if they are the ones that trigger the `done`. For example our test could look as follows:

```
it('trigger messages when values change (switchWins)', (done) => {
   let h = (msg, value) => {
      expect(msg).to.equal('switchWins');
      expect(value).to.equal(2);
      done();
   };
   score.on('change', h);
   score.trigger('change', 'switchWins', 1);
});
```

Run this test instead and see how it fails a bit more gracefully. Now comment out the trigger line and see what happens: The test will time out and get reported as failing if done is never called.

In order to understand this better, take a look at the test/observable.spec.js file, where done is used extensively.

Now on to our Score tests. Replace this test you just put with the following:

```
it('trigger messages when values change (switchWins)', (done) => {
   let h = (msg, value) => {
      expect(msg).to.equal('switchWins');
      expect(value).to.equal(1);
      done();
   };
   score.on('change', h);
   score.addResult(Score.ACTION_SWITCH, Score.RESULT_WIN);
});
```

This test should fail, because we have not done anything in addResult that would trigger an event. So run your tests and you should see this test time out after 2 seconds.

Now we need to make this test pass. If you look in your addResult function, you can imagine various places where we can add this kind of trigger. But if we are not careful, we'll add with 15 different trigger calls, when we really should have one: whenever any of the score value changes, we should have a trigger.

What this means is that we should create a little set function that sets a value. And this function will set the value and then call trigger. And all other places that need to set a value will go through set.

Let's see how we can make that happen.

- Start by adding this new method in the Score class:

  ```
  set(property, value) {
      this[property] = value;
      this.trigger('change', property, value);
  }
  ```

- Now, replace our this.switchWins += 1; line with this.set('switchWins', this.switchWins + 1);. This should make the test pass.

- We could now create 3 copies of this test to handle the other cases, but clearly that could get very unwieldy very quickly. Instead, we will use the **spy** functionality we created for TaskApp. Recall that a spy monitors calls to a function, and can provide you information about what that function did. In our case, we will monitor calls to trigger. This has another great benefit: We don't need to wait for the asynchronous call any more. Therefore change our current test to read:

  ```
  it('trigger messages when values change', () => {
     let spy = new Spy(score, "trigger");
     score.addResult(Score.ACTION_SWITCH, Score.RESULT_WIN);
     expect(spy.argumentsOfCall(0)).to.deep.equal(['change', 'switchWins', 1]);
  });
  ```

You will also need to add an import call for the Spy class near the top of the test file. Your tests should still pass after that.

- Now, add two lines to the above test, one calling addResult with a "switch and loss" information and one expecting the arguments of the call indexed at 0 to deep equal ['change', 'switchLosses', 1]. Your tests should now fail. Then adjust the corresponding line in the Score to call set on a switch loss as well.

- Repeat this process for the stay wins and stay losses.

- Before we move on, did you notice that this was somewhat repetitive? That's never a good idea, and it also forces us to keep in sync things that should always be the same, for example in this line this.set('switchWins', this.switchWins + 1); when we switch things up we need to keep the string and the this.... parts in sync. Ideally we should not need to do that. We can avoid it by creating an incr method instead! Let's do this now. Add a new method to the Score class:

```
incr(property) {
  this.set(property, this[property] + 1);
}
```

  And then replace the four this.set lines you have with this.incr('switchWins'); etc. Make sure your tests still pass.

- We should also update the assignments that take place in the reset method. Make sure you replace the lines like this.switchWins=0; to instead say this.set('switchWins', 0); and similarly for the others.

Now that we have the Score class all in order, time to return to the ScoreController class and its tests.

Now, the first test we wanted to write for the ScoreController is simply a test that the controller signs up to be notified when values change.

- In order to test this, we need to "spy" on the score object's "on" method, and then recreate the controller and see if it registered itself:

```
it('register to listen to score changes', () => {
  let spy = new Spy(score, 'on');
  new ScoreController(score);
  expect(spy.numberOfCalls()).to.equal(1);
  let [topic, handler] = spy.argumentsOfCall(0);
  expect(topic).to.equal('change');
});
```

  Run your test and make sure it fails.

  There is a new syntax in the above code, namely the [topic, handler] = ... bit. This is called **Destructuring assignment** and you can read more about it here[1].

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

- In order to make the test pass, we'll need to do some work on the ScoreController constructor:

```
constructor(score) {
   this.score = score;
   this.score.on('change', (property, value) => this.modelChanged(property, value));
}
```

  Check that the test now passes. Note that we have not yet implemented the modelChanged method; that's not a problem yet because we have not triggered the event yet.

- Next up, our controller is supposed to call the initialize method of the ScorecardUI class when it starts up, and to give it a copy of the model. In order to make that happen, we'll need to let the controller constructor take a second parameter, namely the ui instance, and store it in an instance variable. As we don't want to use the actual UI in our tests, we will use a "fake UI" class. You can create this class directly into the scoreController.spec.js test file, before all the describe part:

```
class FakeScorecardUI {
   initialize(o) {}
   update(field, newValue) {}
}
```

  As you can tell, the class doesn't actually need to do anything, yet; we'll use spies to look into calls to those methods.

- Now we need to hook this class in. In our test file, right before the beforeEach call, add a let ui; line, and inside the beforeEach call change the lines to be:

```
beforeEach(function() {
   score = new Score();
   ui = new FakeScorecardUI();
   controller = new ScoreController(score, ui);
});
```

  Run your tests to make sure we didn't break anything. You should also fix the other call to new ScoreController(..) to also take the second parameter.

- Now we need to write our test. In our test, we'll need to recreate the controller after we've set up to spy on the initialize method:

```
it('initialize the ui when they start', () => {
   let spy = new Spy(ui, 'initialize');
   new ScoreController(score, ui);
   expect(spy.numberOfCalls()).to.equal(1);
   expect(spy.argumentsOfCall(0)).to.deep.equal([score]);
});
```

  Make sure to understand what this test code does, and why it currently fails.

- Now let's make it pass, by fixing the controller constructor, in js/scoreController.js:

```
constructor(score, ui) {
    this.score = score;
    this.ui = ui;
```

```
        this.ui.initialize(score);
        this.score.on('change', (property, value) => this.modelChanged(property, value));
    }
```

Run your tests and make sure they pass.

This might be a good time to make a commit. Make sure to reference the appropriate issue number in your message!

Now it is time for our first "real" test on the controller. We need to make sure that when the model triggers a change event, then the controller calls the UI's update method with the correct arguments. We can easily spy on the update method. But the problem we have is that the controller will respond to the trigger event asynchronously. We therefore want to use the done parameter to control when the test ends. We'll also enhance our spy with an extra function parameter to call when the function it spies on is getting called. Take a look at our spy.js code to see how that third parameter behaves.

Here's how that all looks like, add it to the scoreController test file:

```
it('call the ui update when the model triggers a change', done => {
    let spy = new Spy(ui, 'update', (field, value) => {
        expect(field).to.equal('switchWins');
        expect(value).to.equal(score.switchWins);
        done();
    });
    score.trigger('change', 'switchWins', score.switchWins);
});
```

So we set up a spy on the update method, and we also provide a third "callback" function that should be called instead of the actual update method. That callback checks that update was given the correct arguments by the controller, and then calls done to end the test.

Run this test now and you should see it time out.

To make the test pass, we simply need to finally implement the modelChanged method of the controller, by adding the following method to the ScoreController class:

```
modelChanged(property, value) {
   this.ui.update(property, value);
}
```

You should see your tests pass after this.

In order to wrap up our controller, we need one final component. We need for our controller to set up to listen to the ui for a resetRequested trigger, and call the model's reset method when that happens. Here is an overview of the steps you will need to implement for this:

- First, have our FakeScorecardUI class extend Observable. You'll need to also import Observable to make that happen.

- Write a 'register to listen to ui resetRequested' test, which is similar to the 'register to listen to score changes' test, but instead spies on the ui's resetRequested method. Run your test and watch it fail because the number of calls is 0 instead of 1.
- Add a line to the ScoreController constructor to have it register with the ui to listen to the resetRequested event. It should call this.resetRequested() in response to that event (instead of modelChanged. That event will not need to pass any parameters, so your handler does not need to take any parameters. Make sure your test passes.
- Write a 'call the model reset method when the ui triggers resetRequested' test that is similiar to the 'call the ui update when the model triggers a change' event: It should set up a spy on the score's reset method, with a handler that simply calls done. Then it should use ui.trigger('resetRequested') to trigger the event, and then it is done. Run your tests to see this test throw an error because the resetRequested method that the controller offered as its handler doesn't exit yet.
- Add a resetRequested method to the controller, with no parameters and an empty body. Watch your test time out.
- Implement the resetRequested method in the controller, to simply call the score's reset method. Watch your tests pass.

And this concludes our ScoreController class and the overall "Scorecard" component! Make sure you make a commit, close the corresponding issue, and check off the corresponding box in our master list.

This concludes part b of the lab. Due to its size, the lab is broken into pieces. Continue to part c[2]

[2] ./4c.html