Lesson 14

IIFEs, private variables, Handlebars templates,
Feedr project intro

Lesson Objectives

- Use closures to implement private variables
- Use closures and IIFEs to implement the module pattern
- Use Handlebarsjs templates to separate models from views
- Begin Feedr project (grabbing news feeds from APIs and displaying them on a web page)

Going over the homework

All code for today:

- http://bit.ly/jsdev2-lesson14-starter-code

Immediately Invoked Function Expressions

- What if we want to create a new scope in the code — a place where we can have private variables that won't be visible to the rest of the code — but we don't really have any need to make a function that will be run later?

- We can just make a function literal and run it right now.

- This is called an **I**mmediately **I**nvoked **F**unction **E**xpression (IIFE).

- To do this, we just put the parentheses at the end of the function, to show that we're running it.

Immediately Invoked Function Expressions

```
for (var counter = 3; counter > 0; counter--) {
  console.log(counter);
}
```

can become

```
(function() {
  for (var counter = 3; counter > 0; counter--) {
    console.log(counter);
  }
})();
```

Immediately Invoked Function Expressions

- There are a number of reasons you might want to use these.

- We'll go over a couple of them:
  - One solution to the "mutation trap" homework problem
  - Making clean modules of code that don't pollute the rest of the code with irrelevant variable names

Going over the homework again (IIFE version)

Private variables

- Closures can help us hide state by creating an object with private data that no one else will be able to mutate:

- Here, **kind** and **wheelCount** are variables local to the factory function.

- Once the factory function runs, they can *never be accessed directly.*

- But the **car.startEngine()** method can get at them through a closure, to print them out.

```
function carFactory(kind) {
  var wheelCount = 4;
  var start = function() {
    console.log('started the ' +
                wheelCount + ' wheel ' +
                kind + '.');
  };

  return {
      startEngine: start
  };
}

var car = carFactory('Delorean');

car.startEngine();
// => started the 4 wheel Delorean.
```

The module pattern

- What if you knew you only wanted to make one car?
- You don't even really need the **carFactory**, you can just use an **IIFE**
- This is called the “module pattern”, because this block of code forms a module, with private data and a public interface

```
var car = (function (kind) {
  var wheelCount = 4;
  var start = function() {
    console.log('started the ' +
                wheelCount + ' wheel ' +
                kind + '.');
  };

  return {
      startEngine: start
  };
})('Delorean');

car.startEngine();
// => started the 4 wheel Delorean.
```

(quick aside: the window object)

- You may be familiar with the **window** object in the browser.
- The **window** object is known as the "global object"
- All properties on the window object are also global variable names, and vice versa:

```
var a = 5;
a;
// => 5
window.a;
// => 5

window.b = 10;
window.b;
// => 10
b
// => 10
```

Real-world examples of closures and the module pattern: 500px SDK, jQuery

Practice on your own: make an id generator (10 min)

- in the **id-generator-practice** folder

The flexibility and power of closures

- Unlike most other object-oriented languages, Javascript doesn't have a built-in concept of private variables.

- However, closures allow us to abstract away private variables and functions so that the public doesn't need to worry about the complicated details of how the program works

- In the module pattern example, we returned an object which has one method, which is the only way to access the private variables in the module

- There are many, many other examples of ways that closures — along with functions as first-class values — provide us with the flexibility and power to do most of what can be done in any other programming language.

Template demo: 500px revisited

Templates

- **Templating** lets us write a snippet of what's called an **HTML template**

- We can then use the template, along with some data that we've stored in a regular JS object, to render some HTML

- Then we can add that rendered HTML to the DOM

- There are many JavaScript templating libraries like Handlebars, Mustache, and Underscore templates. Today we will be working with Handlebars.

Handlebars: the steps

- Handlebars has a 5 step process for implementing templates in our applications:

1. **Include the Handlebarsjs library in your HTML file**
2. **Create the template** (as a script tag in your HTML file)
3. **Use Handlebars to compile the template** (into a function)
4. **Pass a data object to the compiled template** (the new function); this will produce some rendered HTML
5. **Add the newly rendered HTML to the DOM**

Handlebars: the steps

1. **Include the Handlebars library in your HTML file**

```
<script src="http://builds.handlebarsjs.com.s3.amazonaws.com/handlebars-v4.0.5.js"></script>
```

Handlebars: the steps

2. Create the template

- Our template comes in the form of a script tag,
- It has a reference id and a special type called "text/x-handlebars-template", that indicates it's a handlebars template.
- Inside the template, we surround the areas where we want to pass in data with "handlebar" moustaches:

```
<script id="hello-world-template" type="text/x-handlebars-template">
  <h1>{{helloTitle}}</h1>
  <p>{{helloContent}}</p>
</script>
```

Handlebars: the steps

3. Use Handlebars to compile the template

- We then get the html string for the newly created template,
- then pass that to **Handlebars.compile**,
- which returns a rendering function that we can use to pass data to:

```
var template = $('#hello-world-template').html();
var renderHelloHTML = Handlebars.compile(template);
```

Handlebars: the steps

4. Pass a data object to the new rendering function (the compiled template)

- The property names in the data object correspond to the words between the double brackets in the template.

```
var helloStatement = {
  helloTitle: "Hello World",
  helloContent: "How are you doing this fine day?"
};
var renderedHTML = renderHelloHTML(helloStatement);
```

Handlebars: the steps

5. Add the newly rendered HTML to the DOM

```
$('body').append(renderedHTML);
```

Handlebars: putting it all together

```
<!-- Step 1: include the library -->
<script src="http://builds.handlebarsjs.com.s3.amazonaws.co

<!-- Step 1: write the template -->
<script id="hello-world-template"
        type="text/x-handlebars-template">
  <h1>{{helloTitle}}</h1>
  <p>{{helloContent}}</p>
</script>
```

```
// Step 3: compile the template
var template = $('#hello-world-template').html();
var renderHelloHTML = Handlebars.compile(template);

// Step 4: render the HTML
var helloStatement = {
  helloTitle: "Hello World",
  helloContent: "How are you doing this fine day?"
};
var renderedHTML = renderHelloHTML(helloStatement);

// Step 5: add the rendered HTML to the DOM
$('body').append(renderedHTML);

// or you can also compress any of these steps:
// var template = $('#hello-world-template').html()
// $('body').append(Handlebars.compile(template)({
//   helloTitle: "Hello World",
//   helloContent: "How are you doing this fine day?"
// }));
```

Templates wrapup

- Templates make it very easy to organize our code in such a way that we keep the JS structures containing our data (called the "models") separate from all code that renders our data on the page (called the “views").

- This is very good practice, and makes our code cleaner and more manageable.

- You don't need templates to keep models separate from views — you can do it all in your JS code — but templates make it a lot more convenient.

Independent Practice: templates

Unit 2 project: Feedr (a feed reader)

- Instructions: http://bit.ly/jsdev2-feedr-project-instructions