Notes

CS50's Mobile App Development with React Native on EdX

Sparsh Jain

January 20, 2021

Contents

Chapter 1

JavaScript

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

Appendices

List of Programs