JavaScript

CSC309
Mark Kazakevich

1995

- The browser that dominated the market was **Netscape Navigator**
  - But with only static HTML pages, they were eager to improve the experience

- They decided they needed a **scripting language** that would let developers make the internet more **dynamic**

Javascript beginnings

- A Netscape employee name **Brendan Eich** made the first version in **10 days**.

- After a few iterations, they named the language **JavaScript** since Java was popular back then
  - Otherwise the languages don't have much connection

ECMAscript

- JavaScript (JS) was becoming popular

- A standard was created by **ECMA** for scripting languages; the standard was based on JS
    - ECMAscript (**ES**)
    - **ES** is the standard, **JS** *implements* that standard

ES over time

- As time went on, ES standard improved

- ES3 (1999) is the baseline for modern day Javascript

- A bunch of others, like Mozilla, started to work hard on ES5, which was released in 2009

ES6

- Although more versions of the standard have been released, we will mostly talk about new features up to **ES6** (2015)

- Browsers adopt new ES standards slowly, but ES6 is mostly completely adopted by modern browsers

Writing *Vanilla* JavaScript

Vanilla JS

- As we'll see later, JS is very extendable
  - Lots of libraries, plugins, etc.

- *Vanilla* is not a version of JS, it just means JS without any extra stuff

- It's important to learn it!
  - Having a good grounding in the features of JS helps understand all the libraries available

Basic JS

- Many things are as you would expect from your experience with Python, C and Java. You will **learn the syntax** for these as you start coding.
    - Variables
    - Functions
    - If-else, for, while
    - Strings, numbers, booleans, collections

- **In class**, we will focus on features that are not quite what you might expect

Variables and Scope

Variable Declarations

- In JS, we can **declare** variables using the **var** keyword
  - `var a;`
  - You can then **define** it: `a = 4;`
    - Or all at once as `var a = 4;`

- JS separates declaring and defining variables
  - Writing `b = 7;` and calling it a day is problematic…
    - …let's see some code.

Let’s run some JS and observe `var`

- We will run some some javascript code in the browser

- Modern browsers can run JS natively
  - JS console included

- We can link to our .js file in any HTML file using the `<script>` tag
  - HTTP gets .js files from server, but they run on client

Demo

Variable Scope with **var**

- Variables declared using var have **function** scope

- They can be accessed within the **function they are declared in**
  - This includes any other nested **{}** blocks like loops, if-statements, nested functions, etc.
    - This is known as **lexical** scope

var scope

```
function f() {
   var a = 3;

   if (true) {
       console.log(a)  // 3 lexical, function scope
   }

    console.log(a) // 3

}
```

var scope

```
function f() {

   if (true) {
      var a = 3;
      console.log(a)  // 3
   }

   console.log(a); // 3, function scope

}
```

Hoisting

- What about not getting an error when accessing variable/calling f^n before definition?

- Remember that var declarations and definitions are **separate**

- All var variable and function **declarations** are **‘hoisted’** up to the top of their function scope (or global scope if not in function)
    - Variable *definitions* stay in place

var Hoisting

```
console.log(a) // undefined, not error
var a = 3
```

What’s going on under the hood:

```
var a; // declaration of ‘a’ is hoisted
console.log(a);
a = 3; // definition executed later
```

var Hoisting inside function

```
function f() {

   if (true) {
      var a = 3;
      console.log(a)  // 3
  }

  console.log(a); // 3, function scope

}
```

var Hoisting inside function

```
function f() {
   var a; // declaration hoisted to top of scope (function)

   if (true) {
      a = 3; // definition stays in place
      console.log(a)  // 3
   }

   console.log(a); // 3, function scope

}
```

Hoisting of a function

```
f()  // ‘in f’    (called before definition)
function f() {
   console.log(‘in f’)
}
```

Entire function definition hoisted to top:

```
function f() {
   console.log(‘in f’)
}
f() // ‘in f’
```

For loop demo

Unexpected issues

- What happens when we just type `a = 7;`
  - (without `var`)

- Without var, there is no declaration to hoist

- Ends up in **global** scope
  - Now available to everyone in lexical scope
  - Hard to manage

Unexpected issues

- `“use strict”` at top of file will help you catch errors
  - Such as defining variables before declaring

```
“use strict”;
a = 3; // will cause error since not declared
```

Still, with `var` it’s not always easy...

Enter...**ES6**

- Two new ways to declare variables in ES6 (2015)
    - `let` and `const`

- Important difference: they have **block scope**

- Only the **current block** can access them
    - Lexical scope still applies
        - Any inner block can also access

let scope

```
function f() {
   let a = 3; // block scope

   if (condition) {
       console.log(a)  // 3 lexical, block scope
   }

    console.log(a) // 3

}
```

let scope

```
function f() {

    if (true) {
        let a = 3;
        console.log(a)  // block scope
    }

    console.log(a); // ERROR! a not defined

}
```

For loop `let` demo

`const`

- Const has same scope rules as `let`
- Used for variables that will not be re-assigned
- **Rule:** In this class (and everywhere else), **default to using** `const` unless you know you will have to re-assign a variable

Do not use **var**.

But know how it works for backwards-compatibility

let and const

```
const num = 100;

function logNum (times) {
  for (let i = 0; i < 5; i++) {
    console.log(num);
  }
  console.log(i); // error, i in block scope
}
```