A PDF Reference for

The Modern JavaScript Bootcamp

Version 1.1

Taught by Andrew Mead

Table of Contents

Section 1: Course Overview

In this first section, we're going to explore what you'll learn in this course. You'll learn why JavaScript is a great language to know, and you'll explore what's covered in the class.

There are no individual lecture notes for this first section. These lectures are important, but they don't cover any JavaScript features.

Section 2: Setting up Your Machine

Lesson 1: Section Intro

In this section, you'll be setting up your machine for the rest of the course. There are a few things we have to install, and then we'll start learning how to create JavaScript applications!

Lesson 2: Installing Visual Studio Code

In this lecture, you'll install Visual Studio Code. This is the text editor that we'll be using throughout the course.

Software Installed:

- Visual Studio Code (text editor)

Lesson 3: Installing Node.js

In this video, you'll install Node.js. This is going to allow us to run JavaScript files on your machine.

Software Installed:

- Node.js

Lesson 4: [Windows Only] Install cmder

This lecture is for Windows users only. You'll be installing cmder, a console emulator for Windows that'll make working in the terminal a whole lot easier.

Software Installed:

- cmder (console emulator)

Lesson 5: Introduction to the Terminal

In this video, you'll learn how to use the terminal. We'll cover some basic commands that'll make it possible to run JavaScript files from our machines.

Common Commands

pwd - Print working directory

```
# Print the full path to the current working directory
pwd
```

clear - Clear the terminal output

```
# Clear the terminal output
clear
```

ls - List the contents of a directory

```
# List the contents of the working directory
ls
```

cd - Change directory

```
# Change directories to home
cd ~


# Navigate up a directory then into a nested directory
cd ../Downloads
```

Lesson 6: Hello JavaScript!

In this lecture, you'll create and run your first JavaScript script! We're not going to dive into any complex JavaScript just yet, but this will verify that everything's been installed correctly.

Running a File Though Node

```
# Run a JavaScript file though Node.js
node script.js
```

Section 3. JavaScript Basics - Variables and Statements

Lesson 1: Section Intro

In this section, you're going to learn the very basics of JavaScript. We'll explore the JavaScript syntax and some of the core building blocks that JavaScript provides. By the end of this section you'll be able to create basic JavaScript programs!

Lesson 2: Strings and Variables

In this lecture, you're going to start learning JavaScript! We'll kick things off by talking about two important things, variables and strings.

Here's an example where we use variables and strings together:

```
// Create a variable to store the city
let city = 'Philadelphia'

// Create a variable to store the state
let state = 'Pennsylvania'

// Create a variable to combine the city and state
let location = city + ', ' + state

// Print the combined location to the terminal
console.log(location)
```

Documentation Links

- The let statement
- Strings

Lesson 3: Numbers

In this lesson, you'll learn about numbers in JavaScript. This includes creating numbers and performing basic arithmetic operations like addition, subtraction, multiplication, and division.

Here's an example of each:

```
// Create a number and assign to a variable
let age = 26

// Add
let addResult = age + 1   // 27

// Subtract
let subtractResult = age - 26.5   // -0.5

// Multiply
let multiplyResult = age * 10   // 260

// Divide
let divideResult = age / 2   // 13

// Parenthesis can be used to make sure one operation happens before another
let result = (age + 3) * 7   // 203
```

Comments

JavaScript comments allow you to document your code. The content of your comments is for humans only, which means it doesn't follow any special syntax rules. Everything after `//` is part of the comment.

Here's an example:

```
// A comment on its own line

console.log('Welcome!') // A comment at the end of a line
```

Documentation Links

- Arithmetic operators
- Numbers

Lesson 4: More on Variables

In this lesson, you'll learn a bit more about variables. We'll talk about the rules for variable naming and I'll show you a few common traps you'll want to avoid.

Variable naming rules

A JavaScript identifier must start with a letter, underscore (_), or dollar sign ($); subsequent characters can also be digits (0-9). Because JavaScript is case sensitive, letters include the characters "A" through "Z" (uppercase) and the characters "a" through "z" (lowercase).

- From the MDN docs

Lesson 5: Build a Temperature Converter

In this lesson, you'll be writing an application from scratch. The application is a temperature converter that converts Fahrenheit to Celsius and Kelvin.

There are no notes for this lesson, as no new language features were explored.

Lesson 6: Booleans and Comparison Operators

In this lesson, you'll learn about our third JavaScript type, the Boolean. You'll also learn how you can compare two things such as two strings or two numbers.

Comparison Operators

There are six main comparison operators. The code below contains an example of each.

```
// === equal
'Andrew' === 'Andrew' // True
'Mike' === 'mike' // False

// !== not equal
'Andrew' !== 32 // True
12 !== 12 // False

// < less than
3 < 40 // True
3 < 3 // False

// <= less than or equal to
// 3 <= 3 // True
// 0 <= -23 // False

// > greater than
16 > -16 // True
-16 > 16 // False

// >= greater than or equal to
-23 >= -40 // True
100 => -100 // False
```

Documentation Links

- Boolean
- Comparison Operators

Lesson 7: If Statements

In this video, you'll learn how to use if statements. These will let you conditionally run JavaScript code.

Example of a plain if statement:

```
let ticketStatus = 'closed'

if (ticket === 'closed') {
    console.log('This support ticket has been closed')
}
```

Documentation Links

- If statement

Lesson 8: Advanced If Statements

In this lesson, we're going to continue looking at the if statement. We know about "if" and "else", but there's also an "else if" available to us.

Else

Here's an example of an if statement with an else clause attached.

```
let age = 18

if (age >= 18) {
    console.log('You are an adult')
} else {
    console.log('You are not an adult')
}
```

Else if

Else if allows us to add another conditional check to our if statement. Here's an example of an if statement with an else clause and else if clause attached.

```
let age = 26
let isChild = age <= 7
let isSenior = age >= 65

if (isChild) {
    console.log('Welcome! You are free.')
} else if (isSenior) {
    console.log('Welcome! You get a discount.')
} else {
    console.log('Welcome!')
}
```

Logic Not Operator

The logic not operator allows you to flip a Boolean value. You can convert true to false, and false to true.

```
let isAccountActive = false

// This condition will pass if isAccountActive is false
if (!isAccountActive) {
    console.log('Account is locked')
}
```

Documentation Links

- Logical Not Operator

Lesson 9: Logical “And” and “Or” Operators

In this lecture, you’re going to learn about the logical “and” and “or” operators. These operators allow you to set up more complex conditional logic involving two or more comparisons.

Logical And

The code below uses the logical “and” operator (`&&`) to check that the temp value is greater than or equal to 70, and that it’s less than or equal to 90. This allows us to run some code when the temperature is between 70 degrees and 90 degrees.

```
let temp = 170

if (temp >= 70 && temp <= 90) {
    console.log('Get outside!')
} else {
    console.log('Stay inside')
}
```

Logical OR

The code below uses the logical “or” operator (`||`) to check if either of the guests are vegan. If you were to convert that condition to English, it would read “Run this code if the first guest is vegan or if the second guest is vegan. It’s fine if both are vegan, just make sure at least one of them is.

```
let isGuestOneVegan = true
let isGuestTwoVegan = true

if (isGuestOneVegan && isGuestTwoVegan) {
    console.log('Only offer up vegan food.')
} else if (isGuestOneVegan || isGuestTwoVegan) {
    console.log('Make sure to offer up some vegan food.')
} else {
    console.log('Offer up anything on the menu.')
}
```

Documentation Links

- Logical “And” and “Or” Operator

Lesson 10: Variable Scope: Part I

In this lesson, we’re going to explore variable scope in JavaScript. Variable scope determines where in your program a given variable is accessible.

Variable Scope

Not all programming languages use the same scoping rules. JavaScript uses lexical scoping, also known as static scoping. This is the idea that variables created in one part of a program might not be accessible in another part. The context in which the variable was created plays a role in where it can be accessed.

When looking at JavaScript code, it’s the code blocks that we want to keep an eye out for. This includes our functions and if statements where you use “{” and “}” to group together line of JavaScript code. Code blocks let us define a new scope.

There are two types of scope:

1. Global scope - This is the scope that’s visible to all other scopes. It contains variables defined outside of any code block. These variables are accessible anywhere in the program.
2. Local scope - This is a scope created by a code block. A local scope can access values defined in itself or in any parent/ancestor scope. It’s unable to access variables in a child scope.

Here’s an example script along with its scope hierarchy.

```
// Global scope (name, getName)
  // Local scope (age)
    // Local scope (location)
  // Local scope (height)

let name = 'Andrew'
let getName = function () {
    console.log(name)
    let age = 26

    if (2 > 0) {
        console.log(name)
        console.log(age)
        let location = 'Philadelphia'
    }
}

getName()
console.log(age)

if (1 < 2) {
    console.log(name)
    let height = 72
    getName()
}
```

Lesson 11: Variable Scope: Part II

In this lesson, we're going to talk about two more scope related topics. The first is shadowing. The second is leaked globals.

Shadowing

We know we can define the same variable twice, but that rule only applies if the two variables are defined in the same scope. It's perfectly fine to define the same variable in two different scopes. Below, we have `score` in the global scope and in our local function scope.

What happens when we try to reference one of those values? JavaScript starts in the current scope. It looks for the variable. If JavaScript finds the variable then the variable's value is used; however, if it doesn't find it, it goes to the parent scope and tries again. It'll go all the way up to the global scope looking for the variable.

Shadowing occurs when a local variable shadows over a global variable and takes precedent. In the example below, the local `score` variable shadows the global `score`

variable when we create `` `Score: ${score}` `` and `2` is used. The value `5` is used for the log call on the last line of that file.

```
let score = 5
let getNewScoreMessage = function () {
    let score = 2
    return `Score: ${score}`
}

let result = getNewScoreMessage()
console.log(result)
console.log(score)
```

Leaked Globals

With shadowing, we learned that JavaScript will go all the way up to the global scope when trying to find a variable's value. What happens if we're trying to set a variable's value? This same process happens. We go all the way up to the global scope trying to find the variable's value we want to set. The only catch is that if we get to the global scope and don't find it, JavaScript just creates it as a global variable. This means we might think a variable is in the local scope when in reality it's been leaked into the global scope which can cause hard-to-track-down bugs.

To avoid leaked globals, make sure to define a variable you're going to assign a value to. The below example shows both situations.

```
// Example 1: Leaked global

let getNewScoreMessage = function () {
    if (1 < 2) {
        score = 3
    }

    console.log(score)
}
getNewScoreMessage()
console.log(score) // Prints 3


// Example 2: No leaked global

let getNewScoreMessage = function () {
    let score

    if (1 < 2) {
        score = 3
    }

    console.log(score)
}
getNewScoreMessage()
console.log(score) // Will throw an error as score wasn't leaked to the
global scope
```

Section 4. Functions

Lesson 1: Section Intro

In this section, you're going to learn the very basics of JavaScript. We'll explore the JavaScript syntax and some of the core building blocks that JavaScript provides. By the end of this section, you'll be able to create basic JavaScript programs!

Lesson 2: Function Basics

In this video, you're going to create and run your first JavaScript function! Functions are like programs in your programs. It's a piece of code we can run whenever we want.

Functions

There are three important pieces to creating a function:

1. Arguments - The arguments are the values passed into a function. If I create a function that validates an email, I'd need to accept the email as a function argument.
2. Function Code - The function code is what goes inside of the curly braces. This is where you define what your function does.
3. Return Value - This is part of the function code. It lets you define what value to send back to the function caller.

In the below example, we have a single argument for the temperature in Fahrenheit. The function code then converts the argument from Fahrenheit to Celsius. Finally, the function returns the calculated value so the called can do something with it.

```
let fahrenheitToCelsius = function (fahrenheit) {
    const celsius = (fahrenheit - 32) * (5 / 9)
    return celsius
}

let temp70 = fahrenheitToCelsius(70)
console.log(temp70)
let temp32 = fahrenheitToCelsius(32)
console.log(temp32)
```

Documentation Links

- Functions

Lesson 3: Undefined and Null

In this video, you're going to learn about two new types, undefined and null.

Undefined

Undefined is often used by JavaScript itself.

First up, if you create a variable without assigning a value, JavaScript will assign it the value of `undefined`. For example:

```
let name
console.log(name) // Will print undefined
```

We also know that function arguments are set to `undefined` if they're not passed in when called. For example:

```
let double = function (x) {
    if (x === undefined) {
        console.log('Please provide x') // This will print
    } else {
        console.log(x)
    }
}

double()
```

Null

Unlike undefined, null is something we typically assign to a value. This is a way to clear a variable's value as shown below.

```
let name = 'Andrew'

name = null

if (name === null) {
    console.log('No name is set!')
}
```

Documentation Links

- Undefined
- Null

Lesson 4: Multiple Arguments and Argument Defaults

In this video, we're going to take a deeper dive into function arguments. You'll learn how to set up multiple arguments and come up with default argument values if none is provided.

Multiple Arguments

There are two ways to set up multiple arguments. We have the code that defines the function, and we have the code that calls the function. In both cases, we use a comma separated list. For the function definition, we name the arguments and separate them by a comma. For the function call, we provide the values separated by a comma.

```
let add = function (x, y, z) {
    return x + y + z
}

let result = add(1, 12, 3)
console.log(result)
```

Argument Defaults

JavaScript does not provide a way to mark a function argument as required. As we've already seen, we can enforce this ourselves with if statements, but we can also set up default values if none is provided.

```
let getScoreText = function (name = 'Anonymous', score = 0) {
    return `${name} Score: ${score}`
}

let text = getScoreText(undefined, 11)
console.log(text)
```

Documentation Links

- Argument defaults

Lesson 5: Function Scope

In this lesson, you'll explore how scope relates to our functions. The scoping rules we learned in the last section still apply!

Function Scope

The scoping rules covered in the last section still apply. Just like if statements, functions also create a new local scope. This contains the variables defined in the function body as well as the function arguments.

Here's an example script along with its scope hierarchy.

```
// Global scope (add)
  // Local scope (a, b, result)

let add = function (a, b) {
  let result = a + b
  return result
}

console.log(add(2, 4))
// Unable to access a, b, or result. They are not in scope in the global
scope.
```

Lesson 6: Template Strings

In this lesson, we'll cover the template string syntax. This will allow you to create a string using other strings, numbers, and Booleans.

Template Strings

Here's an example that shows two ways to get the job done. The `bio` variable is created using regular strings and concatenation. The `altBio` variable is created using template strings.

Example:

```
let petName = 'Hal'
let petAge = 3

let bio = petName + ' is ' + petAge + ' years old.'
console.log(bio)

let altBio = `${petName} is ${petAge} years old.`
console.log(altBio)
```

Documentation Links

- Template strings

Lesson 7: Build a Grade Calculator

In this lesson, it's up to you to build a grade calculator. Give the student's score and the total number of points; you're going to generate a message that uses the letter grade.

There are no notes for this lesson, as no new language features were explored.

Section 5. Objects

Lesson 1: Section Intro

In this section, you're going to learn about objects in JavaScript. Objects allow us to store related information in a single place. This is useful when dealing with something like a user, a book, or a car. We can store all the properties of the book such as the title, author, and number of pages in a single place.

Lesson 2: Object Basics

In this video, you're going to learn the basics of JavaScript objects. You'll learn how to create a new object and how to read and modify your objects.

Creating Objects

You can create a new object by opening and closing some curly braces. You define the object properties inside those curly braces. Below is an example of an object with three properties; title, author, and page count.

```
let myBook = {
    title: '1984',
    author: 'George Orwell',
    pageCount: 326
}
```

Dot Notation for Object Properties

Dot notation allows you to access object properties. This is useful if you want to read or change a property value. Here's an example using dot notation for both reading and updating.

```
let myBook = {
    title: '1984',
    author: 'George Orwell',
    pageCount: 326
}

// Reading a property value with dot notation
console.log(`${myBook.title} by ${myBook.author}`)

// Chaing an object property value using dot notation
myBook.title = 'Animal Farm'
```

Documentation Links

- Objects

Lesson 3: Objects with Functions

In this lesson, you're going to explore how you can use objects with functions. This includes calling functions with object arguments and returning objects from functions.

Objects as Function Arguments

As we already know, objects allow us to store related data in a single place. When calling functions, we can pass an object as an argument. This lets you pass all that related data to a function without having to pass in many individual arguments. All you need is the single object argument as shown below:

```
let myBook = {
    title: '1984',
    author: 'George Orwell',
    pageCount: 326
}

let otherBook = {
    title: 'A Peoples History',
    author: 'Howard Zinn',
    pageCount: 723
}

let getSummary = function (book) {
    console.log(`${book.title} by ${book.author}`)
}

getSummary(myBook)
```

Objects as Function Return Values

JavaScript functions can return a single value. This can be a problem if you have a function where you want to return multiple values. The solution is to return an object. The object you return can have as many properties as you like.

As shown in the example below, we're able to return three different numbers from our function when we return an object and add them as object properties:

```
let convertFahrenheit = function (fahrenheit) {
    return {
        fahrenheit: fahrenheit,
        kelvin: (fahrenheit + 459.67) * (5 / 9),
        celsius: (fahrenheit - 32) * (5 / 9)
    }
}

let temps = convertFahrenheit(74)
console.log(temps)
```

Lesson 4: Object References

In this video, we're going to explore what happens when we pass object around our program.

Object References

Imagine we have an object and we pass that object into a function as an argument. Because we passed the object in, we can now access it in the function body. What I want to explore is what happens if we change an object property in the function. Does it change the object in the function, the object passed into the function, or both?

The answer is both. This is because objects are passed by reference. In the example below, any change made to `account` in `addExpense` will also be reflected in `myAccount`. Both `account` and `myAccount` are references to the same object in memory.

```
// Both logs print the same exact thing
let myAccount = {
    name: 'Andrew Mead',
    expenses: 0,
    income: 0
}

let addExpense = function (account, amount) {
    account.expenses = account.expenses + amount
    console.log(account)
}

addExpense(myAccount, 2000)
console.log(myAccount)
```

While property changes will be reflected in `myAccount`, this relationship will be broken if we try to assign a new value to `account` like so:

```
// Both logs print differen things
let myAccount = {
    name: 'Andrew Mead',
    expenses: 0,
    income: 0
}

let addExpense = function (account, amount) {
    account = {}
    account.age = 1
    console.log(account)
}

addExpense(myAccount, 2000)
console.log(myAccount)
```

Lesson 5: Build an Expense Tracker

In this video, you'll be writing an expense tracker application using what you've learned about JavaScript so far.

There are no notes for this lesson, as no new language features were explored.

Lesson 6: Methods

In this video, you're going to learn about methods. A method is nothing more than an object property with a value set to a function.

Method Example

Below is an example. We have a property on our object named `checkAvailability`. Its value is a function. That makes `checkAvailability` a method.

Methods are still functions. They can take arguments, do some work, and return a value. We call them by accessing the property via dot notation and adding our parentheses as well as any arguments.

```
let restaurant = {
    name: 'ASB',
    guestCapacity: 75,
    guestCount: 0,
    checkAvailability: function (partySize) {
        let seatsLeft = this.guestCapacity - this.guestCount
        return partySize <= seatsLeft
    }
}

console.log(restaurant.checkAvailability(4))
```

This keyword

Our methods get access to a special `this` keyword. Methods live on an object. The `this` keyword is a reference to that object. In the example above, we are able to read the object properties from `checkAvailability` using dot notation via `this.guestCapacity` and `this.guestCount`.

Documentation Links

- This Keyword

Lesson 7: Exploring String Methods

In this video, we're going to turn our attention from methods we've written to methods that are built into the language. The focus will be on methods for our strings.

Methods

Below is a small sample of the methods we have access to with our strings. These allow us to do things we otherwise wouldn't be able to do like check the length of a string or search one string for a set of characters.

```
let name = '  Andrew Mead '

// Length property
console.log(name.length)

// Convert to upper case
console.log(name.toUpperCase())

// Convert to lower case
console.log(name.toLowerCase())

let password = 'abc123asdf098'

// Includes method
console.log(password.includes('password'))
```

Documentation Links

- String Methods

Lesson 8: Exploring Number Methods

In this video, you're going to explore more of the built-in methods we have access to in JavaScript. Our focus will be on numbers.

Methods

While numbers also have methods, there are fewer and they are used less frequently.

Numbers can be used with `Math`. This opens up many more operations such as rounding numbers or generating random numers. I've included a few examples below.

```
let num = 103.941

// Specify decimal points
console.log(num.toFixed(2))

// Round numbers
console.log(Math.round(num))
console.log(Math.floor(num))
console.log(Math.ceil(num))

// Generate a random number
let min = 0
let max = 1
let randomNum = Math.floor(Math.random() * (max - min + 1)) + min
```

Documentation Links

- Number Methods
- Math Object

Lesson 9: Constant Variables

In this video, you're going to learn about an alternative way to create variables. This will allow you to better describe what your program is doing.

Const

When we create a variable with `let`, we can reassign its value later on in our script. This is great, but what about the variables that get initialized and never have their values changed? For those, we have `const`.

Variables created with `const` cannot be reassigned. That means the following program will fail since we're trying to change the value of `isRaining`.

```
const isRaining = true

isRaining = false

console.log(isRaining)
```

This allows us to create a program that's much easier to reason about. When we see `const` we know the program is creating a variable and that the variable's value won't ever change. When we see a `let` we know the program is creating a variable and that the

variable's value will change. This difference is subtle, but when we end up with larger program it's nice to get as much information about what's going on as possible.

Documentation Links

- Const

Lesson 10: Bonus: Variables with Var

In this video, you're going to learn about a third way to create variables in JavaScript. While this isn't a tool I'd recommend using anymore, it's important to understand, as you'll likely see it out in the real world.

Var

Creating a variable with `var` is identical to creating a variable with `const` or `let`. Like with `let`, variables created with `var` can also be reassigned.

```
var name = 'Andrew'
name = 'Vikram'
console.log(name)
```

There are some import differences between `var` and `const`/`let`. These quirks and flaws are why it's no longer recommended to use `var` and why it's not used throughout the course.

Function Scope

The first difference is that variables created with `var` are function scoped. This means that code blocks created by if statements or for loops won't create a new scope.

This example shows that a variable created with `var` can be accessed outside of the if statement where it was declared. This was not the case for `const` or `let`.

```
if (true) {
    var name = 'Andrew'
    let age = 26
}

console.log(name) // Will print: Andrew
console.log(age) // Will print: ReferenceError
```

This example shows that variables created with `var` are function scoped.

```
const setupVariables = () => {
    var name = 'Andrew'
    let age = 26
}

setupVariables()
console.log(name) // Will print: ReferenceError
console.log(age) // Will print: ReferenceError
```

Redeclaring Variables

Another difference is that variables created with `var` can be redeclared. This is a behavior can cause some interesting bugs where a variable is accidentally overridden and the program doesn't complain.

The example below shows this redeclaration behavior. If the same code was run with `let` or `const`, it would generate a syntax error.

```
var name = 'Andrew'
var name = 'Mike'
console.log(name) // Will print: Mike
```

Hoisting

Last up, variables created with `var` can be accessed before they're declared. This hoisting of variable declarations can cause some unexpected (or at least unintuitive) behaviors.

```
age = 6
console.log(age) // Will print: 6
var age
```

The problem above will execute and print `6` without throwing any errors. That's because var-based variable declarations are hoisted to the top of the scope. The code above runs as if it was the code below.

```
var age
age = 6
console.log(age) // Will print: 6
```

Documentation Links

- Var

Section 6. Arrays

Lesson 1: Section Intro

In this section, you're going to explore arrays in JavaScript. Arrays allow us to store a list of items. This could be a list of strings, numbers, objects, or any other type in JavaScript.

Arrays are essential to software development. Think about an email app. It's a list of conversations, and each conversation has a list of messages. What about accounting software? It's a list of clients, a list of expense, and a list of income.

Lists are everywhere, and getting comfortable with them is going to be important if the goal is to create useful applications.

Lesson 2: Array Basics

In this video, you're going to learn the basics of JavaScript arrays. You'll learn how to create new arrays and how to read items from your arrays.

Creating Arrays

We create an array by opening and closing a set of square brackets. We can choose to start off with an empty array, or we can initialize it with some items. To initialize it with some items, all you need to do is provide a comma separated list of values.

```
// Empty array
const emptyArray = []

// Array of numbers
const expenses = [100.10, 45, -20]

// Array of mixed types
const data = [true, 12, 'Andrew']
```

Reading Array Values

We can read values from arrays using bracket notation. This allows us to get an item by its position. This position is known as an index and it starts at `0`. So the index for the first item is `0` and the index for the third item is `2`.

```
const names = ['Andrew', 'Vikram', 'Zhu']

// Get the first item
console.log(names[0])

// Get the last item using bracket notation and the length property
console.log(names[names.length - 1])
```

Documentation Links

- Arrays
- Length Property

Lesson 3: Manipulating Arrays with Methods

In this video, you're going to learn how to manipulate an existing array. This includes adding, removing, alerting, and replacing array items.

Adding Items to an Array

We can add new items onto an array using the `push` and `unshift` methods. The `push` method allows us to add a new item onto the end. The `unshift` method allows us to add a new item onto the beginning.

```
const nums = [1]

nums.push(12)
nums.unshift(3)

console.log(nums) // Will print [3, 1, 12]
```

We can add items anywhere using `splice`. As we saw in the video, `splice` allows us to delete, add, or replace any number of items. The second argument is the number of items to delete. Leave that at `0` and you'll be adding items without removing anything.

```
const nums = [99, 199, 299]

// Add a new item at the second position without deleting any existing items
nums.splice(1, 0, 399)

console.log(nums) // Will print [99, 399, 199, 299]
```

Removing Items from an Array

We can remove items from an array using the `pop` and `shift` methods. The `pop` method allows us to remove an item from the end while the `shift` method allows us to remove an item from the beginning.

```
const nums = [10, 20, 30, 40]

nums.pop()
nums.pop()
nums.shift()

console.log(nums) // Will print [20]
```

You can remove an item from anywhere in an array using `splice`. The first argument is the index to start at. The second argument is the number of items you want to remove. Set the second arguments value to `1` to remove a single item.

```
const nums = [99, 199, 299]

// Remove the second item in the list
nums.splice(1, 1)

console.log(nums) // Will print [99, 299]
```

Changing Array Items

There are two main ways to change an item. The first is to directly override it with bracket notation. The second is to use `splice`. When using splice, you delete the item and provide its replacement as the third argument.

```
const nums = [10, 20, 30, 40]

// Use bracket notation to replace the last item
nums[3] = 1000

// Use splice to replace the first item with 2000
nums.splice(0, 1, 2000)

console.log(nums) // Will print [2000, 20, 30, 1000]
```

Documentation Links

- Push Method
- Pop Method
- Shift Method
- Unshift Method
- Splice Method

Lesson 4: Looping Over Arrays

In this video, you're going to learn how to loop over arrays. This will allow you to do something for each item in your list, such as showing them to your users.

The forEach Method

The `forEach` method is a bit different from any method we've used before. Like others, `forEach` takes an argument. The difference is that the argument's values is a function. That means we're going to end up passing a function into a function. When we do this, the function we define and pass into the other function is known as a callback function.

In the case of `forEach`, our callback function gets called one time for each item in the array. The callback function gets called with the individual array item and the index for that item.

```
const todos = ['Order cat food', 'Clean kitchen', 'Buy food', 'Do work',
'Exercise']

// This will print a numbered list for each todo item
todos.forEach(function (todo, index) {
    const num = index + 1
    console.log(`${num}. ${todo}`)
})
```

Documentation Links

- forEach Method

Lesson 4.5: The For Loop

In this video, you're going to explore the for statement. The for statement, also known as the for loop, is a looping mechanism similar `forEach` .

Looping With the For Statement

The for statement is similar to forEach, but it's more flexible. The for statement doesn't require an array to iterate over.

The below example uses a for statement to count from 0 to 2. The output would be `0`, `1`, then `2`.

```
for (let count = 0; count <= 2; count++) {
    console.log(count)
}
```

The configuration lives inside of the parenthesis. There are three components, the initializer, the condition, and the final expression.

The initialize lets you setup the starting point. This is represented as a number. The initilizer will run a single when the for statement executes.

The condition is what determines if the for statement should loop again. If the condition is true, the loop will run again. If the condition is false, the loop will not run again.

The final expression runs after an iteration of the loop. This is where we increment our counter so we can move on to the next item.

Looping over Arrays

In the example below, a for statement is used to print the names of the top three players on the team. This would print a message for Breanna, Kathi, and Hunter. It wouldn't print a message for Andrew, since he's not in the top three.

```
const players = ['Breanna', 'Kathi', 'Hunter', 'Andrew']

for (let count = 0; count < 3 && count < players.length; count++) {
  const num = count + 1
  const player = players[count]
  console.log(`${num}. ${player}`)
}
```

Documentation Links

- for loop

Lesson 5: Searching Arrays: Part I

In the next two videos, you're going to learn how to search for items in an array.

The indexOf Method

The `indexOf` method allows us to get the position of an item in the array. If the item exists, its index will be returned. If the item does not exist, `-1` will be returned.

```
const places = ['Philadelphia', 'Texas', 'Vermont']
const index = places.indexOf('Vermont')

console.log(index) // Will print 2
```

Remember, this is not going to work for an array of objects. `indexOf` uses `===` to compare the values. We know that `'Test' === 'Test'` will be `true`, but `{} === {}` will be `false`. For objects, we have `findIndex`.

The findIndex Method

The `findIndex` method allows us to provide our own definition of what makes two things equal. Maybe it's the `title` property for our notes or the `text` property for our todos. In

the example below, we use the notes `title` property value. Return `true` from the callback function when you've found the match.

```
const notes = [{
    title: 'My next trip',
    body: 'I would like to go to Spain'
}, {
    title: 'Habbits to work on',
    body: 'Exercise. Eating a bit better.'
}, {
    title: 'Office modification',
    body: 'Get a new seat'
}]

const index = notes.findIndex(function (note, index) {
    return note.title === 'Habbits to work on'
})

console.log(index) // Will print 1
```

Documentation Links

- indexOf Method
- findIndex Method

Lesson 6: Searching Arrays: Part II

This video is a continuation of the last one. You'll continue to explore array searching.

The find Method

The **`find`** method is similar to **`findIndex`**. We provide a callback function that returns `true` for a match and `false` otherwise. The difference is that **`findIndex`** returns the index while **`find`** returns the item itself.

```
const notes = [{
    title: 'My next trip',
    body: 'I would like to go to Spain'
}, {
    title: 'Habbits to work on',
    body: 'Exercise. Eating a bit better.'
}, {
    title: 'Office modification',
    body: 'Get a new seat'
}]

const findNote = function (notes, noteTitle) {
    return notes.find(function (note, index) {
        return note.title.toLowerCase() === noteTitle.toLowerCase()
    })
}

const note = findNote(notes, 'my next trip')
console.log(note) // Will print the first object from our array above
```

Documentation Links

- find Method

Lesson 7: Filtering Arrays

In this video, you're going to learn how to filter arrays. Imagine you want to filter all the todos down to just the todos that still need to be completed.

The filter Method

The `filter` method allows us to filter an array. It returns a new array with just the filtered items.

`filter` takes a callback function which gets called one time for each item. As always, the callback function gets called with the array item and the array item index. If you want to include an item in the new array, the callback function should return `true`. If you want to exclude an item from the new array, the callback function should return `false`. The new array is returned from the `filter` method.

You can filter your items based on what makes sense for your application. In the following example, we set up `getThingsToDo` to return a list of the todos that still need to be completed. This doesn't modify the `todos` array in any way.

```
const todos = [{
    text: 'Order cat food',
    completed: false
}, {
    text: 'Clean kitchen',
    completed: true


}, {
    text: 'Do work',
    completed: false
}]


const getThingsToDo = function (todos) {
    return todos.filter(function (todo) {
        return !todo.completed
    })
}

// Will print an array of all todos that still need to be completed
console.log(getThingsToDo(todos))
```

Documentation Links

- filter Method

Lesson 8: Sorting Arrays

In this video, you'll learn how to sort your arrays. It doesn't make much sense to have completed todos at the top of the list. What if we could keep the completed ones at the bottom and the incomplete ones up top?

The sort Method

The `sort` method allows you to sort an array based off of whatever criteria you'd like. We could sort the todos by their `text` value in alphabetical order. We could also sort them by their `completed` property. Anything goes.

The `sort` method takes a callback function. The biggest difference between `sort` and many other array methods is that this callback function doesn't get called once for each item. Instead, the callback function gets called with two items from the array commonly called `a` and `b`. The function you create decides which should come first.

If `a` should come first, you want to return `-1`. If `b` should come first, you want to return `1`. If they're equal, you want to return `0`.

The example below uses sort to rearrange our todos. We keep the incomplete todos at the top of the list and move the completed ones to the bottom.

```
const todos = [{
    text: 'Buy food',
    completed: true
}, {
    text: 'Do work',
    completed: false
}, {
    text: 'Exercise',
    completed: true
}]

const sortTodos = function (todos) {
    todos.sort(function (a, b) {
        if (!a.completed && b.completed) {
            return -1
        } else if (!b.completed && a.completed) {
            return 1
        } else {
            return 0
        }
    })
}

sortTodos(todos)
console.log(todos)
```

Documentation Links

- sort Method

Lesson 9: Improve Our Expense Tracker

In this video, it's going to be up to you to build out a new version of our expense tracker.

There are no notes for this lesson as we don't cover any new JavaScript features.

Section 7. JavaScript in the Browser

Lesson 1: Section Intro

In this section, you're going to learn how to use JavaScript in the browser. This will let you create website that can request information from the user and render dynamic content. We also start building out the UI for the note taking application and the to-do app.

Lesson 2: Setting up a Web Server

In this video, you're going to install a web server and create a bare-bones HTML file. By the end you'll have a URL you can visit in the browser to view the simple website.

Live-Server

The web server we're going to use for local development is live-server. We can install it using the following command:

```
npm install -g live-server
```

We can then create a web server from a give folder using:

```
live-server folder-to-server
```

HTML

HTML (HyperText Markup Language) is used to create our web pages. It's the more fundamental building block for the web. HTML is where you specify the content to show in the browser.

Here's the basic HTML document that we create in this video to render a title and message to the screen:

```
<!DOCTYPE html>

<html>
    <head>

    </head>
    <body>
        <h1>My App</h1>
        <p>This application was created by Andrew Mead</p>
    </body>
</html>
```

Documentation Links

- HTML
- Live-Server

Lesson 3: JavaScript in the Browser

In this video, you're going to run JavaScript in the browser. You'll learn how to connect your script with your HTML so your script runs when the pages load. This will allow you to create dynamic web apps as opposed to static websites.

Script Tag

The script tag lets you run JavaScript from the browser. The script can be in-line or external. An in-line script is where the JavaScript code ends up right inside the HTML file. This is not preferred and typically creates a mess as your app gets more complex.

```
<script>
    console.log('A message from JavaScript!')
</script>
```

It's best to keep your JavaScript in a separate JavaScript file. You can then load the script in using the script tag and the "src" attribute like so.

```
<script src="my-script.js"></script>
```

Documentation Links

- Script Tag

Lesson 4: DOM Manipulation

In this lesson, you're going to learn about the DOM (Document Object Model). The DOM is what allows you to change what the user sees by modifying the HTML document from JavaScript.

Querying for Elements

The DOM is a JavaScript object. It represents the HTML document and we can use it in our JavaScript to manipulate what the user sees in the browser. The DOM is provided to us via the `document` global variable.

When we want to manipulate the document, we first need to query the DOM for the thing we're trying to change. For example, we might want to remove a paragraph or change the text of a message. We can't do either of those until we have queried the DOM and found the element we were hoping to change.

We can query the DOM with the `querySelector` method. This method takes a single argument where we provide our query. In the case below, we're searching for a "p" element.

```
const element = document.querySelector('p')
```

The `querySelector` method is going to serach the DOM for the first match it finds. It'll then return that match.

If you need to select multiple things, you can use `querySelectorAll`. This will return an array with all the matches found. The below example would give us an array with all the paragraph elements found in the DOM.

```
const elements = document.querySelectorAll('p')
```

Manipulating Elements

We can remove an element using the `remove` method. The below example would remove the first `h1` element in the DOM.

```
const element = document.querySelector('h1')
element.remove()
```

We can also read and change an element's text value. In the example below, we add an exclamation mark onto the text shown in a paragraph.

```
const element = document.querySelector('p')
element.textContent = element.textContent + '!'
```

Documentation Links

- DOM
- querySelector
- querySelectorAll
- remove
- textContent

Lesson 5: DOM Challenge

In this video, you're going to use what you've learned about DOM manipulation.

There are no notes for this challenge video since no new information was covered.

Lesson 6: Adding Elements via the DOM

In this video, you're going to learn how to add a new element to a web page via the DOM. This is useful when it comes to rendering a dynamic web page. You'll be using this to render your application data such as the list of to-do items to be completed.

Adding Elements

There are three main steps when adding a new element. We need to create the element, customize the element, and render the element somewhere in the document. Getting this done requires two new methods, `createElement` and `appendChild`.

`createElement` allows us to create a new DOM element. We provide a single argument to set up the element type. In the case below, I create a new paragraph element.

`appendChild` allows us to add a new item somewhere in the DOM. We pass in the newly created element as the first and only argument. Below, I add the paragraph onto the end of the body.

```
const newParagraph = document.createElement('p')
newParagraph.textContent = 'This is a new element from JavaScript'
document.querySelector('body').appendChild(newParagraph)
```

Documentation Links

- createElement method
- appendChild method

Lesson 7: Handling User Interaction

In this video, you'll learn how to handle user interaction. This will allow us to respond to things the user does like clicking a button or entering some text in a field.

Adding Event Handlers

Adding an event handler allows you to run some JavaScript code when something happens. This would enable you to execute some JavaScript code when a user clicks a specific button on the screen.

We add an event handler onto a DOM element using `addEventListener`. The method takes two arguments. The first argument is where you specify what event you want to watch for. The second argument is where you specify what you want to do when the event happens.

In the example below, I add an event listener onto a button. When the button gets clicked I change its text.

```
document.querySelector('button').addEventListener('click', function (e) {
    e.target.textContent = 'The button was clicked'
})
```

Documentation Links

- addEventListener (the sidebar on that page contains a list of available events)

Lesson 8: Advanced Queries

In this video, you're going to learn how to better select DOM elements. Selecting by tag name is great, but that's not going to work well once we have multiple elements on the page with the same tag name.

IDs and Classes

HTML gives us a couple of attributes to make targeting our DOM elements a bit easier.

The first is the `id` attribute. The `id` attribute is designed to be a unique identifier for an element in your document. You shouldn't have two elements with the same id.

Here's an example of a tag with an `id` attribute set.

```
<h1 id="my-id">My title</h1>
```

The second is the class attribute. The class attribute is not unique, so you could have multiple elements in your document with the same class. An element can also have multiple classes by setting a space-separated list of classes.

Here's an example of an h1 tag with one class and a p tag with two classes.

```
<h1 id="second-class">My title</h1>
<p class="class-one second-class">My title</p>
```

Targeting Elements By Id and Class

You can target elements by their id or classes via `querySelector` or `querySelectorAll`. Ids should be prefixed with "#" while classes should be prefixed with ".".

```
// Select an element with an id of "my-id"
document.querySelector('#my-id')

// Select all elements with an class of "special"
document.querySelectorAll('.special')
```

Lesson 9: Text Inputs and Live Data Filtering

In this video, you're going to set up your first text input. This will allow your app users to type in a value such as their name, email, or anything else.

Creating an Input

You can create a text input using the "input" tag. The example below also sets `type` equal to `text`. This is the default value, so the `type` attribute could be removed altogether. It's

kept here for completeness. The example also sets up an optional `placeholder` value to show to the user before text is entered.

```
<input id="search-text" type="text" placeholder="Filter todos">
```

Listening for Input Changes

Your text inputs will fire an "input" event for each change made to the input. You can listen for this event using the same old `addEventListener` method. You can access the inputs text value using the `value` property. The below example listens for any changes to an input and prints the new value when a change occurs.

```
document.querySelector('#search-text').addEventListener('input', function (e)
{
    console.log(e.target.value)
})
```

Documentation Links

- Input Element
- Input Event

Lesson 10: Rendering Our Filtered Data

In this video, you'll explore how to set up live data filtering. This will allow users of the notes app to filter their notes by entering part of the note title in a text input.

This video uses filtering techniques covered in the section on arrays. There are no notes as no new language features were explored.

Lesson 11: Todo Filter Challenge

In this challenge video, you'll be setting up filter for the to-do application. Get to it!

There are no notes as no new language features were explored.

Lesson 12: Working With Forms

In this video, you're going to create your first form. At first, the forms will only use text inputs. Other field types like checkboxes and dropdowns will be explored later in this section.

Setting up a Form

Forms are created using the “form” tag. Our form fields and buttons go inside the form tag. The example form below asks the user for a single value and gives them a button to click when they’re ready to submit the form.

```
<form id="name">
    <input type="text" placeholder="Enter your name" name="firstName">
    <button>Submit</button>
</form>
```

The text input above also has its `name` attribute set. This is done so we can access the value of that input which we do below.

Handling Form Submissions

You can run some JavaScript code when a form is submitted by listening for the `submit` event. In the past, form submissions were not handled with client-side JavaScript as a large percentage of browsers didn’t support JavaScript. That’s no longer the case, but we do need to override the default behavior if we want to run some code for handling the form submissions. This is done via our call to `e.preventDefault()` below. This disabled the default browser behavior and lets you handle the submission via your code.

```
document.querySelector('#new-todo').addEventListener('submit', function (e) {
    e.preventDefault()
    console.log(e.target.elements.firstName.value)
    e.target.elements.text.value = ''
})
```

Documentation Links

- Form Tag
- Submit Event

Lesson 13: Checkboxes

In this video, you’re going to learn about the checkbox. You’ll learn how to set up a checkbox and respond to changes when the box is checked and unchecked.

Setting up a Checkbox

Creating a checkbox uses the “input” tag. The difference between a checkbox and a text field is that the checkbox input sets `type` equal to `checkbox`.

```
<label>
    <input id="delivery" type="checkbox"> Check for delivery
</label>
```

Listening for Checkbox Changes

A checkbox fires the `change` event when its value is changed. You can access this value via the `checked` property on the element. The `checked`property will be `true` if the checkbox is checked and `false` if the checkbox is unchecked.

```
document.querySelector('#delivery').addEventListener('change', function (e) {
    console.log(e.target.checked)
})
```

Documentation Links

- Input Tag
- Change Event

Lesson 14: Dropdowns

In this video, you’re going to learn about the dropdown. You’ll learn how to set up the dropdown, configure its options, and respond to changes when a new option is picked.

Setting up a Select Dropdown

You’ll use two elements when setting up a dropdown. The first is “select”. This sets up the actual dropdown in the browser. The second is `option`. It’s used to add options to the dropdown.

The value of the select element depends on the option that was clicked. If that option has a `value` attribute set, that attribute’s value will be used. If the option doesn’t have value then the option text will be used.

```
<select id="filter-by">
    <option value="byEdited">Sort by last edited</option>
    <option value="byCreated">Sort by recently created</option>
    <option value="alphabetical">Sort alphabetically</option>
</select>
```

Listening for Select Dropdown Changes

The select element fires the “change” event when a new option is picked from the dropdown. You can access the value of the selected item by using its `value` property like you did for text inputs.

```
document.querySelector('#filter-by').addEventListener('change', function (e)
{
    console.log(e.target.value)
})
```

Documentation Links

- Select Tag
- Option Tag
- Change Event

Section 8. Data Storage, Libraries, and More

Lesson 1: Section Intro

In this section, you’re going to continue working on the note and to-do applications. You’ll learn how to save user data and work with third-party libraries.

Lesson 2: Saving Our Data in LocalStorage: Part I

In this video, you’ll learn about local storage. This is a storage mechanism built right into the browser. This is going to allow us to store notes and to-dos for later. We’ll cover the basic CRUD operations (create, read, update, delete).

The setItem Method

We can access local storage via the `localStorage` global object. Similar to an object, local storage is a key/value store. We can store some data under a specific key and access it later.

The `setItem` method is used to save data. It requires two arguments. The first is the key and the second is the value. The example below stores the string `"Andrew"` in the key `"username"`.

```
localStorage.setItem('username', 'Andrew)
```

The getItem Method

The `getItem` method is used to get saved data out of local storage. It takes a single arugment which is the key. `getItem` returns the value found for that key.

In the below example, we fetch the saved name and print it to the console.

```
const name = localStorage.get('username')
console.log(name)
```

The removeItem Method

The `removeItem` method is used to delete some data from local storage. It takes as its only argument the key for the data you want to remove. It doesn't return anything.

The below example removes the data associated with `'username'` in local storage.

```
localStorage.removeItem('username')
```

The clear Method

The `clear` method allows you to delete all the data stored in localStorage. It takes no arguments and returns nothing.

```
localStorage.clear()
```

Documentation Links

- localStorage

Lesson 3: Saving Our Data in LocalStorage: Part II

In this video, you'll start integrating local storage into the apps. This will allow users to add some to-dos or notes and pick up where they left off.

This video uses filtering techniques covered in the last video. There are no notes as no new language features were explored.

Lesson 4: Splitting up Our Application Code

In this video, we're going to spend a bit of time refactoring our code. This is going to improve the quality and organization of our code without changing the application's behavior.

Breaking Up Large Files

Adding new features is going to increase the amount of code we have and the complexity of our code. If we continue building, we're going to end up with a ton of code sitting in a single JavaScript file. That makes things hard to find and it makes writing and maintaining our code more difficult than it needs to be.

You can avoid these large complex files by breaking up your JavaScript into smaller separate files. This can be done by creating another file and loading it with a new script tag.

Going forward, we're going to have two files for our apps. One is going to define some functions and the other is going to use those functions as well as set up our browser events.

```
<script src="notes-functions.js"></script>
<script src="notes-app.js"></script>
```

Lesson 5: Refactor Challenge

In this video, you're going to refactor the to-do application using what you learned in the last video.

This video uses the refactoring techniques discussed in the last video. There are no notes as no new language features were explored.

Lesson 6: Debugging Our Applications

In this video, you're going to learn how to debug your JavaScript code. This will reduce the amount of time you spend tracking down small typos.

The Debugger

When our JavaScript code isn't working as expected, we want to be able to track down where things went wrong. This typically involves us printing out variable's values as well as "Does this code run?!?!" so we can better track the state of our program as it executes. Getting this done with `console.log` is a great way to start, but there's a better way.

The `debugger` statement is going to be our replacement for debugging with `console.log`. When we use `debugger` in our code, the browser stops at that line. You can then explore variable's values and code flow while the script is paused. This makes it easy to dig into all parts of your application to see where things went wrong.

In the below example, we pause the script before passing the data to our function.

```
const hotel = 'Radish Inn'
debugger
checkIn(hotel)
```

Documentation Links

- Debugger

Lesson 7: Complex DOM Rendering

In this video, you're going to learn how to render more real-world elements to the DOM. Rendering a single element is fine, but what if we want a checkbox, some text, and a delete button for each to-do item?

Complex DOM Rendering

We don't need any new methods to render more complex content to the DOM. We're still going to use `createElement` and `appendChild` to create and render our content. The only difference is that we'll end up with a few more of those method calls in our code.

To start, we want to create a new element which will serve as the root element. Everything we want to show will end up going inside of this root element. The root element gets rendered to the document after we've set up all the content needed.

The example below shows how we can render a button and span inside a div. The div then gets rendered by appending it onto the body.

```
const root = document.createElement('div')
const text = document.createElement('span')
const button = document.createElement('button')

// Setup the text
text.textContent = 'Scranton, PA'
root.appendChild(text)

// Setup the button
button.textContent = 'x'
root.appendChild(button)

document.body.appendChild(root)
```

Lesson 8: Setting up a Third-Party Library

In this video, you're going to work with your first third-party JavaScript library. Third-party libraries allow you to use code written by others.

UUID

The library we're going to set up is UUID (Universally Unique Identifier). This library gives us a function we can call to generate secure and unique id for our notes and to-dos.

We need to load in the third-party script before we can call the new function. In the example below, we load in the library before loading in the code we wrote.

```
<script src="http://wzrd.in/standalone/uuid%2Fv4@latest"></script>
<script src="notes-functions.js"></script>
<script src="notes-app.js"></script>
```

Now, we're ready to use the library. You'll need to refer to the library docs to figure out how exactly you use it. Each library is different and provides you with a unique set of things you can do. In the case of UUID, we get a single function `uuidv4` which we can call.

This is not a function built into JavaScript and it's not something we wrote. It's defined in the library file we included above.

```
const id = uuidv4()
console.log(id) // Will print the new random id
```

Documentation Links

- UUID Docs

Lesson 9: Targeting by UUID

In this video, you're going to target your note and to-dos by their id. This will allow you to read their data, update the data, or remove the item altogether.

There are no notes as no new language features were explored.

Lesson 10: Checkbox Challenges

In this video you're going to wire up the checkbox to actually toggle the completed status of your to-dos.

This is a challenge video. There are no notes as no new language features were explored.

Lesson 11: The Edit Note Page: Part I

In this video, you're going to set up a second HTML page for our note application. This is where we'll allow users to edit the note's title and body content.

Navigating Between Pages

Once you have two documents you need a way to switch between them. Sure, someone can change the URL directly in the address bar, but we want to change pages via the application itself.

There are two primary ways we'll be changing pages. The first is to set up links that the user can click. The second is to set up JavaScript that redirects users between the pages.

You can set up a link with the anchor tag.

```
<a href="/page.html">Go to my page</a>
```

You can redirect via JavaScript using `location.assign`. This method chooses the path to go through, similar to the “href” attribute of our link above.

```
location.assign(`/page.html`)
```

Lesson 12: The Edit Note Page: Part II

In this video, you’re going to learn how to pass an id between pages. This is going to allow us to tell the edit page which note we want to edit.

URL Hash

When the edit page loads it needs to know which note it should be editing. We provide the id of the note to edit via the URL hash.

The URL hash is the part of the URL that comes after “#”. The value that comes after the hash can be accessed via JavaScript. For example, “mysite.com/edit.html#123abc” where “123abc” is the id.

The hash value lives on `location.hash` and includes the “#” character. We can remove that using the string `substring` method. The example below extracts the id from the URL hash.

```
const id = location.hash.substring(1)
```

Lesson 13: Syncing Data Across Pages

In this video, you’re going to set up real-time data syncing across pages. If data changes on one page, all other pages will be automatically updated.

The storage Event

The "storage" event lets you run some code whenever localStorage changes. We set up the event with `addEventListener` just like we did for our other browser events.

The below code sets up a function to run whenever localStorage changes.

```
window.addEventListener('storage', function (e) {
    // Will fire for localStorage changes that come from a different page
})
```

It's important to note that the event will only fire for localStorage changes made by other pages.

Lesson 14: JavaScript Dates

In this video, you're going to learn about the built-in date in JavaScript. This gives you a way to work with dates and time in your application.

Creating a Date

We can create a date that represents the current point in time with the following code.

```
const now = new Date()
```

We can also create a date that represents a point in the past. To do this, provide a string as the first argument to `Date` like so.

```
const dateOne = new Date('March 1 2017 12:00:00')
```

Extracting Data from a Date

Once you have a date, you can extract various pieces of information using one of its many methods. In the below example, we print a date with the month, then the day of the month, and the year.

```
const dateOne = new Date('March 1 2017 12:00:00')
const month = dateOne.getMonth() + 1
const day = dateOne.getDate()
const year = dateOne.getFullYear()

console.log(`${month}/${day}/${year}`) // Wil print "3/1/2017"
```

We can also extract the hours, minutes, and seconds by using `getHours`, `getMinutes`, and `getSeconds` respectevly.

Working with Timestamps

A timestamp is nothing more than a number. This number represents the number of milliseconds since January 1st 1970 at midnight. Positive numbers go forward in time from that point. Negative numbers go into the past. A date in 2018 would be represented by a positive number. A date in 1969 would be represented by a negative number.

Timestamps are great because passing around a single number is super easy. You can, for example, store it in localStorage which is exactly what we're going to do.

You can create a timestamp from a date with the following.

```
const date = new Date('March 1 2017 12:00:00')
const timestamp = date.getTime()
console.log(timestamp) // Will print 1488387600000
```

You can get a JavaScript date from a timestamp using the following.

```
const timestamp = 1488387600000
const date = new Date(timestamp)
```

Documentation Links

- Date

Lesson 15: Moment

Does JavaScript come with built-in support for dates? Yup. Is it fun to use? Nope. In this video, you'll learn about moment.js, a third-party JavaScript library that makes working with dates better.

Creating a Moment

The moment library comes with a nicer and more usable set of methods. These make it easy to create dates and format them to fit your needs. You can create a new moment with the following.

```
const date = moment()
```

You can create a moment from a timestamp with the following:

```
const timestamp = 1488387600000
const date = moment(timestamp)
```

Formatting a Moment

The `format` method is one of the best reasons to use moment.js. The format method lets you provide a date pattern which describes what information you want to show. Moment then replaces parts of the pattern with the actual information.

For example, if I wanted to use a format like “January 1st, 2018”, all I need to do is call format with a few patterns.

```
const timestamp = 1488387600000
const date = moment(timestamp)
console.log(date.format('MMMM Do, YYYY')) // Wil print "March 1st, 2017"
```

Documentation Links

- Moment docs
- Format method docs

Lesson 16: Integrating Dates: Part I

In this video, you’re going to start integrating moment into the notes application.

There are no notes as no new language features were explored.

Lesson 17: Integrating Dates: Part II

In this video, you’re going to finish integrating moment into the notes application.

There are no notes as no new language features were explored.

Section 9. Expanding Our JavaScript Knowledge

Lesson 1: Section Intro

In this section, you’ll continue learning new language features. The focus here is on improving our code quality and creating cleaner code. You’ll learn about the arrow function, the conditional operator, type coercion, and more.

Lesson 2: Arrow Functions: Part I

In this video, you're going to learn about arrow functions. You'll learn what they are and how they differ from the functions we've created so far.

Syntax

The syntax for creating an arrow function is going to look familiar. It's close to the syntax for a regular function with a couple of differences. The first difference is that the "function" keyword is no longer needed. The second difference is the addition of the arrow "=>" between the arguments and the function body.

Here's an example.

```
const squareLong = (num) => {
    return num * num
}
console.log(squareLong(3)) // Will print 9
```

Shorthand Syntax

Arrow functions also have a shorthand syntax. This works great when you have a function that returns the result of some expression. We see this a lot with the callback functions we pass to array methods like `filter`.

To use this shorthand syntax, we remove the curly braces from the function definition. We then put the expression we want to return right after the arrow. There's no need for the return keyword as the expression we add is implicitly returned.

The below example is a modified version of `squareLong` from above. This time, it's taking advantage of the shorthand syntax.

```
const squareLong = (num) => num * num
console.log(squareLong(3)) // Will print 9
```

Documentation Links

- Arrow functions
- Shorthand syntax

Lesson 3: Arrow Functions: Part II

In this video, you're going to stick with arrow functions. We'll be exploring a few more features as well as integrating them into the notes and to-do applications.

"this" Bindings

Arrow functions don't bind `this`. This typically isn't a problem, but we'll want to avoid using arrow functions for methods since we won't have access to `this`.

The below example won't work. Arrow functions don't bind `this` so we won't be able to access the pet name.

```
const pet = {
    name: 'Hal',
    getGreeting: () => {
        return `Hello ${this.name}!`
    }
}

console.log(pet.getGreeting())
```

JavaScript does provide a method definition shorthand syntax. This is a way to define a method without needing the function keyword. The below example creates a method which is not an arrow function. This example will correctly print the pet name.

```
const pet = {
    name: 'Hal',
    getGreeting() {
        return `Hello ${this.name}!`
    }
}

console.log(pet.getGreeting())
```

Arrow Functions and "arguments"

Arrow functions don't bind `arguments` either. If you rely on `arguments` you'll need to stick with regular functions.

This example won't work since `add` is set up as an arrow function.

```
const add = () => {
    return arguments[0] + arguments[1]
}

console.log(add(11, 22, 33, 44))
```

This example will work since `add` is set up as a regular function.

```
const add = function () {
    return arguments[0] + arguments[1]
}

console.log(add(11, 22, 33, 44)) // Will print 33
```

Documentation Links

- ES6 method definition

Lesson 4: Conditional (Ternary) Operator

In this video, you're going to learn about the conditional operator. This is a nice shorthand for an if/else statement where you want to do one of two things.

There are three parts of a conditional operator:

1. The condition.
2. The first expression. This is run if the condition passes.
3. The second expression. This is run if the condition fails.

The example below uses the conditional operator to correctly generate the message based off of the team size. The condition checks if the team size is valid and has four or less players. If the condition passes, we print the current team size. If the condition fails, we print an error letting them know they have too many team members.

```
const team = ['Tyler', 'Porter', 'Andrew', 'Ben', 'Mike']
const message = team.length <= 4 ? `Team size: ${team.length}` : 'Too many
people on your team'
console.log(message)
```

The conditional operator can also be used to call one of two functions. The example below sets up a condition to check if the person is 21 or older. If they are, the `showPage` function is called. If they are not, the `showErrorPage` function is called.

```
const age = 21

const showPage = () => {
    // Show some page
}

const showErrorPage = () => {
    // Show an error page
}

age >= 21 ? showPage() : showErrorPage()
```

Documentation Links

- Conditional operator

Lesson 5: Truthy and Falsy Values

In this video, you're going to learn about truthy and falsy values in JavaScript. This allows us to reduce the complexity of our conditionals.

Any value of any type can be used in a Boolean context, such as the condition of an if statement. Some of these values such as `47` are considered truthy and others like `''` are considered falsy. If a truthy value is used in a condition that condition will pass as if `true` was used. If a falsy values is used in a condition that condition will fail as if `false` was used.

Here's an example of a truthy value causing an if statement to run. A string with content is considered truthy.

```
const value = 'Andrew'

if (value) {
    // This will run
} else {
    // This won't run
}
```

Here's an example of a falsy value that will cause the condition to fail. A string with no content is considered falsy.

```
const value = ''

if (value) {
    // This won't run
} else {
    // This will run
}
```

The real question is what values are falsy and which are truth. The easiest way is to look at the six different falsy values. Everything else is truthy.

1. false
2. 0 (number zero)
3. '' (empty string)
4. null
5. undefined
6. NaN

Documentation Links

- Truthy
- Falsy

Lesson 6: Type Coercion

In this video, you're going to learn about type coercion. Type coercion is the process of converting a value from one type, such as a number, to another type, such as a string.

The typeof Operator

The `typeof` operator is used to check the type of a value. It'll return "number", "string", "boolean", "object", "function", or "undefined".

```
// Typeof examples

console.log(typeof 43) // Will print "number"

console.log(typeof 'Andrew') // Will print "string"

console.log(typeof undefined) // Will print "undefined"
```

Type Coercion

Type coercion occurs when the operands of an operator are of different types. You might think that JavaScript would just throw an error to crash the script, but it won't. Instead, JavaScript will try to convert the types of the operands so they're the same. This can cause some unexpected behavior.

In the example below we try to add `false` and `12`. JavaScript doesn't know how to add booleans and numbers together so it converts the boolean over to a number.

```
const value = false + 12
const type = typeof value
console.log(type) // Will print "number"
console.log(value) // Will print 0
```

When `false` is converted to a number, it's represented with `0`. When `true` is converted to a number, it's represented with `1`. So `false + 12` is `12` and `true + 12` is `13`.

It's important to know that JavaScript will try to coerce values of different types, though this is not a behavior that we want to rely on.

Documentation Links

- The typeof operator
- Advanced reading on type coercion

Lesson 7: Catching and Throwing Errors

In this video, you're going to learn how to work with errors in JavaScript. This covers both sides, how we can create errors and how we can recover from them.

Throwing Errors

Errors can be thrown with the `throw` statement. This is going to stop function execution and end the program. This error can be customized to describe the exact reason things failed.

The function `getTip` below will throw an error if it's called with a non-number argument. The example below will terminate with the error "Error: Argument must be a number" since it's called with a string.

```
const getTip = (amount) => {
    if (typeof amount !== 'number') {
        throw Error('Argument must be a number')
    }

    return amount * .25
}
const result = getTip('12')
```

Catching Errors

You don't have to let errors crash the program. Not all errors are due to the JavaScript code itself. Errors can be triggered by external circumstances such as bad input data. There's no code change that can prevent this, so you'll want to catch this error and fix it.

The try/catch statement lets you catch errors. We provide two code blocks, one for "try" and one for "catch". The "try" block contains the code we want to try and run, knowing that it might throw an error. The "catch" block contains the code to run if the "try" block throws an error.

The below example uses a try/catch statement when calling `getTip` with an invalid argument. The "catch" block executes after the error is thrown. We handle the error by printing a message to the console.

```
const getTip = (amount) => {
    if (typeof amount !== 'number') {
        throw Error('Argument must be a number')
    }

    return amount * .25
}

try {
    const result = getTip('12')
    console.log(result) // This won't print
} catch (e) {
    console.log('Error!') // This will print
}
```

Documentation Links

- Try/Catch

Lesson 8: Handling Application Errors

In this video, you're going to integrate a try/catch statement into the to-do and note application. This is going to let us respond to errors thrown by `JSON.parse` when invalid JSON is parsed.

There are no notes for this lesson, as no new language features were explored.

Lesson 9: Working in Strict Mode

In this video, you're going to learn about strict mode in JavaScript. When you enable strict mode, you're opting-in to a modified version of JavaScript that removes dangerous and potentially buggy behavior.

You enable strict mode by adding `'use strict'` as the first line in your script. That's it.

One change made by strict mode is that it's impossible to leak global variables. Leaked globals were covered when function scope was covered. As a reminder, the following code will create a global variable `data` even though that's not what we wanted.

```
const processData = () => {
    data = '1230987234'
}
processData()
console.log(data) // Will print '1230987234'
```

That same code will throw an error if executed in strict mode. The below code throws the following error: `ReferenceError: assignment to undeclared variable data`. This makes it easier for us to catch and fix our mistakes.

```
'use strict'

const processData = () => {
    data = '1230987234'
}
processData()
console.log(data)
```

Check out the link “ECMA-262-5 in detail” below for an in-depth list of all strict mode changes.

Documentation Links

- Strict mode
- ECMA-262-5 in detail

Section 10. Advanced Objects and Functions

Lesson 1: Section Intro

In this section, you’re going to explore how JavaScript handles inheritance via prototypal inheritance. This includes the new operator, the constructor function, the class syntax, and more.

Lesson 2: Object Oriented Programming

In this video, you’re going to be introduced to object-oriented programming. You’ll explore what it is and why it’s a useful tool when working with JavaScript.

This is a presentation video. Please refer to the video for the visualization.

Lesson 3: Constructor Functions

In this video, you're going to build and use a constructor function. Constructor functions allow us to define new types in JavaScript other than the built-in types.

A constructor function is just a regular old function. It defines how to initialize objects for the new type. Below is an example constructor function which contains nothing we haven't seen before in JavaScript.

```
const Person = function (firstName, lastName, age) {
    this.firstName = firstName
    this.lastName = lastName
    this.age = age
}
```

The only thing worth mentioning about the constructor function is that it has a capitalized first letter. This is just a convention and is not enforced by the language. You could call a constructor function "hangman", but "Hangman" is more appropriate. This convention makes it easy to determine if a given function is a constructor function or not.

To create a new instance of `Person` we need to call the constructor function with the `new` operator. This example creates a new person and then dumps it to the console.

```
const me = new Person('Andrew', 'Mead', 27)
console.log(me)
```

The example won't work if you remove the `new` operator. That's because it plays an important role behind the scenes by doing the following:

1. A new object is created for the new instance.
2. The constructor function is called with `this` bound to the new object.
3. The newly created object is implicitly returned. You can override this and manually return any object you like, but there's no real reason to do this.

To summarize, the above code calls the constructor function. The constructor function then configures this new object with some values. Finally, we get access to this newly configured object via the return value. This allows us to create new instances of a custom type.

Documentation Links

- new operator
- Constructor functions

Lesson 4: Setting up the Prototype Object

In this video, you're going to learn about the prototype property. The prototype property gives you a way to define a shared set of properties/methods for all your instances.

The Prototype Property

The things that make each person unique are set up in the constructor function. Each person can have a unique name, age, and a set of interests. The things that make each person the same are set up on the prototype property. This typically includes a set of methods that define shared functionality.

This example sets `getBio` on `Person.prototype`. Whatever we put on `Person.prototype` is shared with all instances of `Person`. This can be seen below where two instances are created and `getBio` gets called for both. The same code is running in both cases. It's the unique instance properties that make the output unique. Methods can access the instance and its properties on `this`.

```
const Person = function (firstName, lastName, age, likes = []) {
    this.firstName = firstName
    this.lastName = lastName
    this.age = age
    this.likes = likes
}

Person.prototype.getBio = function () {
    let bio = `${this.firstName} is ${this.age}.`

    this.likes.forEach((like) => {
        bio += ` ${this.firstName} likes ${like}.`
    })

    return bio
}

const me = new Person('Andrew', 'Mead', 27, ['Teaching', 'Biking'])
console.log(me.getBio())

const person2 = new Person('Clancey', 'Turner', 51)
console.log(person2.getBio())
```

Lesson 5: Hangman Challenge: Part I

In this video, you're to build out part of the hangman game.

There are no notes for this lesson, as no new language features were explored.

Lesson 6: Digging Into Prototypical Inheritance

In this presentation, you're going to explore how prototypal inheritance works behind the scenes. This will explain why we're able to add our shared methods onto the constructor's `prototype` and then access them on each instance.

The "[[Prototype]]" Property

JavaScript creates a hidden link behind the scenes that connects your instance object with the constructor functions `prototype` property. This takes place when you call the constructor function with the `new` operator. This internal link is what allows instances to access object properties like `firstName` and prototype properties like `getBio`.

The link itself is created by setting the instance objects `[[Prototype]]` property. This is a hidden internal property that's used behind the scenes to make prototypal inheritance

work. You can image that for a new person, the following line runs when creating the object.

```
me.[[Prototype]] = Person.prototype
```

The Prototype Chain

The prototype chain describes the link between instance objects and their prototypes. This is an important relationship that explains how code is shared.

What happens when you access a property on an object? JavaScript starts by looking for it on the object itself. JavaScript checks for a matching property name and gives you back the value for the matching property.

What happens when JavaScript doesn't find the property on the object itself? JavaScript looks to see if the object has a prototype by checking if the internal property `[[Prototype]]` is set. If it is set, JavaScript will look for the property value there.

This explain why we're able to access both `firstName` and `getBio` on an instance of `Person`. `firstName` lives directly on the objet itself. `getBio` lives on the prototype object.

Documentation Links

- Prototypal Inheritance

Lesson 7: Primitives and Objects: Part I

In this video, you're going to explore how JavaScript uses prototypal inheritance for built-in types.

The Prototype Chain for Objects

You've seen that all arrays have a `filter` method, all strings have a `toLowerCase` method, and all objects have a method called `hasOwnProperty`. This is possible because JavaScript uses prototypal inheritance to set up a shared set of methods just like we did for our custom types.

Take a look at the prototype chain for a regular old object.

```
// myObject --> Object.prototype --> null
const myObject = {}

console.log(myObject.hasOwnProperty('doesNotExist')) // Will print false
```

JavaScript starts by looking for the property on `myObject` itself. If it's not found there, JavaScript goes to the next link in the chain and checks for it there. At some point, the next link in the chain will be `null`. That's when JavaScript stops looking for the object property and gives back the value of `undefined`. This explains why you can access `hasOwnProperty` on an object even though it was never manually set.

Notice that the above example doesn't use the `new` operator with a constructor function. There is a constructor function for creating an object, but it's not required for this built-in type.

```
const myObject = new Object({})

console.log(myObject.hasOwnProperty('doesNotExist')) // Will print false
```

This would work just the same as our previous program.

Documentation Links

- Object.prototype

Lesson 8: Primitives and Objects: Part II

In this video, you're going to explore how JavaScript uses prototypal inheritance for the other built-in types. This is part 2 of 2.

Primitive Values

There are five primitive values in JavaScript. Strings, numbers, booleans, null, and undefined. A primitive value is a value that's not an object and has no method.

Strings, Numbers, and Booleans have an object wrapper. This is what allows us to access methods such as `toLowerCase` on a string. JavaScript converts the primitive to an object, runs the method, and then returns the value back to you.

The Prototype Chains for Various JavaScript Values

Below is the prototype chain for arrays, functions, strings, numbers, and boolean. Notice that all of them inherit from `Object.prototype`. They are all just custom object types with their own constructor functions and their own set of shared methods.

```
// Array: myArray --> Array.prototype --> Object.prototype --> null
// Function: myFunc --> Function.prototype --> Object.prototype --> null
// String: myString --> String.prototype --> Object.prototype --> null
// Number: myNumber --> Number.prototype --> Object.prototype --> null
// Boolean: myBoolean --> Boolean.prototype --> Object.prototype --> null
```

There are constructor functions for all our types above. This is exactly what we saw for objects in the last video. As an example, here are two strings. One was created as a primitive and the other was created using the constructor. Both examples output the same value.

```
const product = 'Computer'
console.log(product.toLowerCase()) // Primitive value gets converted into an
object to run method

const otherProduct = new String('Computer')
console.log(otherProduct.toLowerCase())
```

Documentation Links

- Array.prototype
- Function.prototype
- String.prototype
- Number.prototype
- Boolean.prototype

Lesson 9: Hangman Challenge: Part II

In this video, you're to build out part of the hangman game.

There are no notes for this lesson, as no new language features were explored.

Lesson 10: Hangman Challenge: Part III

In this video, you're to build out part of the hangman game.

There are no notes for this lesson, as no new language features were explored.

Lesson 11: Hangman Challenge: Part IV

In this video, you're to build out part of the hangman game.

There are no notes for this lesson, as no new language features were explored.

Lesson 12: The Class Syntax

In this video, you're going to explore the class syntax. This is a newer language feature that makes setting up our constructors and methods a bit easier.

It's important to note that the class syntax doesn't change what's happening behind the scenes. It's just syntactical sugar. It's still the same old prototypal inheritance.

The following example is a recreation of `Person` as a class. This class still has the constructor function and instances of the class still inherit the `getBio` method. Notice that the usage in the last two lines is exactly the same as before.

The class definition allows you to set up some methods. The `constructor` method ends up getting used as the constructor function. All other methods end up being treated as if they were added on the `prototype` property of our old constructor function. One last thing to point out is that the class definition does not accept `,` between the methods.

```
class Person {
    constructor(firstName, lastName, age, likes = []) {
        this.firstName = firstName
        this.lastName = lastName
        this.age = age
        this.likes = likes
    }
    getBio() {
        let bio = `${this.firstName} is ${this.age}.`

        this.likes.forEach((like) => {
            bio += ` ${this.firstName} likes ${like}.`
        })

        return bio
    }
}

const person2 = new Person('Clancey', 'Turner', 51)
console.log(person2.getBio())
```

Documentation Links

- Classes

Lesson 13: Creating Subclasses

In this video, you'll learn how to create and use subclasses. This allows you to create a class that inherits behavior from another class you created.

Extending a Class

You can use subclassing to create a class that inherits from another class. This allows you to share functionality while still customizing the subclass. The subclass might have additional instance properties, it might have a few new methods, or it might override some methods from the parent class.

For example, we can create a subclass of `Person` called `Student`. A student has many of the same qualities as a person, but it also has some things specific to the student such as their grade. That could look like this.

```
class Student extends Person {
    constructor(firstName, lastName, age, grade, likes) {
        super(firstName, lastName, age, likes)
        this.grade = grade
    }
    updateGrade(change) {
        this.grade += change
    }
    getBio() {
        const status = this.grade >= 70 ? 'passing' : 'failing'
        return `${this.firstName} is ${status} the class.`
    }
}
```

The above example uses `extends` to create the subclass. After `extends` you provide the name of the class you want to extend.

The above example also adds on a new property called `grade` which has been added to constructor function arguments. In the constructor function `super` is called. `super` is the parent classes constructor function. This gets called with the arguments specific to the `Person` constructor.

The subclass also adds on a new method called `updateGrade` that allows you to change the students grade. Lastly, the subclass overrides getBio. This allows it to provide a version of `getBio` specific to the student. You don't have to override methods from the parent, but in this case we did want to include the student's grade in the bio.

Documentation Links

- Using extends

Lesson 14: Getters and Setters

In this video, you're going to play around with custom getters and setters in JavaScript. These allow you customize what happens when someone sets or gets an object property.

Getter

Setting up a getter allows you to override what happens when an object property is looked up. Instead of looking for a property value, JavaScript will run your function and use the value your function returns. In the below example, you're able to access `human.name` and get `"Alexis Turner"` even both the first and last name are stored on separate properties. This is because the getter for `name` returns a template string that combines the two.

```
const human = {
  firstName: 'Alexis',
  lastName: 'Turner',
  get name() {
    return `${this.firstName} ${this.lastName}`
  }
}

console.log(human.name) // Prints "Alexis Turner"
```

Setter

A setter allows you to override what happens when an object property is set. This can be useful if you want to validate or sanitize data. It can also be useful if you need to do something unique with the value.

This example adds a setter for `name`. When name is set, we split it up and actually change `firstName` and `lastName` behind the scenes. The code also trims leading and trailing spaces from the name to sanitize the data.

```
const human = {
  firstName: 'Alexis',
  lastName: 'Turner',
  get name() {
    return `${this.firstName} ${this.lastName}`
  },
  set name(name) {
    const names = name.trim().split(' ')
    this.firstName = names[0]
    this.lastName = names[1]
  }
}

human.name = '    Andrew Mead  '
console.log(human.firstName) // Prints "Andrew"
console.log(human.lastName)  // Prints "Mead"
```

Documentation Links

- Getter
- Setter

Lesson 15: Fixing an Edge Case

In this video, you're going to address an edge case with our JavaScript code.

There are no notes for this lesson, as no new language features were explored.

Section 11. Asynchronous JavaScript

Lesson 1: Section Intro

In this section, you're going to explore asynchronous programming in JavaScript. The big picture goal is to figure out how we can communicate with servers via HTTP requests, allowing our scripts to be able to fetch and send data.

Lesson 2: HTTP Requests from JavaScript

In this lesson, you'll learn how to make HTTP requests from client-side JavaScript code. This is going to allow us to fetch a randomly generated puzzle for the hangman game.

XMLHttpRequest

Your browser makes HTTP requests when you visit a URL. The browser makes a request for the HTML page and any assets that the page needs. These assets might include images, JavaScript files, CSS files, and more.

You can also instruct your browser to make additional HTTP requests using JavaScript. This is done with `XMLHttpRequest`. While we can send and receive almost anything, it's typically to use `XMLHttpRequest` to send and receive JSON data.

The example below uses!! `XMLHttpRequest` to fetch a new puzzle for the hangman game. You start by creating a new request and configuring it. In this case we're making `GET` request to http://puzzle.mead.io/puzzle where we can get a new random puzzle.

Once the request is configured, you can send it off with `send` and wait for a response by listening for the "readystatechange" event. The `readyState` is `4` when the request is complete and the response is available. This is where you can parse the JSON into a JavaScript object and do something with the information.

```
// Making an HTTP request
const request = new XMLHttpRequest()

request.addEventListener('readystatechange', (e) => {
    if (e.target.readyState === 4) {
        const data = JSON.parse(e.target.responseText)
        console.log(data) // Will print a new random puzzle
    }
})

request.open('GET', 'http://puzzle.mead.io/puzzle')
request.send()
```

Documentation Links

- XMLHttpRequest

Lesson 3: HTTP Headers and Errors

In this lesson, you're going to dive into the details of an HTTP request. You'll dissect a request to see how the request and response are exchanged.

HTTP Status Codes

Every HTTP response includes a status code. The status code is a number that signifies if a given request was successful. As an example, the status code `200` represents success while the status code `400` signifies the HTTP request contained an error. Check the documentation link below for a complete list of status codes.

The status code for your HTTP requests lives on the **`status`** property. The code below checks if the **`status`** was `200` before assuming that the requested succeeded. If the status code was `200`, the response is parsed and the object is printed. If the status code was not `200`, and error message was printed.

```
// Making an HTTP request
const request = new XMLHttpRequest()

request.addEventListener('readystatechange', (e) => {
   if (e.target.readyState === 4 && e.target.status === 200) {
       const data = JSON.parse(e.target.responseText)
       console.log(data)
   } else if (e.target.readyState === 4) {
       console.log('An error has taken place')
   }
})

request.open('GET', 'http://puzzle.mead.io/puzzle?wordCount=3')
request.send()
```

Documentation Links

- HTTP Status Codes

Lesson 4: Exploring Another API

This challenge lesson is going to require you to use `XMLHttpRequest` to make a request to a new URL.

There are no notes for this lesson, as no new language features were explored.

Lesson 5: Callback Abstraction

In this lesson, you're going to learn how to use callbacks to simplify the code that makes our HTTP requests. You'll learn how to abstract the complexity of `XMLHttpRequest` into a simple function call.

Callback Abstraction

You've seen how to make a single HTTP request, but imagine you needed to get two new puzzles. How would you set up that code? You could duplicate the code for making the request by copying and pasting it into your script. This would work, but it wouldn't be DRY (Don't Repeat Yourself).

It would be better to create a function we can call that makes the request and gives us back the response. If we wanted to get two puzzles we'd just need to call the function twice. This is exactly what the example below does.

The important thing to note is that we pass a callback into `getPuzzle` when we call it. `getPuzzle` then makes the HTTP request and calls our callback function once it gets the

response. The callback gets called with one of two arguments. If the request was a failure, you call the callback with an error as the first argument. If the request was a success, you call the callback with passing the data in as the second argument.

You could fetch a second puzzle by calling `getPuzzle` again. There would be no need to duplicate the complex code that makes the request.

```
const getPuzzle = (callback) => {
    const request = new XMLHttpRequest()

    request.addEventListener('readystatechange', (e) => {
        if (e.target.readyState === 4 && e.target.status === 200) {
            const data = JSON.parse(e.target.responseText)
            callback(undefined, data.puzzle)
        } else if (e.target.readyState === 4) {
            callback('An error has taken place', undefined)
        }
    })

    request.open('GET', 'http://puzzle.mead.io/puzzle?wordCount=3')
    request.send()
}

getPuzzle((error, puzzle) => {
    if (error) {
        console.log(`Error: ${error}`)
    } else {
        console.log(puzzle)
    }
})
```

Documentation Links

- Callback functions
- Abstraction (software)

Lesson 6: Asynchronous vs. Synchronous Execution

This lesson explores the differences between asynchronous and synchronous execution via a visualization.

There are no notes for this lesson, as no new language features were explored. Check out the visualization for details.

Lesson 7: Callback Abstraction Challenge

In this challenge lesson, you're going to create a `getCountry` function to abstract away the details of working with the REST Countries API.

There are no notes for this lesson, as no new language features were explored.

Lesson 8: Closures

In this lesson, you're going to learn about closures. This is a feature of that language that you've already been using, but defining and exploring them is going to help explain why the callback pattern works.

Closures by Example

A closure is the combination of a function and lexical scope in which the function was defined. A function has access to its lexical scope even in situations where it might not be obvious. Take a look at the following example.

```
const createCounter = () => {
    let count = 0

    return () => {
        return count
    }
}

const counter = createCounter()
console.log(counter()) // Will print 0
```

The above code will print `0`, but that might come as a surprise. The `createCounter` function has a local variable `count`. That local variable `count` is accessible by the inner function even after `createCounter` has completed. `count` was initialized by `createCounter`, but it lives on in the lexical scope for our inner function.

The callback pattern wouldn't work if closures didn't exist.

Currying with Closures

Closures can be used to perform currying. Currying is the process of converting a function with several arguments into a sequence of functions that each takes a single argument. The benefits of currying are best illustrated with an example.

```
const createTipper = (baseTip) => {
    return (amount) => {
        return baseTip * amount
    }
}
const tip20 = createTipper(.2)
const tip30 = createTipper(.3)
console.log(tip20(100))
console.log(tip20(80))
console.log(tip30(100))
```

The example above uses currying to create a tip calculator. Notice that there isn't a single function that takes two arguments for the bill and the tip percent. What we have instead is a function `createTipper` that returns a function. `createTipper` gets the tip percent via a single argument. It returns a function that takes in the total bill. The closure created by the inner function gives it access to `baseTip` and `amount` for the calculation.

The currying pattern turned `createTipper` into a function generator. It generates functions that calculate the tip at a fixed tip percent.

Documentation Links

- Closures
- Currying

Lesson 9: Exploring Promises

In this lesson, you're going to learn about promises. The Promise API is an alternative to callback functions which makes it easier to manage complex asynchronous code.

Creating a Promise

You can create a promise by using the `new` operator with the `Promise` constructor function. The constructor function requires a function as its one and only argument. In this function, where we can perform our long running asynchronous task.

A promise can be in one of three states, pending, fulfilled, or rejected. A promise starts off in the pending state. It's up to our code to determine if the promise should be fulfilled (considered a success) or rejected (considered a failure). Our function gets called with two arguments, `resolve`, and `reject`. You call resolve to fulfill the promise and reject to reject the promise.

A promise is considered settled when it's not pending. A settled promise is one that's been fulfilled or rejected.

Both `resolve` and `reject` take a single optional argument. Imagine that a request succeeds and you get the data you asked for. You'd call `resolve` with that data. You'd call `reject` with the error if the request did not succeed. A promises is locked once it's been fulfilled or rejected. It can't be fulfilled or rejected again.

Here's an example of a promise that waits 2 seconds and then get fulfilled with a simple string message.

```
const myPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve('Example data')
    }, 2000)
})
```

Working with a Promise

Creating a promise gets us halfway there. We still need a way to be able to do something with the data or error when the promise is either fulfilled or rejected.

You can run some code when the promise is fulfilled or rejected by calling `then` on the promise. The first argument is a function to run when the promise is fulfilled. It's called with whatever value was passed to `resolve`. The second argument is a function to run if the promise is rejected. It's called with the error that was passed to `reject`.

```
// Using myPromise from above

myPromise.then((data) => {
    console.log(data) // Will print "Example data"
}, (err) => {
    console.log(err)
})
```

Documentation Links

- Promise
- JavaScript Promises (blog post)

Lesson 10: Converting to Promises

In this challenge lesson, it's up to you to switch from the callback pattern to promises.

There are no notes for this lesson, as no new language features were explored.

Lesson 11: Promise Chaining

In this lesson, you're going to explore promise chaining. Promise chaining makes it easy to link together multiple promises and do one thing after something else has finished.

Chaining

Let's start with a function that returns a new promise. Below is `getDataPromise`. It accepts a number and returns a new promise. The promise is resolved after two seconds with a number. The resolved value is just the initial value times 2.

```
const getDataPromise = (num) => new Promise((resolve, reject) => {
    setTimeout(() => {
        typeof num === 'number' ? resolve(num * 2) : reject('Number must be
provided')
    }, 2000)
})
```

You could call this function once to double a number, but what if you wanted to double a number twice? The following would work, but is not ideal. Notice the complex nesting and the duplicate error handler.

```
getDataPromise(10).then((data) => {
    getDataPromise(data).then((data) => {
        console.log(data) // Will print "40"
    }).catch((err) => {
        console.log(err)
    })
}).catch((err) => {
    console.log(err)
})
```

The better way to structure this is using chaining. When you return a promise from the `then` callback, you're able to chain on another `then` call to do something when the second promise is fulfilled.

```
getDataPromise(10).then((data) => {
    return getDataPromise(data)
}).then((data) => {
    console.log(data) // Will print "40"
}).catch((err) => {
    console.log(err)
})
```

Another important thing to notice is that our `then` calls don't define error handlers. Instead, a single call to `catch` is added onto the end of the chain. This allows you to set up a function to run if any promise in the chain is rejected.

Lesson 12: The Fetch API

In this lesson, you're going to learn about the fetch API. The Fetch API provides a simpler alternative to XMLHttpRequest.

Using Fetch

A big reason to use `fetch` is that it's promise-based. The return value for `fetch` is a promise which makes it much easier to integrate into modern promise-based code. There's no need to manually create a new promise and there is no need to call `resolve` or `reject`.

The example below uses the Fetch API to get a new hangman puzzle and print it to the console. The return value for `fetch` is a promise so you can directly attach `then` and `catch` method calls. If the request completes, the promise gets resolved with the `response`, giving you access to the status code and JSON data. The promise will be rejected with an error if the request fails.

```
fetch('http://puzzle.mead.io/puzzle', {}).then((response) => {
    if (response.status === 200) {
        return response.json()
    } else {
        throw new Error('Unable to fetch the puzzle')
    }
}).then((data) => {
    console.log(data.puzzle)
}).catch((error) => {
    console.log(error)
})
```

The biggest difference is that `fetch` doesn't give you direct access to `responseText`. What you get is a `json` method which returns a promise. That promise resolves with the parsed JSON data.

Documentation Links

- Fetch API
- Response

Lesson 13: A Fetch Challenge

In this challenge lesson, it's up to you to make an HTTP request with the Fetch API.

There are no notes for this lesson, as no new language features were explored.

Lesson 14: A Promise Challenge

In this challenge lesson, it's up to you to use promise chaining with a brand new HTTP request.

There are no notes for this lesson, as no new language features were explored.

Lesson 15: Async/Await

In this lesson, you're going to learn about async functions and the await operator, also known as async/await. Async/await allows you to create asynchronous promise-based code that looks like standard synchronous code.

Async Functions

When defining a function, you can use the `async` keyword to create an async function. Async functions are special because they always return a promise. This promise is going

to resolve with whatever value you return from the function. The promise would reject if the function throws an error.

Here's an example of an async function that returns a number. Notice that the actual return value is a promise that resolves with that number.

```
const processData = async () => {
  return 101
}

processData().then((data) => {
  console.log(data) // Will print "101"
}).catch((error) => {
  console.log(error)
})
```

Here's an example of an async function that throws an error. Notice that the actual return value is a promise that rejects with the thrown error.

```
const processData = async () => {
  throw new Error('Something went wrong!')
  return 101
}

processData().then((data) => {
  console.log(data)
}).catch((error) => {
  console.log(error) // Will print error object
})
```

Await Operator

Async functions become useful when we pair them with the await operator. The await operator is a special operator that can only be used inside of async functions. The await operator allows you to pause the execution of a function and await the promise being settled.

The below example uses `await` with an async version of `getPuzzle`. There are two instances of `await`, the first is just before `fetch` and the second is just before `response.json`. In both cases, the `getPuzzle` function pauses and waits for the promise to settle.

If a promise is fulfilled, the resolved value is returned from the await operation. If a promise is rejected, the rejected value is thrown as an error. This allows us to create synchronous looking code using asynchronous language features.

```
const getPuzzle = async (wordCount) => {
    const response = await
fetch(`http://puzzle.mead.io/puzzle?wordCount=${wordCount}`)

    if (response.status === 200) {
        const data = await response.json()
        return data.puzzle
    } else {
        throw new Error('Unable to get puzzle')
    }
}

getPuzzle('2').then((puzzle) => {
    console.log(puzzle)
}).catch((err) => {
    console.log(`Error: ${err}`)
})
```

Documentation Links

- Async Function
- Await Operator

Lesson 16: Async/Await Challenge

In this challenge video, you'll need to use async/await to make our application requests.

There are no notes for this lesson, as no new language features were explored.

Lesson 17: Integrating Data into the Application

In this video, you're going to wrap up the functionality for the hangman game.

There are no notes for this lesson, as no new language features were explored.

Section 14. Cutting-Edge JavaScript with Babel and Webpack

Lesson 1: Section Intro

In this section, you're going to explore cutting-edge JavaScript features. You'll also learn about Babel and webpack, two tools that'll make it easier to create real-world JavaScript applications that work everywhere.

Lesson 2: The Problem: Cross-Browser Compatibility

In this video, you're going to learn about cross-browser compatibility. This refers to the ability of a web application to work in a wide range of environments. You can't control what browser or operating system a visitor is using, so it's important to make sure the code you write works in as many browsers as possible.

There are no notes for this lesson, as no new language features were explored.

Lesson 3: Exploring Babel

In this video, you're going to set up Babel. Babel is a compiler for JavaScript. It's going to solve all our cross-browser compatibility issues.

Babel

Imagine you're working on a new application. You write some code using arrow functions and the new ES6 method syntax. After opening the app in the browser, you're greeted with a white screen and some errors in the console. The problem? A syntax error. The browser doesn't support those features and never will.

You could avoid using arrow functions and ES6 methods in your code, but that's not ideal. We want to be able to take advantage of new language features while still supporting old browsers. The solution is to pass your code through Babel.

When you run Babel, it reads your application code. It then converts your code by swapping out cutting-edge features for more standardized features. The converted code is functionally equivalent to the original code. The only difference is that the converted code was optimized to work in almost every browser.

This is easiest to understand with an example. The script below uses a couple newer JavaScript features. It takes advantage of the ES6 method syntax and the property

shorthand. This code might run fine in the latest version of Chrome, but it's not going to work well for all users.

```
// index.js file
const name = 'Andrew'
const person = {
  name,
  getName() {
    return this.name
  }
}

console.log(person.getName())
```

If you pass that script through Babel, you'll get the script below. These two scripts are functionally equivalent, but Babel has stripped out newer features and replaced them with better supported features.

```
// bundle.js file
'use strict';

var name = 'Andrew';
var person = {
  name: name,
  getName: function getName() {
    return this.name;
  }
};

console.log(person.getName());
```

Setting up Babel

Both Babel and webpack require a bit configuration. First up, let's set up our project.

```
# Setup a boilerplate package.json file
npm init -y


# Install babel so we can run our command
npm install -g babel-cli@6.26.0


# Install a babel preset that tells babel what to change
npm install babel-preset-env@1.6.1
```

With everything installed, the following command will process a file called index.js and save the compiled output in a file named output.js.

```
babel input.js --out-file output.js --presets env
```

Documentation Links

- Babel

Lesson 4: Setting up Our Boilerplate

In this video, you're going to set up the boilerplate project to serve up the compiled JavaScript code to the browser.

npm Scripts

The command to run your code through Babel is getting long. There's nothing wrong with that, but it gets more annoying to type out as it grows. Luckily, npm gives you a way to set up an alias using the "scripts" property in package.json.

The example below moves the Babel command into a "build" script.

```
// package.json
// Other properties removed to keep things short
{
  "scripts": {
    "build": "babel src/index.js --out-file public/scripts/bundle.js --
presets env --watch"
  }
}
```

You can execute a script by its name using `npm run <name>` from the command line.

```
npm run build
```

Lesson 5: Avoiding Global Modules

In this video, you're going to learn why relying on global npm modules is generally a bad idea. You'll be uninstalling the global npm modules in favor of local npm modules.

App Dependencies

The downside of using global modules is that they're not a project dependency that's listed in package.json. That means it's not enough for someone to run `npm install`, they'd also need to figure out what global modules are needed and install them separately; otherwise, the build script will fail because babel-cli won't be installed.

It's a best practice to avoid global modules and install all dependencies needed by a project as local modules.

Local module can still be used from npm scripts like we do with global modules. You won't be able to access them directly from the command line, but it's still valid to reference them in a script.

The package.json file below shows what this setup would look like. Notice both babel-cli and live-server are now local modules.

```
// Other properties removed to keep things short
{
  "scripts": {
    "serve": "live-server public",
    "build": "babel src/index.js --out-file public/scripts/bundle.js --
presets env --watch"
  },
  "dependencies": {
    "babel-cli": "^6.26.0",
    "babel-preset-env": "^1.6.1",
    "live-server": "^1.2.0"
  }
}
```

Lesson 6: Exploring Webpack

In this video, you're going to learn about webpack. We're not going to install it yet, but we will cover what it does and why it's useful.

This is a presentation video. Please refer to the video for the visualization.

Lesson 7: Setting up Webpack

In this video, you're going to set up webpack. This will allow us to take advantage of JavaScript modules.

Webpack

Like Babel, webpack is a build tool. You pass it through JavaScript code you wrote, and it outputs a modified version of it. The outputted code is the code that you run in the browser.

To set it up, you'll need to install webpack and webpack-cli. Then you can create a script that runs the webpack command.

```
// Other properties removed to keep things short
{
  "scripts": {
    "serve": "live-server public",
    "webpack": "webpack",
    "build": "babel src/index.js --out-file public/scripts/bundle.js --
presets env --watch"
  },
  "dependencies": {
    "babel-cli": "^6.26.0",
    "babel-preset-env": "^1.6.1",
    "live-server": "^1.2.0",
    "webpack": "^4.5.0",
    "webpack-cli": "^2.0.14"
  }
}
```

Webpack requires a bit more configuration than Babel. Because of this, webpack allows us to create a configuration file in our project as opposed to adding on a bunch of command line arguments. The file needs to be called webpack.config.js, and it should live in the root of your project alongside package.json.

Below is a great starter configuration. This tells webpack where to find our code, and where to save the outputted file.

```
const path = require('path')

module.exports = {
    entry: './src/index.js',
    output: {
        path: path.resolve(__dirname, 'public/scripts'),
        filename: 'bundle.js'
    }
}
```

Lesson 8: JavaScript Modules: Part I

In this video, you're going to learn about JavaScript modules. JavaScript modules make it easy to share code between files, allowing us to create more advanced applications without messy code.

Exporting

There are two sides to the module system. There's a file that sets up some exports, and then there are one or more files that import the exported functionality. You might have a file called add.js that exports a function for adding numbers. Any file in the program that needs to add numbers can then import the exported function from add.js.

It's important to note that each file, which could also be referred to as a module, has its own scope. That means exporting and importing is the only way to share code.

The file below demonstrates this by setting up two functions and exporting them using the `export` statement. It exports a function for adding two numbers and a function for subtracting two numbers. You can export as many things as you need; in this case, two exports is all that's necessary.

```
// utilities.js
export const add = (a, b) => a + b

export const subtract = (a, b) => a - b
```

Importing

The `import` statement is used to grab exports from another file or module. The above file already sets up some exports, so these can be used by another file as shown in the code below.

```
// index.js
import { add, subtract } from './utilities'

console.log(add(32, 1)) // Will print: 33
console.log(subtractadd(32, 1)) // Will print: 31
```

Documentation Links

- Import statement
- Export statement

Lesson 9: JavaScript Modules: Part II

In this video, you're going to continue exploring the module system provided by webpack.

Default Exports

A module can export as many named exports as needed. A module can also choose to set up a single default export. This is completely optional, but it provides a nice way to set up imports and exports if a file exports just one thing.

The code below now exports three functions. `add` and `subtract` are set up as named exports. `square` is set up as the default export.

```
// utilities.js
const add = (a, b) => a + b
const subtract = (a, b) => a - b
const square = (x) => x * x

export { add, subtract, square as default }
```

Grabbing a default export requires a small change to the import statement. Instead of naming "square" in the curly braces, it gets set up just before the curly braces. You can also name it anything you like. It's not the name that links it to the correct export; it's the fact that it's the default export.

The code below grabs the `square` export and stores it as `otherSquare`. It also grabs the named export `add`.

```
// index.js
import otherSquare, { add } from './utilities'

console.log(add(32, 1)) // Will print: 33
console.log(otherSquare(10)) // Will print: 100
```

Documentation Links

- Import statement
- Export statement

Lesson 10: Adding Babel into Webpack

In this video, you're going to configure webpack to use Babel. This will give you all the advantages of webpack, as well as all the advantages of Babel.

Module Loaders

Setting up webpack to use Babel requires a module loader. A module loader lets you customize how webpack processes file in your application. Using babel-loader, webpack can ensure all our code gets compiled by Babel.

The config below shows how babel-loader gets set up. The `test` and `exclude` properties lets you target your JavaScript files. The `use` property lets you set up the babel loader as well as the env preset.

```
npm install babel-loader@7.1.4


const path = require('path')

// Other properties removed to keep things short
module.exports = {
    module: {
        rules: [{
            test: /\.js$/,
            exclude: /node_modules/,
            use: {
                loader: 'babel-loader',
                options: {
                    presets: ['env']
                }
            }
        }]
    }
}
```

Webpack is now running Babel for us, so there's no longer a need for a script that explicitly runs babel. The necessary scripts are shown below.

```
// Other properties removed to keep things short
{

  "scripts": {
    "serve": "live-server public",
    "build": "webpack"
  }
}
```

Documentation Links

- Webpack loaders

Lesson 11: The Webpack Development Server

In this video, you're going to set up webpack-dev-server. This will simplify the development environment and let you start everything up with a single command.

Webpack-Dev-Server

With the current setup, it requires two commands to start things up. The serve script is required to serve up the public folder. The build script is required so webpack can

process our files. The development server gives us a way to do both with a single command.

First up, you need to install webpack-dev-server and configure a script to run it.

```
npm install webpack-dev-server@3.1.3


// Other properties removed to keep things short
{
  "scripts": {
    "dev-server": "webpack-dev-server"
  }
}
```

Next up, you need to update the config file to tell the development server where it can find the folder to serve up and where it should serve up the webpack assets from.

```
const path = require('path')

// Other properties removed to keep things short
module.exports = {
    devServer: {
        contentBase: path.resolve(__dirname, 'public'),
        publicPath: '/scripts/'
    }
}
```

Now, you can run `npm run dev-server` and start working on your application!

Documentation Links

- Dev server

Lesson 12: Environments and Source Maps

In this video, you're going to set up webpack for production. This will let you create an optimized version of our bundle that's better suited for production.

Production Builds

Setting up webpack for production couldn't be easier. All you need to do is set up the `--mode` command line argument to either `development` or `production`. That's it!

The mode argument works with `webpack` and with `webpack-dev-server`.

```
// Other properties have been hidden to keep things sort
{
  "scripts": {
    "dev-server": "webpack-dev-server --mode development",
    "build": "webpack --mode production"
  }
}
```

If the production builder is smaller and faster, why not always run webpack in production mode? The reason is that generating this build takes way more time. That means you wouldn't see your changes reflected in the browser right away when working locally on your app.

Source Maps

Webpack dumps everything into a single file. This can be annoying when trying to use the browser developer tools to track down a bug or fix an error. All the error messages come from bundle.js, making it difficult to determine where you need to make code changes. The issue might be in remote.js in the src directory, but it will be reported as coming from bundle.js.

All this can be fixed with source maps. Source maps give additional information to the browser, allowing it to trace logs or errors back to the original uncompiled source.

The config below shows how you can set up source maps for your application.

```
const path = require('path')

// Other properties have been hidden to keep things sort
module.exports = {
    devtool: 'source-map'
}
```

Lesson 13: Converting Hangman App

In this video, you're going to start converting the Hangman application over to our boilerplate project.

Babel Polyfill

As we've seen, Babel is great at compiling our code. It can convert code using a newer feature like the class syntax down to code using better supported syntaxes.

What Babel won't do is add support for things that don't require syntax transformations. For example, there are new array methods like `includes` that are not supported by older browser. Support can be added for this with babel-polyfill.

First up, install the polyfill.

```
npm install babel-polyfill@6.26.0
```

Next, customize the `entry` point of the webpack config. With an array value, you can set up multiple modules to be loaded.

```
const path = require('path')

// Other properties removed to keep things short
module.exports = {
    entry: ['babel-polyfill', './src/index.js']
}
```

Lesson 14: Using Third-Party Libraries

In this video, you're going to learn how to use npm modules in your client side JavaScript code.

npm Modules in the Browser

webpack allows you to import third-party modules as long as they've been installed as local dependencies. This gives you a better way to manage the third-party libraries your application depends on.

First up, you'll need to install the module.

```
npm install validator@9.4.1
```

Next, you'll need to import something from the module. Refer to the library documentation to learn what a give module exports. The code below grabs the default export and accesses the `isEmail` method.

```
import validator from 'validator'

const isValidEmail = validator.isEmail('test@example.com')
console.log(isValidEmail) // Will print true
```

Using npm modules lets you solve common problems without having to write unnecessary code. This frees up time for working on features specific to your application. Validating an email is a common problem that needs to be solved by almost every application out there. There's no need to recreate the wheel and try to write your own validator where there are battle tested solutions out there.

Lesson 15: Converting Notes App: Part I

In this video, you're going to start converting the notes application to use our boilerplate project with Babel and webpack.

There are no notes for this lesson, as no new language features were explored.

Lesson 16: Converting Notes App: Part II

In this video, you're going to continue converting the notes application.

There are no notes for this lesson, as no new language features were explored.

Lesson 17: Converting Notes App: Part III

In this video, you're going to continue converting the notes application.

There are no notes for this lesson, as no new language features were explored.

Lesson 18: Converting Notes App: Part IV

In this video, you're going to wrap up the conversion process for the notes application.

There are no notes for this lesson, as no new language features were explored.

Lesson 19: To-Do App Conversion Setup

In this video, I'm going to walk you through the to-do app challenge. It'll be up to you to convert the application over to the new boilerplate project structure.

There are no notes for this lesson, as no new language features were explored.

Lesson 20: Converting To-Do App: Part I

In this video, we'll start going through the solution for the to-do app challenge.

There are no notes for this lesson, as no new language features were explored.

Lesson 21: Converting To-Do App: Part II

In this video, we'll wrap up the solution for the to-do app challenge.

There are no notes for this lesson, as no new language features were explored.

Lesson 22: The Rest Parameter

In this video, you're going to learn about the rest parameter. The rest parameter lets you access a set of the function arguments as an array.

The Rest Parameter

The rest parameter lets you access a set of the function arguments as an array. This is an alternative to using an array as a function argument.

In the example below, the rest parameter is used to grab all the function arguments and store them in the `numbers` array. This gives you a flexible function that can calculate averages regardless of how many numbers are passed in.

```
const calculateAverage = (...numbers) => {
    let sum = 0
    numbers.forEach((num) => sum += num)
    return sum / numbers.length
}

console.log(calculateAverage(0, 100, 88, 64)) // Will print: 63
```

The rest parameter can be used with regular functions or arrow functions. It can be used alongside other arguments as well. The code below does just that. The first argument is the team name which is pulled out into its own variable `teamName`. The second argument is the coach's name and that's also pulled out into its own variable. Lastly, the rest parameter is used to grab all other arguments and store them in the `players` array.

```
const printTeam = (teamName, coach, ...players) => {
    console.log(`Team: ${teamName}`)
    console.log(`Coach: ${coach}`)
    console.log(`Players: ${players.join(', ')}`)
}

printTeam('Liberty', 'Casey Penn', 'Marge', 'Aiden', 'Herbert', 'Sherry')
```

Documentation Links

- Rest parameter

Lesson 23: The Spread Syntax

In this video, you're going to learn about the spread syntax. The spread syntax gives you a way to expand an array into it's individual values. This can be used when calling a function or when creating an array.

The Spread Syntax

The spread syntax can be used when calling a function. This allows you to spread out an array into its individual values and pass those values in as individual function arguments.

The code below demonstrates this. The data you have access to has the players stored in an array. The problem is that `printTeam` doesn't expect an array argument. It expects the players as individual arguments. Using the spread syntax, you can call printTeam with the correct arguments.

```
const printTeam = (teamName, coach, ...players) => {
    console.log(`Team: ${teamName}`)
    console.log(`Coach: ${coach}`)
    console.log(`Players: ${players.join(', ')}`)
}

const team = {
    name: 'Libery',
    coach: 'Casey Penn',
    players: ['Marge', 'Aiden', 'Herbert', 'Sherry']
}
printTeam(team.name, team.coach, ...team.players)
```

The spread syntax can also be used when creating an array. It's useful for adding items to an array or copying arrays. The code below uses the spread syntax to clone an array and add a new item onto it.

```
let cities = ['Barcelona', 'Cape Town', 'Bordeaux']
let citiesClone = [...cities, 'Santiago']
console.log(cities) // Will print three citites
console.log(citiesClone) // Will print four citites
```

Documentation Links

- Spread

Lesson 24: The Object Spread Syntax

In this video, you're going to learn how to use the spread operator with objects. This gives you an easy way to create, clone, and merge objects.

Setting up Support

The spread syntax is not supported on objects by default. Babel requires an additional plugin to get support for this.

First up, install the Babel plugin.

```
npm install babel-plugin-transform-object-rest-spread@6.26.0
```

Next, add the plugin to the plugins array for Babel.

```
const path = require('path')

// Other properties removed to keep things short
module.exports = {
    module: {
        rules: [{
            test: /\.js$/,
            exclude: /node_modules/,
            use: {
                loader: 'babel-loader',
                options: {
                    presets: ['env'],
                    plugins: ['transform-object-rest-spread']
                }
            }
        }]
    }
}
```

The Object Spread Syntax

The spread syntax can be used when creating a new object. It gives you an easy way to create, clone, or merge objects.

The code below shows how the spread syntax can be used to clone an object.

```
let house = {
    bedrooms: 2,
    bathrooms: 1.5,
    yearBuilt: 2017
}
let clone = {
    ...house
}

console.log(house) // Will print the same as clone
console.log(clone) // Will print the same as house
```

The code below shows how to use the spread syntax to merge two objects while overriding some of their properties.

```
const person = {
    name: 'Andrew',
    age: 27
}
const location = {
    city: 'Philadelphia',
    country: 'USA'
}
const overview = {
    ...person,
    ...location,
    name: 'Mike'
}
console.log(overview)

// The above code will print the following
// {
//   age: 27,
//   city: "Philadelphia",
//   country: "USA",
//   name: "Mike"
// }
```

Documentation Links

- Spread

Lesson 25: Destructuring

In this video, you're going to learn about destructuring. Destructuring lets you pull values out of an object or array and into their own variable.

Destructuring Objects

In JavaScript, you'll often have an entire object passed to a function when the function only needs a couple values. Destructuring will let you pull properties off of objects. It'll also let you pull items out of arrays.

The code below shows how destructuring can be used to pull properties off of an object. It creates individual variables for the object properties that can then be accessed directly.

The code below also makes use of the rest parameter. `...other` will create a new object with all the properties that weren't destructured.

```
const todo = {
    id: 'asdfpoijwermasdf',
    text: 'Pay the bills',
    completed: false
}

const { text:todoText, completed, details = 'No details provided', ...others
} = todo

console.log(todoText) // Will print: "Pay the bills"
console.log(completed) // Will print: false
console.log(details) // Will print: "No details provided"
console.log(others) // Will print: { id: "asdfpoijwermasdf" }
```

Destructuring also works with function arguments. The `printTodo` function below expects an object argument. Instead of naming the argument `todo` and getting properties off of it, it's destructured in the function arguments to provide access to the `text` and `completed` properties as variables.

```
const todo = {
    id: 'asdfpoijwermasdf',
    text: 'Pay the bills',
    completed: false
}

const printTodo = ({ text, completed }) => {
    console.log(`${text}: ${completed}`)
}
printTodo(todo)
```

Destructuring Arrays

Destructuring works with arrays too. Arrays can be destructured on their own or as a function argument.

The code below demonstrates this by grabbing the first array value and storing it in `firstAge`. It then uses the rest parameter to grab the other values and store them in `otherAges`.

```
const age = [65, 0, 13]
const [firstAge, ...otherAges] = age

console.log(firstAge) // Will print: 65
console.log(otherAges) // Will print: [0, 13]
```

Documentation Links

- Destructuring