CS 142 Discussion

JavaScript

Agenda

(1) Review of JavaScript

(2) Getting Started with Project 2

*Most focus on (1) because if you're comfortable with JavaScript, the project is simple; but only talking about the project doesn't cover everything there is to know about JavaScript.

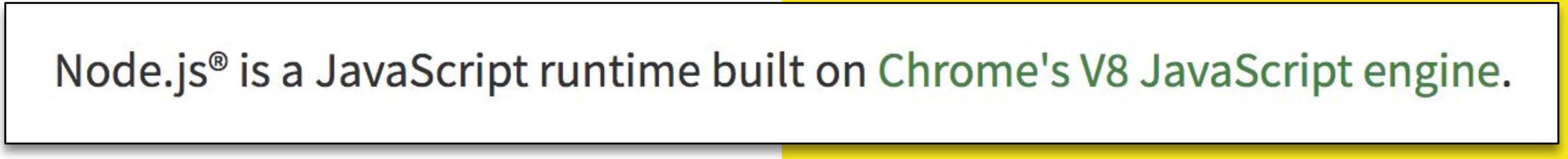
JavaScript

See also: *Eloquent JavaScript* by Marijn Haverbeke

Why JavaScript?

It's the language of the web:
Take it or leave it!

- Web browsers are based on JavaScript.
- We run some programs in Node.js, but Node.js is basically just the guts of Google Chrome running in the Terminal!
- JavaScript is used to interact with the HTML documents shown by web browsers.



Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine.

The Good, the Bad, and the Bizarre

(What makes JS programming special?)

❑ Loose / Dynamic Typing
❑ Dynamic Objects
❑ Prototype Inheritance
❑ First-Class Functions
❑ Function Scoping / Global Variables
❑ Callback Functions
❑ `"this"`

The Good, the Bad, and the Bizarre

(What makes JS programming special?)

People are better off ***not using*** some of the “features” of JavaScript.

Don’t do it!

(do eat your vegetables though)
(and also, use a linter)

The Good, the Bad, and the Bizarre

(What makes JS programming special?)

- ❑ **Loose / Dynamic Typing**
- ❑ **Dynamic Objects**
- ❑ **Prototype Inheritance**
- ❑ **First-Class Functions**
- ❑ **Function Scoping / Global Variables**
- ❑ **Callback Functions**
- ❑ `"this"`

Basics

- **Primitive Types:** number *(yup, just the one)*, string, boolean, null, undefined.
    - Everything else (including functions!) is an object.
- **Variables:** Dynamically typed. Hoisted.
    - `var x = 10;`
    - `x = "hello"`
- **Control Flow & Operators:** Similar to C / C++ / most other languages.
- **No block scoping with var!** (But let and const are block-scoped.)

```
for (var i = 1; i < 11; i++) {
     var j = i * 2;
}
console.log(j);
```

- **This means “global namespace” gets clogged real fast.**

Basics

- **Primitive Types:** number *(yup, just the one)*, string, boolean, null, undefined.
    - Everything else (including functions!) is an object.
- **Variables:** Dynamically typed. Hoisted.
    - `var x = 10;`
    - `x = "hello"`
- **Control Flow & Operators:** Similar to C / C++ / most other languages.
- **No block scoping with var!** (But let and const are block-scoped.)

```
for (var i = 1; i < 11; i++) {
     var j = i * 2;
}
console.log(j);
```

Don't use var!

(use let/const instead)

- **This means "global namespace" gets clogged real fast.**

Basics

- **“Falsy” values:** Evaluate to `false` if treated as boolean.
  - `undefined, '', 0, NaN, false, null`
- Everything else evaluates to true.
  - `if (0) console.log('Bummer');            // Nope.`
  - `if (10) console.log('Hooray');           >> Hooray`
- **Implicit Type Conversions:** JavaScript attempts to “coerce” an unexpected value type to the expected type.
  - Which actually can produce **unexpected results**! (Wat?!)
  - `console.log([] + []);                    // Prints empty string instead`
- **Comparisons:** Use `===` unless you want JS to do type conversion for you. `==` is generally not recommended because it can have behavior you might not anticipate; if you want type conversion you should do it yourself.
  - Similar for `!=` vs. `!==`
  - `5 == '5'                                 // true (?)`

Basics

- **"Falsy" values:** Evaluate to `false` if treated as boolean.
  - `undefined, '', 0, NaN, false, null`
- Everything else evaluates to true.
  - `if (0) console.log('Bummer');                   // Nope.`
  - `if (10) console.log('Hooray');                  >> Hooray`
- **Implicit Type Conversions:** JavaScript attempts to "coerce" an unexpected value type to the expected type.
  - Which actually can produce **unexpected results**! (Wat?!)
  - `console.log([] + []);                           // Prints empty string instead`
- **Comparisons:** Use === unless you want JS to do type conversion for you. == is generally not recommended because it can have behavior you might n anticipate; if you want type conversion you should do it yourself.
  - Similar for != vs. !==
  - `5 == '5'                                        // true (?)`

Be explicit! (next slide)

Never use == and !=
(Always use === and !==)

Explicit (aka Good) Type Conversions

- Boolean()
  - `Boolean(1)`, `Boolean([42, 'answer'])`
  - `Boolean(null)`, `Boolean('')`
  - Fancy (but unreadable) version: `!!thing` (= `Boolean(thing)`)
- Number()
  - `Number('123')`, `Number(false)` (⇒ 0), `Number(true)` (⇒ 1)
  - Fancy (but unreadable) version: `+thing` (= `Number(thing)`)
- String()
  - `String(1.000)` (⇒ `'1'`), `String('already a string')`
  - Fancy version (*that doesn't always work): `thing.toString()` (= `String(thing)`)

The Good, the Bad, and the Bizarre

(What makes JS programming special?)

- ❑ Loose / Dynamic Typing
- ❑ <u>Dynamic Objects</u>
- ❑ <u>Prototype Inheritance</u>
- ❑ First-Class Functions
- ❑ Function Scoping / Global Variables
- ❑ Callback Functions
- ❑ `"this"`

Objects

- **What is an object?**
  - Anything that's not a primitive. (Gee, thanks!)
  - *Mutable, keyed collection.* (Think Python dictionary! *)
- **Using objects**
  - `const student = {};`
  - `student.name = 'Miguel';`
  - `student.grade = 'A';`
    - `{name: 'Miguel', grade: 'A'}`
  - `delete student.grade;`
    - `{name: 'Miguel'}`
  - `student['grade'] = 'S';`
    - `{name: 'Miguel', grade: 'S'}`

Side note: "const" means you can't set the `students` *variable* to something else. But changing its *properties* is fine.

```
const students = {};
students.age = 42; // ✔
students = 42;     // ⛔
```

*: Not perfectly true. Instead, ES6 Map objects are closer.

Prototypes and Inheritance

- **Prototypes:** Every object has a "prototype." *What does it do?*
  - **When you attempt to access a property that does not exist in the object, JavaScript looks in the prototype.**
  - Relationship is **one-way**. *Editing an object doesn't change its prototype.*
  - Relationship is **dynamic**. *Updated prototype will immediately be reflected by all of its "children."*
- By default, all objects have `Object.prototype` as prototype. To change it:
  - `Object.setPrototypeOf(obj, myPrototype);`

Object Cookbook

- Check if an object has its *own* property (not in the prototype) with:

  -
    ```
    Object.prototype.hasOwnProperty.call(obj, 'property');  *
    ```
- Check if an object has *an* property (own *or* in the prototype) with:

  -
    ```
    'property' in obj
    ```
- Iterate through all *own* properties:

  -
    ```
    const keys = Object.keys(obj);
    for (let i = 0; i < keys.length; i++) {
      const key = keys[i]; const value = obj[key]; /* … */
    }
    ```

* Why not obj.hasOwnProperty('property')? ***Unsafe*** if someone somewhere set
obj['hasOwnProperty'] = something else

Object Cookbook

- Check if an object has its *own* property (not in the prototype) with:
  - `Object.prototype.hasOwnProperty.call(obj, 'property');` *
- Check if an object has *an* property (own *or* in the prototype) with:
  - `'property' in obj`
- Iterate through all *own* properties:
  - Alternative ES6 syntax:

    ```
    for (const key of Object.keys(obj)) {
      const value = obj[key]; /* … */
    }
    ```

This is a for-**of** loop, not a for-**in** loop. (Unlike Python.)

* Why not obj.hasOwnProperty('property')? ***Unsafe*** if someone somewhere set
obj['hasOwnProperty'] = something else

Time for some live-coding… please be kind.

The Good, the Bad, and the Bizarre

(What makes JS programming special?)

- ❑ Loose / Dynamic Typing
- ❑ Dynamic Objects
- ❑ Prototype Inheritance
- ❑ First-Class Functions
- ❑ Function Scoping / Global Variables
- ❑ Callback Functions
- ❑ `"this"`

Functions

- **Defining a function...**

```
function add(x, y) {    // give it a name
    return x + y;
}
```

OR

```
let add = function(x, y) {  // store it in a variable -- "first class"
    return x + y;
};
```

OR

```
(function(x, y) {       // anonymous boi 😎
    return x + y;
})(2, 3);               // what will this return?
```

Functions

- **Function scoping:** Inner functions can see things from outer functions, but outer functions cannot see things from inner functions (unless they are returned or stored in an outer variable).

```
(function hello() {
    let i = 'greetings';
    function world() {
        let j = 'planetoid';
        console.log(i);
    }
    world();                >> 'greetings'
    console.log(j);         >> Uncaught ReferenceError: j is not defined
})();
```

- `var` declarations outside functions are inside the **global scope**

```
var webb = 'telescope'; console.log(window.webb);   >> 'telescope'
```

Functions

- **Closures:** Because of function scoping, variables can “persist” inside the scope (closure) of a function that was invoked long ago… this can be nice for making “private” variables.

```
const bankAccount = (function() {
    let balance = 100;
    return { checkBalance: function() { return balance; } };
}(); // wtf just happened…

// The ‘balance’ variable is stuck inside the scope of this anonymous function.
// We can’t change it or even look directly at it from out here. :( But…
bankAccount.checkBalance();
>> 100  // nice...
```

Functions

- **Callback Functions:** Everything we do in JavaScript is “blocking”... that is, we don’t go on to line 2 until line 1 is finished. But some “asynchronous” processes (network operations, reading a file, etc.) take a long time…
    - **Solution: Start** the process, give it a function to **call when it is done**, and **move on** with our lives! This function is called a “callback.”

```
function cb() {
    console.log('What if we moved on to the next line?');
}
doAsyncThingThatTakesALongTime(cb);
console.log('Haha… just kidding… unless? 😳');
```

```
>> Haha… just kidding… unless? 😳
[an hour later…]
>> What if we moved on to the next line?
```

Functions

Invocation Patterns: How a function works depends on how it is invoked.

- *Method invocation:* Function is invoked as a method (a property of an object). In this case, the keyword `this` refers to the object that owns the method!

```
cat.numMeows = 10;
cat.meow = function() {
     for (i = 0; i < this.numMeows; i++) {
          console.log('meow');
     }
}
cat.meow();        // this=cat inside meow; will print 'meow' ten times.
```

- *Function invocation:* Function is invoked by itself (e.g., `myFunc()`). `this=undefined` inside `myFunc` (in strict mode).
- *'Apply' or 'Call':* Function is invoked using one of these methods, which allows you to set `this` to be whatever you want. (See documentation for details.)

Functions

- *Constructor invocation:* Function is invoked with `new` keyword, in which case, it acts as a template (or "constructor") to create an object with the fields specified in the body of the function! (Essentially a "class.")
    - Functions written for this purpose are written differently than functions meant to be invoked normally; and it's customary to name them with a Capital letter.

```
function Cat(name) {
    this.name = name;
    this.meow = function () {                  // Bad practice, methods
        console.log("meow " + this.name);      // shared by all members of a
    };                                         // class should be in the
}                                              // prototype.
const myCat = new Cat("Archie");
myCat.meow();                             >> meow Archie
```

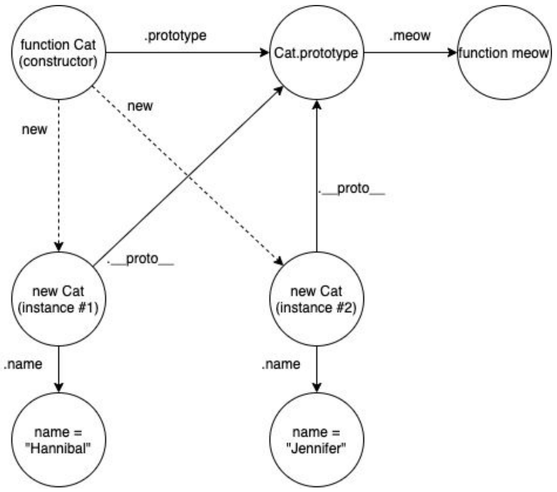
Functions

- *If I want all Cats to share a single "meow" method, add it to the prototype... (this is the proper way to do it; each Cat doesn't need its very own copy of "meow.")*

```
function Cat(name) {
     this.name = name;
}

Cat.prototype.meow = function () {
     console.log('meow ' + this.name);
};

const hann = new Cat('Hannibal');
const jenn = new Cat('Jennifer');
hann.meow();
```



```
>> meow Hannibal
```

Newer Features

- Arrow Functions: (parameters) ⇒ result
  - `var add = (a, b) => a + b;`
  - Keeps the value of ‘`this`’ whatever it was outside the function (can be convenient).
- ES6 (aka ES2015) Classes
  - Instead of using a function with constructor invocation, this allows you to write an honest-to-goodness class! (See the documentation for more details.)

```
class Cat {
    constructor(name) {
        this.name = name;
    }
    meow() { console.log('meow ' + this.name); }
}
```

- You can use these features if you’d like, but it’s not required

Getting Started with Project 2

Getting Started

(i.e., what do I need to write JavaScript?)

- ➔ Install Node.js.
- ➔ Download and unzip the starter code.
- ➔ `npm install`
- ➔ Fire up your favorite text editor. (I recommend Visual Studio Code.)
- ➔ Use `npm run lint` to check your syntax and coding practices .
  - 🙅 If you don’t, you’ll lose style points, 100% guaranteed :)
- ➔ Use `npm test` or open the included HTML file to sanity check your functionality!

`~Warning: Included sanity checks are NOT exhaustive!~`

What’s in the box? (Your newly unzipped directory)

IMPORTANT FOR THE ASSIGNMENT

- `cs142-test-project2.html` — Open this to test your code!
- `cs142-test-project2.js` — Code that runs to test your code!

IN CASE YOU’RE CURIOUS

- `node_modules` 📂
  - Created *after* running `npm install`
  - Has useful packages, like the syntax checker!
- `package.json` — Specifies what packages to install with `npm install`.
- `run-tests-using-node.js` — Used to test your code with Node.js locally.
- `.eslintrc.json` (hidden) — Config for ESLint.

Problem 1:
MultiFilter

Objective

- Write a **function** (`cs142MakeMultiFilter`) that:
  a. **Accepts an array** (e.g., [1, 2, 3]) as input.
  b. **Creates a copy** of that array in its scope, so it persists (think about closures here)...
  c. **Returns a function** (`arrayFilterer`) that allows the user to apply a filter (or multiple filters) to this array.
- Usage:

```
>   const myFilter = cs142MakeMultiFilter([1, 2, 3, 4]);
>   function odd(x)  {return x % 2 === 1}
>   function even(x) {return x % 2 === 0}
>   myFilter();                 // [1, 2, 3, 4]
>   myFilter(odd)();            // [1, 3]
>   myFilter();                 // [1, 3]
>   myFilter(even)();           // []
```

Whoa, wait a sec...

- What's up with the “chaining”? (`myFilter(odd)(even)();`)
  - Our parent function returns a child function, `arrayFilterer`, which is stored in `myFilter`.
  - Because `arrayFilterer` is supposed to return itself, when we call `myFilter()`, the resulting value is *also* `arrayFilterer`, which we can immediately invoke with another `();`
  - This is why it's perfectly okay to do `myFilter(f)(g)(h)();`
  - And, when it’s called with no function, `arrayFilterer` just returns the current array (rather than filtering it). This is what the last empty pair of parentheses does.
  - After filtering out all the odd and even numbers, `currentArray` is empty!

Tips

- The built-in `filter` method of the Array class will come in handy!
- Check if something is a function with `typeof` / `instanceof`.
- Set the value of `this` with `.call()` or `.bind()`.
- Pay attention to the different "cases":
  - **Function?** If a filtering function (like `odd` or `even`) is provided, then filter the array by it, and return `arrayFilterer`. If it's not provided, just immediately return `currentArray`.
  - **Callback?** If a function & a callback is provided, call the callback after filtering, and before returning `arrayFilterer`. If it's not provided, then ignore it.

Problem 2:
Template Processor

Objective

- Write a **class** (using the old-fashioned “function” paradigm) that:
  (1) Accepts a template string (e.g., `'{{greeting}}, my name is {{name}}'`) as parameter.
  (2) Has a method `fillIn` which, when given a “dictionary” (object of key–value pairs), returns a “filled-in” version of the template string where each `{{key}}` is replaced with the corresponding value from the dictionary, and any `{{key}}` not in the dictionary is deleted.
- Usage:

```
>   const tp = new Cs142TemplateProcessor('{{greeting}}, my name is {{name}}');
>   let result = tp.fillIn({greeting: 'hello', name: 'tim'});
>   console.log(result);
      ■   'hello, my name is tim'
>   result = tp.fillIn({greeting: 'bienvenidos'});
>   console.log(result);
      ■   'bienvenidos, my name is '
```

Tips

- Spend some time getting familiar with **regular expressions**. They'll definitely be useful for this problem!
- There are lots of ways to comb through a string.
    - `.exec`, `.replace`, `.replaceAll`, `.match` come to mind.
- Remember, if all Template Processors are going to share a function, it's proper to add that function to the **prototype**, rather than each instance of the class having its own copy.

Problem 3: Global Variables

😷

Objective

Remove variables from the global namespace, without ruining the functionality of the code in the file!

Toy Example:

```
var x = 10;           // This toy example has x and y in the
var y = 5;            // global namespace. How can we remove them?
console.log(x + y);
```

Approach

Could we shove them all into an object?

(Sometimes this is done to keep code clean, since the global namespace in the web browser becomes *incredibly* polluted.) **Will this work?**

```
var VARS = { x: 10, y: 5 };
console.log(VARS.x + VARS.y);
```

Approach

Could we shove them all into an object?

(Sometimes this is done to keep code clean, since the global namespace in the web browser becomes *incredibly* polluted.) **Will this work?**

```
var VARS = { x: 10, y: 5 };       // VARS is now in the global
console.log(VARS.x + VARS.y);     // namespace. :(
```

Approach

What about using a function? Functions have their own private scope!

Will this address our problem?

```
function secret() {
    var x = 10;
    var y = 5;
    console.log(x + y);
}

secret();
```

Approach

What about using a function? Functions have their own private scope!

Will this address our problem?

```
function secret() {
    var x = 10;
    var y = 5;
    console.log(x + y);
}

secret();      // secret is now in the global namespace :(
```

Tips

- The spec says:

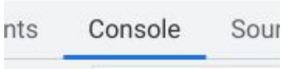
Change cs142-test-project2.js […] using an **<u>anonymous function</u>** to hide symbols in the global namespace yet keep the same checking functionality.

(This should be enough to get you thinking!)

(FYI, another solution is to use let/const instead of var , but **do not** do that for this problem.)

Debugging!

Ways to debug

- Insert `console.log()` statements to test expected values.
  - You can get surprisingly far with plain old print debugging!
  - These things will print to the terminal if your code is running in Node.js, or the browser console (the thing you see when you right-click, hit Inspect, and then click on  if your code is running in the browser
- Using browser DevTools like a pro
  - You can type lines of code into the browser console!
  - Insert `debugger;` statements into your code (your code will pause once reaching it)
- More advanced: Use Chrome DevTools with Node.js (examples)
  - Run `node --inspect-brk run-tests-using-node.js`
  - Visit `chrome://inspect` in Chrome and find the inspector session to debug.
- Finally, if you're failing some tests, try looking at the code for the tests, and see what kinds of inputs are being given to the functions you're writing.

Thank you!

My office hours: Wed 3:15–5:15pm, Thu 11:30am–1:30pm.