Notes

CS50's Mobile App Development with React Native on EdX

Sparsh Jain

January 20, 2021

Contents

Chapter 1

JavaScript Overview

1.1 Introduction

JavaScript is Interpreted!

- Each browser has its own JavaScript engine, which either interprets the code, or uses some sort of lazy compilation.
  - V8: Chrome and Node.js
  - SpiderMonkey: Firefox
  - JavaScriptCore: Safari
  - Chakra: Microsoft Edge/IE
- They each implement the ECMAScript standard, but may differ for anything not defined by the standard.

1.2 Syntax

Semicolons are optional!

```
// comments are prefixed with double slashes
/*
 * Multi-line comments look like this
 */

// camelCase is preferred
// double-quotes create strings
const firstName = "jordan";
```

```
9
10 // semicolons are optional
11 // single-quotes also create strings
12 const lastName = 'Hayashi'
13
14 // arrays can be declared inline
15 // arrays can have multiple types (more on types later)
16 const arr = [
17   'string',
18   42,
19   function() { console.log('hi') },
20 ]
21
22 // this returns the element at the 2nd index and invokes it
23 arr[2]()
24
25 // this will iterate through the array and console log each
    ↪  element
26 for (let i = 0; i < arr.length; i++) {
27   console.log(arr[i])
28 }
```

Program 1.1: JavaScript Syntax

1.3 Types

- Dynamic Typing
- Primitive Types (no methods, immutable)
  - undefined
  - null
  - boolean
  - number
  - string
  - (symbol) (New, in ES6)
- Objects
- Typecasting? Coercion.

- Explicit vs. Implicit Coercion
  - `const x = 42;`
  - `const explicit = String(x); // explicit === '42'`
  - `const implicit = x + ''; // implicit === '42'`
- == vs. ===
  - == coerces the types
  - === requires equivalent types
- Falsy values? return false when cast to boolean
  - undefined
  - null
  - false
  - +0, -0, NaN
  - ""
- Truthy?
  - []
  - {}
  - literally everything except falsy values

```
const x = 42

// get type by using "typeof"
console.log(typeof x)
console.log(typeof undefined)

// this may surprise you...
console.log(typeof null)
```

Program 1.2: JavaScript Types

1.4 Objects

- Everything (except primitive types) are objects
- Prototypal Inheritance

1.4.1 Primitives vs. Objects

- Primitives are immutable
- Objects are mutable and stored by reference
- Passing by reference vs. passing by value

```
const o = new Object()
o.firstName = 'Jordan'
o.lastName = 'Hayashi'
o.isTeaching = true
o.greet = function() { console.log('Hello!') }

console.log(JSON.stringify(o))

const o2 = {}
o2['firstName'] = 'Jordan'
const a = 'lastName'
o2[a] = 'Hayashi'

const o3 = {
  firstName: 'Jordan',
  lastName: 'Hayashi',
  greet: function() {
    console.log('hi')
  },
  address: {
    street: "Main st.",
    number: '111'
  }
}

// see 3-objectsMutation.js for more objects
```

Program 1.3: JavaScript Objects

```
const o = {
  a: 'a',
  b: 'b',
  obj: {
    key: 'key',
  },
}

const o2 = o

o2.a = 'new value'

// o and o2 reference the same object
console.log(o.a)

// this shallow-copies o into o3
const o3 = Object.assign({}, o)

// deep copy
function deepCopy(obj) {
  // check if vals are objects
  // if so, copy that object (deep copy)
  // else return the value
  const keys = Object.keys(obj)

  const newObject = {}

  for (let i = 0; i < keys.length; i++) {
    const key = keys[i]
    if (typeof obj[key] === 'object') {
      newObject[key] = deepCopy(obj[key])
    } else {
      newObject[key] = obj[key]
    }
  }

  return newObject
}

```

```
40 const o4 = deepCopy(o)
41
42 o.obj.key = 'new key!'
43 console.log(o4.obj.key)
```

Program 1.4: Object Mutation

1.5 Prototypal Inheritance

- Non-primitive types have a few properties/methods associated with them.
  - Array.prototype.push()
  - String.prototype.toUpperCase()
- Each object stores a reference to its prototype
- Properties/methods defined most tightly to the instance have priority
- Most primitive types have object wrappers
  - String()
  - Number()
  - Boolean()
  - Object()
  - (Symbol())
- JS will automatically "box" (wrap) primitive values so you have access to methods

```
42.toString()                 // Errors
const x = 42;
x.toString()                 // "42"
x.__proto__                        // [Number: 0]
x instanceof Number         // false
```

- Why use reference to a prototype?
- What's the alternative?
- What's the danger?

1.6 Scope

- Variable Lifetime
  - Lexical scoping (var): from when they're declared until when their function ends
  - Block scoping (const, let): until the next } is reached

```
// "var" is lexically scoped, meaning it exists from time of declaration to end of func
if (true) {
  var lexicallyScoped = 'This exists until the end of the function'
}

console.log(lexicallyScoped)

// "let" and "const" are block scoped
if (true) {
  let blockScoped = 'This exists until the next }'
  const alsoBlockScoped = 'As does this'
}

// this variable doesn't exist
console.log(typeof blockScoped)

thisIsAlsoAVariable = "hello"

const thisIsAConst = 50

// thisIsAConst++ // error!

const constObj = {}

// consts are still mutable
constObj.a = 'a'

let thisIsALet = 51
thisIsALet = 50

// let thisIsALet = 51 // errors!
```

```
32
33 var thisIsAVar = 50
34 thisIsAVar = 51
35 var thisIsAVar = 'new value!'
```

Program 1.5: Variable Scopes

- Hoisting (hoisting the definitions to the top)
  - Function definitions are hoisted, but not lexically-scoped initializations

```
1  // functions are hoisted
2  hoistedFunction()
3
4  // but only if they are declared as functions and not as
   ↪  variables initialized to
5  // anonymous functions
6  console.log("typeof butNotThis: " + typeof butNotThis)
7
8  function thisShouldWork() {
9      console.log("functions are hoisted")
10 }
11
12 var butNotThis = function() {
13     console.log("but variables aren't")
14 }
```

Program 1.6: Function Scopes

- Why? How? JavaScript Engine!

1.7 JavaScript Engine

- Before executing the code, the engine reads the entire file and will throw a syntax error if one is found.
  - Any function definitions will be saved in memory
  - Variable initializations will not be run, but lexically scoped variable names will be declared
- Execution Phase

1.8 The Global Object

- All variables and functions are actually parameters and methods on the global object
  - Browser global object is the ‘window’ object
  - Node.js global object is the ‘global’ object

1.9 Closures

- Functions that refer to variables declared by parent function
- Possible because of scoping

```
function makeFunctionArray() {
        const arr = []

        for (var i = 0; i < 5; i++) {
                arr.push(function() { console.log(i) })
        }

        return arr
}

const arr = makeFunctionArray()

arr[0]()
```

Program 1.7: JavaScript Closure

Chapter 2

JavaScript and ES6

2.1 ES5, ES6, ES2016, Es2017, ES.Next, ...

- ECMAScript (Specs) vs. JavaScript (Implementation)
- What do most environments support?
  - Assume env supports all of ES5
  - Transpilers (Babel, TypeScript, CoffeeScript, etc.) to use newer features and make it backwards compatible to ES5
- Which syntax should we use? Generally use the future syntax (either env will catch up, or transpile backwards)

2.2 Closures

- Functions that refer to variables declared by parent function still have access to those variables
- Possible because of JavaScript's scoping

```
1 function makeFunctionArray() {
2   const arr = []
3
4   for (var i = 0; i < 5; i++) {
5     arr.push(function () { console.log(i) })
6   }
7
```

```
  return arr
}

const functionArr = makeFunctionArray()

// we expect this to log 0, but it doesn't
functionArr[0]()
```

Program 2.1: Bug? due to closures

```
function makeHelloFunction() {
  var message = 'Hello!'

  function sayHello() {
    console.log(message)
  }

  return sayHello
}

const sayHello = makeHelloFunction()

// the variable called message is not in scope here
console.log('typeof message:', typeof message)
// but the function sayHello still references a variable called
 ↪  message
console.log(sayHello.toString())

// because of the closure, sayHello still has access to the
 ↪  variables within scope
// when it was declared
sayHello()
```

Program 2.2: Closure Example

2.3 Immediately Invoked Function Expression

- A function expression that gets invoked immediately
- Creates closure

- Doesn't add to or modify global object

```
// this creates the same closure as in 1-closureExample.js, but doesn't pollute
// the global scope with a function called makeHelloFunction like that example
const sayHello = (function () {
  var message = 'Hello!'

  function sayHello() {
    console.log(message)
  }

  return sayHello
})()

// IIFEs can also be used to create variables that are inaccessible from the global
// scope
const counter = (function() {
  let count = 0

  return {
    inc: function() { count = count + 1 },
    get: function() { console.log(count) },
  }
})()

counter.get()
counter.inc()
counter.get()
```

Program 2.3: Immediately Invoked Function Expression (IIFE)

```
// we can create a closure around each anonymous function pushed to the array by
// turning them into IIFEs
function makeFunctionArray() {
  const arr = []

```

```
  for (var i = 0; i < 5; i++) {
    arr.push((function (x) {
      return function () { console.log(x) }
    })(i))
  }

  return arr
}

const functionArr = makeFunctionArray()

// this now logs 0 as expected
functionArr[0]()
```

Program 2.4: IIFE and Closure

2.4 First Class Functions

- Functions are treated in the same way as any other value
  - Can be assigned to variables, array values, and object values
  - Can be passed as arguments to other functions
  - Can be returned from functions
- Allows for creation of higher order functions
  - Either take one or more functions as arguments or returns a function
  - map(), filter(), reduce()

```
// Higher Order Functions take funcs as args or return funcs
function map(arr, fn) {
  const newArr = []

  arr.forEach(function(val) {
    newArr.push(fn(val))
  })

  return newArr
```

```
}

function addOne(num) { return num + 1 }

const x = [0,1,2,3]

console.log(map(x, addOne))


function filter(arr, fn) {
  const newArr = []
  arr.forEach(val => {
    if (fn(val)) newArr.push(val)
  })

  return newArr
}

function reduce(arr, fn, initialVal) {
  let returnVal = initialVal

  arr.forEach(val => {
    returnVal = fn(returnVal, val)
  })

  return returnVal
}
```

Program 2.5: Higher Order Function

2.5 Synchronous? Async? Single-Threaded?

- JavaScript is a single-threaded, synchronous language
- A function that takes a long time to run will cause the page to become unresponsive

```
// this function will freeze a browser page if run in
 ↪  console
function hang(seconds = 5) {
  const doneAt = Date.now() + seconds * 1000
  while(Date.now() < doneAt) {}
}
```

Program 2.6: Synchronous JS

- JavaScript has functions that act asynchronously

```
function printOne() {
  console.log('one')
}

function printTwo() {
  console.log('two')
}

function printThree() {
  console.log('three')
}

// this may not print in the order that you expect,
 ↪  because of the way the JS
// function queue works
setTimeout(printOne, 1000)
setTimeout(printTwo, 0)
printThree()
```

Program 2.7: Async Functions

- How can it be both? Synchronous and Asynchronous?

2.6 Asynchronous JavaScript

- Execution Stack
- Browser APIs
- Function queue
- Event loop

2.6.1 Execution Stack

- Functions invoked by other functions get added to the call stack
- When functions complete, they are removed from the call stack and the frame below continues executing

```
// when errors are thrown, the entire callstack is logged
function addOne(num) {
  throw new Error('oh no, an error!')
}

function getNum() {
  return addOne(10)
}

function c() {
  console.log(getNum() + getNum())
}

c()
```

Program 2.8: Execution Stack

2.6.2 Asynchronous Functions

- setTimeout()
- XMLHttpRequest(), jQuery.ajax(), fetch()
- Database calls

```
// this will recurse infinitely
function recurse() {
  console.log('recursion!')
  return recurse()
}

// this wall cause a stack overflow
recurse()
```

Program 2.9: Overflow

2.6.3 Callbacks

- Control flow with asynchronous calls
- Execute function once asynchronous call returns value
  - Program doesn't have to halt and wait for the value

```
// this is a HOF that invokes the function argument on 1
function doSomethingWithOne(callback) {
  return callback(1)
}

doSomethingWithOne(console.log)

// this is the same thing, but done asynchronously
function doSomethingWithOneAsync(callback) {
  setTimeout(() => callback(1), 1000)
}

doSomethingWithOneAsync(console.log)

// this simulates a database call that returns an object
 ↪  representing a person
function getUserFromDatabase(callback) {
    // simulates getting data from db
    setTimeout(() => callback({firstName: 'Jordan', lastName:
     ↪  'Hayashi'}), 1000)
}
```

```
20
21 // this is a function that greets a user, which we pass as a
    ↪ callback to getUserFromDatabase
22 function greetUser(user) {
23   console.log('Hi, ' + user.firstName)
24 }
25
26 getUserFromDatabase(greetUser)
```

Program 2.10: Callbacks

Callback Hell

A big christmas tree of callbacks

```
1  // taken from a personal project of mine
2  //
    ↪ https://github.com/jhhayashi/coupon-api/blob/master/controllers/auth.js
3
4  function login(req, res, callback) {
5    User.findOne({email: req.body.email}, function(err, user) {
6      if (err) return callback(err)
7
8      user.comparePassword(req.body.password, (err, isMatch) => {
9        if (err) return callback(err)
10       if (!isMatch) return res.status(401).send('Incorrect
          ↪ password')
11
12       // add relevant data to token
13       const payload = {id: user._id, email: user.email}
14
15       jwt.sign(payload, config.secret, {}, function(err, token)
          ↪ {
16         if (err) return callback(err)
17
18         user.token = token
19         user.save((err) => {
20           if (err) return callback(err)
21           res.json({token})
22         })
23       })
```

```
    })
  })
}
```

Program 2.11: Callback Hell

2.6.4 Promises

- Alleviate "callback hell"
- Allows you to write code that assumes a value is returned within a success function
- Only needs a single error handler

```
// this doesn't actually do anything, it's just a demo of
 ↪  Promise syntax

const url = ''

fetch(url)
  .then(function(res) {
    return res.json()
  })
  .then(function(json) {
    return ({
      importantData: json.importantData,
    })
  })
  .then(function(data) {
    console.log(data)
  })
  .catch(function(err) {
    // handle error
  })
```

Program 2.12: Promises

Escape Callback Hell with Promises

```
function login(req, res, callback) {
  User.findOne({email: req.body.email})
    .then(function(user) {
      return user.comparePassword(req.body.password)
    })
    .then(function(isMatch) {
      // have to throw in order to break Promise chain
      if (!isMatch) {
        res.status(401).send('Incorrect password')
        throw {earlyExit: true}
      }
      const payload = {id: user._id, email: user.email}
      return jwt.sign(payload, config.secret, {})
    })
    .then(function(token) {
      user.token = token
      return user.save()
    })
    .then(function() {
      res.json({token})
    })
    .catch(function(err) {
      if (!err.earlyExit) callback(err)
    })
}
```

Program 2.13: Escape Callback Hell with Promises

2.6.5 Async/Await

- Introduced in ES2017
- Allows people to write async code as if it were synchronous

```
async function login(req, res, callback) {
  try {
    const user = await User.findOne({email: req.body.email})
    const isMatch = await
     ↪ user.comparePassword(req.body.password)
```

```
5
6       if (!isMatch) return res.status(401).send('Incorrect
        ↪  password')
7
8       const payload = {id: user._id, email: user.email}
9       const token = await jwt.sign(payload, config.secret, {})
10
11      user.token = token
12      const success = await user.save()
13
14      res.json({token})
15    } catch (err) {
16      callback(err)
17    }
18  }
```

Program 2.14: Async/Await solution to callbacks

2.7 this

- Refers to an object that's set at the creation of a new execution context (function invocation)
- In the global execution context, refers to global object
- If the function is called as a method of an object, 'this' is bound to the object the method is called on

Setting 'this' manually

- bind(), call(), apply()
- ES6 arrow notation

```
1  // NOTE: this doesn't work as a node script, since they are run
   ↪  as modules
2  // `this` in this case is equal to module.exports, which is an
   ↪  empty object
3  console.log(this)
4
```

```
// this logs the global object
function whatIsThis() {
  console.log(this)
}

whatIsThis()

// =====================================

const person = {
  name: 'Jordan',
  greet: function() { console.log('Hi, ' + this.name) }
}

person.greet() // Hi, Jordan

// =====================================

const friend = {
  name: 'David',
}

friend.greet = person.greet

friend.greet() // Hi, david

// =====================================

const greetPerson = person.greet

greetPerson() // Hi, undefined

// make greetPerson() work, but not in node
this.name = 'Global'

// browser console or node REPL: Hi, Global
// node script: Hi, undefined
greetPerson()

const reallyGreetPerson = person.greet.bind(person)
reallyGreetPerson() // Hi, Jordan
```

```
46
47 person.greet.call({name: 'Yowon'}) // Hi, Yowon
48 person.greet.apply({name: 'Raylen'}) // Hi, Raylen
49
50 // ===================================
51
52 const newPerson = {
53   name: 'Jordan',
54   // arrow notation binds `this` lexically
55   greet: () => console.log('Hi, ' + this.name)
56 }
57
58 newPerson.greet() // Hi, Global
59
60 // bound functions cannot be bound again
61 newPerson.greet.call(person) // Hi, Global
```

Program 2.15: this in JavaScript

2.8 Browsers and DOM

- Browsers render HTML to a webpage
- HTML defines a tree-like structure
- Browsers construct this tree in memory before painting the page
- Tree is called the Document Object Model
- The DOM can be modified using JavaScript

Appendices

List of Programs