Before we begin...

- Videos On!

Welcome

Agenda

- Review
- Methods
- Functional JavaScript
- `this`
- Classes
- Inheritance
    - Classical vs. Prototypal

Review

- Pseudocode
- Advanced Functions
  - Callbacks
  - Scope and Hoisting
  - Closures
  - Higher Order Functions
  - Rest Parameters
  - Spread Operators

Functional Array Methods

Imperative vs. Declarative

An Imperative approach to programming

Describes the "HOW". You explain every single thing in the program (e.g. `for (...) {}`)

A Declarative approach to programming

Describes the "WHAT". You describe a pattern (e.g. `.forEach`)

Declarative Programming

- **Declarative Programming** leads to:
  - More readable code
  - Often more efficient code
- You'll spend less time trying to understand your program and more time figuring out the higher-level logic
- Declarative programming *uses the magic and hides the complexity*

[].forEach

`.forEach`

- The `.forEach` method allows us to iterate through each item in a collection
- We provide a callback function
    - That callback will automatically receive the current item, the index and the entire collection

.forEach

```
let letters = ["a", "b", "c", "d", "e"];

function logLetter(letter, index) {
  let message = "Current Letter: " + letter + ". Index: " + index;
  console.log(message);
}

letters.forEach(logLetter);
```

[].filter

`.filter`

- The `.filter` method allows us to iterate through each item in a collection
- It will return a new collection
- We provide a callback function
  - It'll receive the current item, the index and the array itself
  - If the callback function returns true, the item will be stored in the returned collection. Otherwise, it will be removed

.filter

```javascript
let numbers = [1, 2, 3, 4, 5, 6];

function isEven(num) {
  return num % 2 === 0;
}

let evens = numbers.filter(isEven);

console.log(evens);
```

```
[].map
```

.map

- The .map method allows us to iterate through each item in an array and allow us to transform them
- It will return a new collection
- We provide a callback function
  - It will be provided with the current item, the current index and the entire collection
  - The callback must return a value! The value that you return will be stored in the new collection
  - Essentially it transforms each item!

.map

```javascript
let letters = ["a", "b", "c", "d", "e"];

function uppercaseLetter(letter) {
  return letter.toUpperCase();
}

let upperCased = letters.map(uppercaseLetter);

console.log(upperCased);
```

.map

```
let numbers = [1, 2, 3, 4, 5, 6];

function timesByFive(num) {
  return num * 5;
}

let multiplied = numbers.map(timesByFive);

console.log(multiplied);
```

[].reduce

`.reduce`

- The `.reduce` method iterates through a collection and returns a single value
- We provide a callback function
  - It will be provided with the running total and the current value, as well as a starting value
  - The callback must return a value! The value that you return will be stored as the running total value for the next iteration
  - It's also known as `inject`
- Think of it as reducing an array down to a single value

.reduce

```
let nums = [1, 2, 3, 4, 5, 6];

function addNumbersTogether(currentTotal, currentNumber) {
  let newTotal = currentTotal + currentNumber;
  return newTotal;
}

let total = nums.reduce(addNumbersTogether, 0);

console.log(total);
```

Methods

Methods

- When we create data types, we automatically get:
  - Properties - Static pieces of information about the data
  - Methods - Operations we can perform on the data
- We can create our own on Objects!
- This will help group functionality and organise code

Methods

```
const explorer = {
  firstName: "Jacques",
  lastName: "Cousteau",
  travel: function () {
    console.log("Always!");
  },
  speak: function () {
    console.log("Hi there!");
  },
};

explorer.travel();
explorer.speak();
```

this

What is `this`?

- One of the most confusing mechanisms in JavaScript
- A special identifier that's automatically defined for us
- It can seem downright magical but it aims to represent the current context
  - It's JavaScript's way of telling us what it thinks we care about (e.g. if it is a method, it'll refer to the data that that method was called upon)

Let's get `this` over with

The naming makes it difficult to talk about

So, how does `this` work?

- It all comes back to the call-site
- To understand how the this keyword works, we need to know exactly where and how the function was called (and by who)
  - There are more ways than we have seen so far!
- Every function, when it is running, has access to its current execution context

`this` exists so we can:

- Reuse functions with different contexts
- Change the focus of our code
- Make methods more dynamic
- We don't always know what we are talking about!
  - e.g. Maybe we have a function creating objects for us, or maybe we don't know which element is being interacted with

The Call-Site

Knowing that this represents the context of whatever code is running, there are five main ways of it being automatically defined for us:

- Global Binding (`window`)
- Event Binding
- Implicit Binding
- Explicit Binding
- `new` Binding

The Global binding - `window`

This is the default binding. In websites, the global binding will always be the `window` object

```
console.log(this);

function checkThisOut() {
  console.log(this);
}

checkThisOut();
```

The Event binding

When an event listener is run, the `this` keyword refers to the target of the event (e.g. the element that was clicked)

```
let img = document.querySelector("img");

function onImageClick() {
  console.log(this);
}

img.addEventListener("click", onImageClick);
```

The Implicit binding

When you run a method, the `this` keyword will refer to the containing object

```
let person = {
  name: "Groucho",
  speak: function () {
    console.log(this, this.name);
  },
};

person.speak();
```

The Implicit binding

```
const obj = {
  name: "Outer",
  inner: {
    name: "Inner",
    innerLogThis: function () {
      console.log(this, this.name);
    },
  },
  outerLogThis: function () {
    console.log(this, this.name);
  },
};
```

The Explicit binding

When you use `.call`, the `this` keyword refers to the parameter you provide

```
function sayHello() {
  console.log("Hello, " + this.name);
}

let person = { name: "Zeppo" };

sayHello.call(person);

// This explicitly sets the `this` keyword to person
```

The Explicit binding

When you use `.apply`, the `this` keyword refers to the parameter you provide

```javascript
function sayHello() {
  console.log("Hello, " + this.name);
}

let person = { name: "Zeppo" };

sayHello.apply(person);

// This explicitly sets the `this` keyword to person
```

The Explicit binding

When you use `.bind`, the `this` keyword refers to the parameter you provide

```
function sayHello() {
  console.log("Hello, " + this.name);
}

let person = { name: "Zeppo" };

let personsHello = sayHello.bind(person);
personsHello();

// This explicitly sets the `this` keyword to person
```

The `new` binding

When you use `new`, the `this` keyword refers to a new empty object that you can add properties to (it's also automatically returned for you)

```
class Person {
  constructor() {
    console.log(this);
  }
  speak() {
    console.log(this);
  }
}

let p = new Person();
p.speak();
```

Determining `this`

The order of precedence:

- Is the function called with the `new` keyword?
- Is the function called with `.call`, `.apply` or `.bind`?
- Is the function run by an event listener?
- Is the function called on an object (is it a method)?
- Otherwise, it is the default binding - the window object*

`this` Resources

- Tyler McGinniss: WTF is `this?`
- Todd Motto: `this`
- MDN: `this`
- Kyle Simpson: `this` and Object Prototypes
- JavaScript is Sexy: `this`
- Rachel Ralston: `this`
- Quirks Mode: `this`

Classes

Why do we need `class`?

- Relatively regularly, we need to create many objects of the same kind (e.g. in an app, we may need Users, Posts etc.)
- Classes are a way for us to model data with JavaScript
  - They encapsulate data and functionality together
- Think of them as blueprints
  - We create "instances" of a `class`
- They are just fancy functions though

How do we use `class`es?

- We define the blueprint
- We create instances

The `class` syntax

```
class MyClass {}

class Person {
  constructor() {
    console.log("A person was born!");
  }
}

class Post {
  constructor() {
    console.log("A post was written!");
  }
  edit() {}
  save() {}
}
```

The `class` syntax

```
class Person {
  constructor() {
    console.log("A person was born!");
  }
}

let p = new Person();
```

The `class` syntax

```
class Person {
  constructor(name) {
    this.name = name;
  }
  print() {
    console.log(this);
  }
}

let p = new Person("Douglas");
p.print();
```

Inheritance

- Inheritance is one way for classes to extend other classes
  - Instances of the child class will have access to the parent's class data and functionality
- This allows us to model complex systems

The `class` syntax

```javascript
class Shape {
  constructor(type) {
    this.type = type;
    console.log(type + " was created!");
  }
}

class Rectangle extends Shape {
  constructor(width, height) {
    super("Rectangle"); // Call the parent class' constructor
    this.width = width;
    this.height = height;
  }
  getArea() {
    return this.width * this.height;
  }
}
```

`class` Resources

- JavaScript.info: Class
- MDN: Classes
- Exploring JS: Classes
- Codecademy: Classes
- Mastering JS: Class

That's all for tonight!

Homework

- Read JavaScript.info's `class` page and `class` inheritance page
- Read MDN's Class page

Homework (if you are on Mac)

- Install iTerm2
- Install Node
- Install Git using Homebrew
- Check that everything that has been installed correctly:
  - Restart the iTerm2 app
  - Run `node -v`
  - Run `npm -v`
  - Run `git --version`

Homework (if you are on Windows)

- Install Git For Windows
- Install Node
- Check that everything that has been installed correctly:
    - Restart the Git for Windows app
    - Run `node -v`
    - Run `npm -v`
    - Run `git --version`

What's Next?

- Terminal
- Git and GitHub
- Asynchronous vs. Synchronous Programming
- Promises
- A.P.I.s
- A.J.A.X.

Thank You!