Object Orientation in JavaScript

(https://github.com/learn-co-curriculum/phase-1-object-orientation-in-javascript) (https://github.com/learn-co-curriculum/phase-1-object-orientation-in-javascript/issues/new)

“Every application is a collection of code; the code’s arrangement is the design." - Sandi Metz

Learning Goals

- Review the meaning of Object Orientation
- Review the benefits of Object Orientation
- Introduce how JavaScript can be written using Object Orientation

Introduction

So far in JavaScript, we've discussed the "Three Pillars of Web Programming", **Recognizing JS events**, **Manipulating the DOM**, and **Communicate with the server**, and we saw how these pillars are incorporated into web applications.

The process we've learned so far has been:

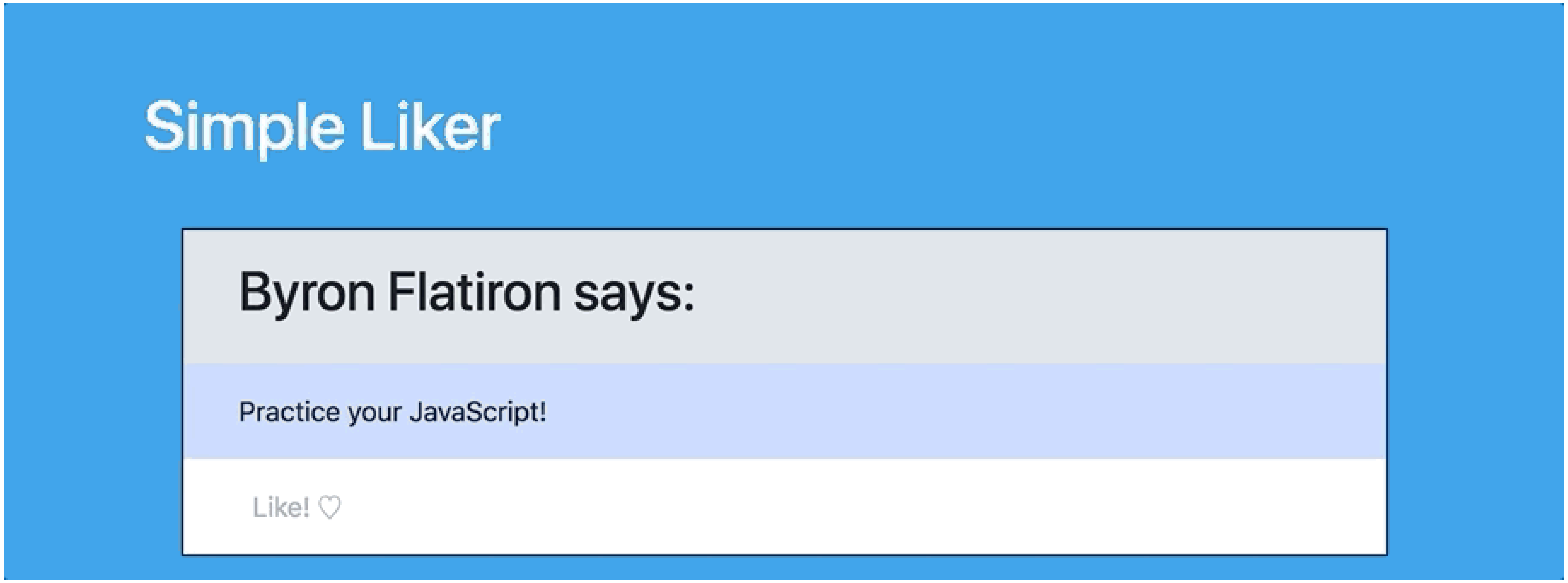
1. An HTML page renders on the screen
2. JavaScript is executed when the page loads
3. In the JavaScript, event listeners are created pointing to 'free-standing' functions, listed out in the JavaScript file.

In that process, we have three primary takeaways:

- Some of these functions handle DOM manipulation
- Some functions handle server communication
- Still other functions may serve as 'helper' functions to others

These functions can be invoked in whatever order we set, or from within each other. This set up works great when just getting off the ground; with a handful of functions, we can get an interactive website up and running. What happens, though, when we want to go beyond a small website?

Let's go back and take a look at the first app we discussed, **[Simple Liker](https://github.com/learn-co-curriculum/phase-1-stitching-together-the-three-pillars)** **(https://github.com/learn-co-curriculum/phase-1-stitching-together-the-three-pillars)**:



Imagine we want to add a feature to this - we want to display the number of users who have liked this message. Let's say we can get this info from the server, so we write a function that fetches the data and updates the DOM to display it. This function could be called when the page loads as well as whenever anyone clicks the 'like' button.

Great, but what happens when we start to expand further? Right now we only have the one message from Byron. What if we could have multiple messages to like? More functions would be needed. We'd probably have to modify the code that we have, as well, making it a more complicated.

Continuing to expand our app, what about displaying a list of names of people who liked this post? More code is needed. How about being able to post our *own* messages for other users to like? Even more code. What about adding the option to add message comments? A lot more code.

At this point, our Simple Liker would no longer be simple, and pretty soon, the code needed to keep all our features working has grown quite long. What's worse, all of the different things that happen on our website are starting to get jumbled together in a heap of functions. You may be able to get everything working for every feature added but the functions have become intertwined and dependent on each other. Coding this way comes at a cost:

The more complicated our code gets, the harder it is to understand and change it.

In larger applications, we might be dealing with hundreds of functions, DOM elements, and events, all tied together, forming a web of relationships. It becomes increasingly difficult to follow the flow of actions in your code as more and more functions are introduced. Changes to one function may have unforeseen affects on functions *'downstream'*.

This web of dependent, free-standing functions can be improved with organization. We do things like group functions together, arrange code in a readable way. Our options, however, are still limited - it's still a big web.

In this and the following lessons, we're going to take a look at an alternative way to structure our code, Object Orientation. By understanding and using OO, we can *design* our code to be easier to read, understand and change.

Define Object Orientation

With Object Orientation, instead of a web, we can think of our code as a collection of *cells*. These cells are separated from each other, can contain *information*, bits of data like variables, as well as *behaviors*, functions directly related to that data.

Consider the lessons so far: any sort of data we had to store and manipulate has been either stored in variables or passed to functions as arguments. Code like this:

```
let name = 'Evan';
let age = 34;

function sayHello(nameOfPerson) {
  console.log(`Hello, my name is ${nameOfPerson}.`);
}

function sayAge(age) {
  console.log(`I am ${age} years old.`);
}
```

```
function haveBirthday(age) {
  return age + 1;
}

sayHello(name);
// => Hello, my name is Evan.
sayAge(age);
// => I am 34 years old.
age = haveBirthday(age);
sayAge(age);
// => I am 35 years old.
```

Here, we've got some data, a `String` and an `Integer` assigned to variables, and a few functions for using and reading that data. The functions and data are closely related. Combined, they convey information about a *person*. While we, as human beings, might be able to interpret the above code as all related, as far as code structure, **there isn't anything structurally that actually *encodes* those relationships**. The *names* of our functions and variables are the only indicators that they are related.

Object Orientation grants us the ability to write code that **structurally** establishes the relationships between data and functions. If we rewrite our first code snippet applying Object Orientation, we could write something like this:

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  sayHello() {
    console.log(`Hello, my name is ${this.name}.`);
  }

  sayAge() {
    console.log(`I am ${this.age} years old.`);
  }

  haveBirthday(age) {
    console.log(`It's my birthday!`);
    this.age += 1;
  }
}

let evan = new Person('Evan', 34);

evan.sayHello();
// => Hello, my name is Evan.
evan.sayAge();
// => I am 34 years old.
evan.haveBirthday();
// => It's my birthday.
evan.sayAge();
// => I am 35 years old.
evan;
// => Person { name: 'Evan', age: 35 }
```

> You can try out the code above. Type `node` in your terminal, then copy and paste the above code into the Node environment to play around

Do not be alarmed if some of this looks unfamiliar. We will go into greater detail later about specific syntax. For now, though, take a moment to notice what changed. Where did our variables go? Now we just have one: `evan` . Instead of being assigned to a data type like a `String` or an `Integer` , `evan` assigned to a `Person` object that *contains* our data.

This `Person` object is a `class` , the fundamental structure underlying all Object Oriented code in JavaScript. The line:

```
let evan = new Person('Evan', 34);
```

...initializes a copy, a single ***instance*** of the `Person` `class` (more on this later). Two arguments are passed in, `'Evan'` and `34` , and the resulting instance is assigned to a variable.

The instance we just created contains the functions that were previously free-standing, `sayHello`, `sayAge`, and `haveBirthday`. Notice here that we're no longer passing data as arguments to these functions when we call them.

> For clarity, in these lessons, functions that are contained within a `class` or `class` instance will be referred to as 'methods'. The word *'function'* will be used when referring to functions outside of any `class` object.

Instead of receiving arguments, our *methods* have access to the data as `this.name` and `this.age`! These are referred to as *properties*, and are both assigned when a new instance of the `Person` class is created.

With our data and *methods* captured in the `Person` instance, we've ***encapsulated*** all the information and behaviors that represent a *person* in our code!

The Benefits of Object Orientation

The end result of our code example hasn't changed - we are still able to perform the same actions with minor modifications. However, by designing and creating `class`es like this, we can gain some important benefits:

Easier to Change

If we want to add more code related to a *person*, instead of just adding it in to a long list of functions, we now have a clear, logical place to put it. This makes it easier to extend code, and encourages good organization in the future.

Offers Better Data Control

By encapsulating our data in a `class`, we can protect that data from unexpected changes. When using local variables like `let name = "Evan"`, the variable is vulnerable to change from any function. Functions, meanwhile, will do what they are programmed to do, regardless of the data they are given. They are oblivious.

```
function sayHello(nameOfPerson) {
  console.log(`Hello, my name is ${nameOfPerson}.`);
}
```

```
sayHello([23, 'Golden Apple']);
// => Hello, my name is 23,Golden Apple.
```

In a `class` , however, we can design our code so that methods with a specific purpose only interact with and change the data they are supposed to. We don't need a stand alone `sayHello` function that takes in any argument and tries to log it, we just need it for saying a particular person's name.

```
let sarah = new Person('Sarah', 31);
sarah.sayHello();
// => Hello, my name is Sarah.
```

Equally, using a `class` to encapsulate our data allows us to be specific in how that data is used. The `sarah` variable in the code snippet above points to an entire *instance* of the `Person` class. Wherever `sarah` goes, `sarah` will always carry its properties and methods with it. If we need to access data stored on the instance, we can get property values directly:

```
sarah.name;
// => 'Sarah'
```

Or use the built in methods *we've* defined to access the information however we choose:

```
sarah.sayAge();
// => I am 31 years old.
```

Easy to Replicate

Every time we initialize a new instance of the `Person` class, we create a unique object:

```
let sarah = new Person('Sarah', 31);
let evan = new Person('Evan', 34);
```

Since each instance of `Person` has unique data, our instance methods will behave accordingly and only use *their* data:

```
evan.sayHello();
// => Hello, my name is Evan.
sarah.sayHello();
// => Hello, my name is Sarah.
```

This turns out to be a fantastic help when dealing with many collections of similar data. For example, comments on a blog post - while the data is unique to each, comments should always 'behave' the same way - they all display the same on the page. With Object Orientation, we can write a `Comment` `class` and create an 'instance' for each unique comment.

Once we establish the data and methods for a `class` , we can create as many copies as we need.

Improves Understanding by Adding Meaning

By encapsulating related information and behavior, we have organized our code in a more meaningful way. It *makes sense* to us as humans to group related things like this. Object Orientation works well for representing real world systems and relationships in code, which makes it easier to comprehend.

```
let sarah = new Person('Sarah', 31);
let evan = new Person('Evan', 34);
let restaurant = new Restaurant('La Villa', '261 5th Ave, Brooklyn, NY');

evan.sayHello();
// => Hello, my name is Evan.
sarah.sayHello();
// => Hello, my name is Sarah.

restaurant.addGuest(evan);
restaurant.addGuest(sarah);
restaurant.serveGuest(evan, new Drink('Water'));
restaurant.serveGuest(sarah, new Drink('Water'));
```

```
evan.saySmallTalk();
// => How about this weather?
```

All `class` es are structured in a consistent way, so even though we don't know the details of the `Restaurant` `class` , we can still infer how the `Restaurant` , `Person` and `Drink` `class` es might interact.

Many systems of inter-related things can be represented as objects interacting with one another, and being able to visualize these relationships helps when designing complex applications. Even the more abstract concepts of web programming can be easier to understand using Object Orientation, though it takes some practice to think in the OO mindset.

Conclusion

There is a lot involved in writing Object Oriented code. In the upcoming lessons, we will build `class` es from scratch and talk about the different ways we can design them. Later on, we will introduce `class` interactions and how many instances of different `class` es can work together. Object Orientation can fundamentally change the way we design our code, but as we explore, we will see how proper design encourages code that is easier to use, understand, change and maintain.

Resources

- **MDN - Object Oriented JavaScript for Beginners (https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object-oriented_JS)**