

Why do you learn a new language?

- Platform requires it, e.g., JavaScript in the web browser
- You want to use a particular framework or library, e.g., Ruby on Rails

In an academic setting, we often learn new languages for pedagogical purposes, to get exposure to new ways of thinking. In a professional setting, we usually need a more pragmatic reason.

Often there simply is no choice – web browsers run/require Javascript. Often though we choose a particular framework or library first, as we think it will be a good fit for a particular problem. The choice of framework determines the choice of language.

When we talk about learning a new language, we are often actually talking about learning an entire “stack”, an ecosystem consisting of language(s), framework(s), and tools (e.g., database systems, development tools, etc.)

This is a moment where we made one of those “no right answer” decisions. To minimize the number of languages we learn in this class, we will use Javascript for both the front-end and back-end of our applications, not just the front-end components that run in the browser. This is not the only choice, many (e.g., the SaaS book authors) argue one should use different languages/frameworks on the back-end.

Tips for learning a new language

- Most imperative OO languages are similar, but analogous doesn't mean identical...
- A language likely has features that made it a good foundation for that framework/tool
 - Such features/idioms are likely heavily used and so are important to master!
 - May also be the aspects that are least familiar
- Master the mechanics of debugging, installing libraries, etc.

Typically, there was some reason that a language was chosen for a specific task, or to be the base for a particular framework. That reason, or more specifically those features are likely heavily used and so it is something we want to master. In the case of JS, we will see that the language has evolved for its specific use case – enabling interaction in the browser and so the features that we want to master relate to facilitating that interaction.

Learning JavaScript (in CS312)

JavaScript is an object-oriented, prototype-based, dynamic, “brackets” language

- A pragmatic language that “evolved” (instead of being “designed”)
- Gotchas abound
- Recent versions (ES6) have smoothed some rough edges (e.g., introduced “classes”)

The tools (and the notes) will help teach us the gotchas, our goal in-class is the main ideas

A key thing to remember: Javascript is to Java as Hamburger is to Ham ... Javascript has nothing to do with Java, that was purely a marketing move (Java was very popular/prominent at the time).

Gotchas? Smoothed? Variable definition example

~~no declaration~~

- Implicitly create a new global variable

~~var myVariable;~~

- Create new variable with function (or global) scope
- Variables are *hoisted* to the top of their context

let myVariable;

- Create new variable with block-level scope

const myVariable;

- Create a new constant variable with block-level scope

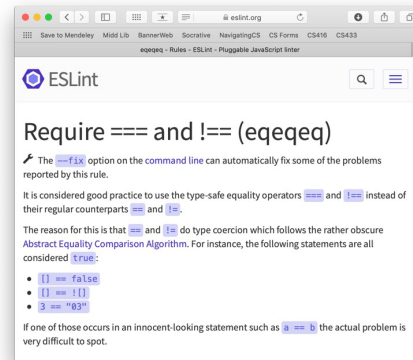
This is one of the places in JS that is improving. You should always use let or const, and preferentially use const whenever possible. Note that making an object (like an array) a const variable doesn't mean you can't change its contents, instead const refers the reference. You can't change the value the variable refers to. Using the most restrictive form of variable definition is the programming equivalent of defensive driving, it reduces the "surface area" for things to go wrong.

Note that you will find many instances of var in outdated online examples. Don't copy those! And more generally, pay attention to the dates on Stack Overflow, etc.

Tools and gotchas

```
$ npm run lint  
index.js
```

```
74:16 error Expected '===' and instead saw '==' eeqeq  
✖ 1 problem (1 error, 0 warnings)
```



Google "eslint eqeqeq"

We will use a variety of development tools to help us try to avoid these kinds of gotchas (note "help" and "try", no tool is perfect). For example, ESLint is a static analysis tool that can flag certain errors and/or bad style (like removing fuzzy "lint" from a sweater). We will make extensive use of ESLint throughout the semester.

As example, == for equality in JS will do some surprising forms of type coercion. === (triple equals) behaves in a more consistent and expected way; it should be used instead. ESLint would flag the use == in our code.

From the ESLint description:

It is considered good practice to use the type-safe equality operators === and !== instead of their regular counterparts == and !=. The reason for this is that == and != do type coercion which follows the rather obscure [Abstract Equality Comparison Algorithm](#).

You can setup your development environment to help you (i.e., link directly to rule explanations).

Function declarations

| Form | Example |
|---|---|
| Function declaration | <pre>function double(x) { return x * 2; }</pre> |
| Function expression | <pre>const double = function(x) { return x * 2; }</pre> |
| Named function expression | <pre>const double = function f(x) { return x * 2; }</pre> |
| Function expression (fat arrow) | <pre>const double = (x) => { return x * 2; }</pre> |
| Function expression (fat arrow, implicit return) | <pre>const double = (x) => x * 2;</pre> |

As another example, as part of that evolution, JS has many ways to declare a function... Function expressions, and particularly the “fat arrow” flavor, tend to be favored since they are more versatile. They also highlight the fact that functions are just objects (and we can do object related things with them, e.g., assign to variable, attach properties).

Higher-order functions

```
const m = [4,6,2,7];
for (let i=0; i<m.length; i++) {
  console.log(m[i]);
}

m.forEach(function(i) {
  console.log(i);
});
// or...
m.forEach((i) => {
  console.log(i)
});
```

Abstract over “actions” not just values
by passing functions as arguments

Common operations of this kind are map, filter, reduce and sort

What do we mean by abstracting over actions? Instead of writing a function that filters data with specific (and fixed) predicate and applying that function to arbitrary data, we are writing a generic filter function that can be applied to arbitrary data *and* implement arbitrary predicates (by supplying a different predicate function value).

We prefer these high-order approaches, listed at the bottom, over loops. Why? Easier to reason about and the compiler/runtime to optimize. All of these methods are applied to an array, i.e., the input is an array of elements. The choice is determined by the desired return value:

- No return value, i.e., we only care about side-effects: `forEach`
- An array of the same length with some transformation applied: `map`
- An array that contains a subset of the original items: `filter`
- A single (“scalar”) value: `reduce`

How would you implement

`map(a, f)`

such that

```
> const m = [4,6,7,9];
```

```
> map(m, item => item + 1);
```

```
[ 5, 7, 8, 10 ]
```


How would you implement

`map(a, f)`

such that

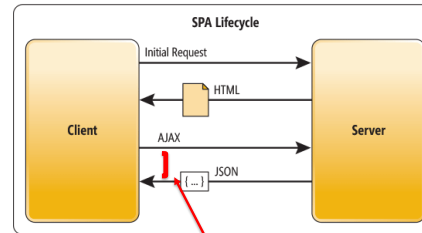
```
> const m = [4,6,7,9];
```

```
> map(m, item => item + 1);
```

```
[ 5, 7, 8, 10 ]
```

```
const map = (a, f) => {  
  let result = [];  
  a.forEach((item) => {  
    result.push(f(item));  
  });  
  return result;  
};
```

What is the browser doing with its time?



What is happening during this time?

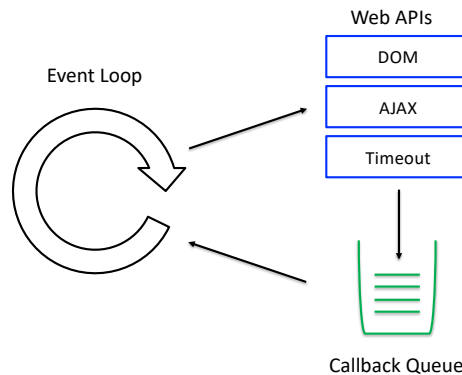
[Wasson, Microsoft](#)

Recall that Javascript was originally developed, and primarily used in the browser. Thus, the best way to understand its design is to think about the kinds of problems that arise in a web browser and thus the problems that JS was designed to solve.

Short answer: It is doing other stuff while waiting for the server to respond with the data, and specifically handling other "interactions", e.g., the user clicking on something else.

What that means is we need a way in the language to specify operations – code – we want to execute in the future when some event occurs, e.g., the server sends the data back. The design of the language and our use of it is built around this need.

The browser is *asynchronous*



The heart of JS execution is the event loop. The Event loop is constantly “spinning” executing callbacks in response to events. So, if the user clicks a link, doing so adds a click handler to the queue. When that handler is executed, it might launch a network request. While that the browser is waiting for the response it is processing other events (and the response will eventually trigger adding additional callbacks to the queue). That is, the browser is executing actions asynchronously (i.e., the click handler executes some time in the future). Note that this is not the same as executing actions in parallel. The event loop effectively single threaded, i.e., it executes one handler at a time. If you have ever observed the browser hang, that is JS code monopolizing that single thread preventing the event loop from advancing.

What exactly is a callback? A callback is a function that is executed when another operation has completed, i.e., when a network request has completed. But it is not just the “next” code in the program, instead it is a function we have supplied (typically as an argument) to be executed at some point in the future. What do we need to make that work?

- Be able to supply functions as argument (functions as 1st class objects) – we saw this already
- Be able to hold on to state in a function (closures)

Making callbacks work in JS

```
const wrapValue = (n) => { // function(n) {  
  const local = n;  
  return () => local; // function () { return local; }  
}  
  
let wrap1 = wrapValue(1); // () => 1  
let wrap2 = wrapValue(2); // () => 2  
console.log(wrap2()); // What will print here?  
console.log(wrap1()); // What will print here?
```

Functions as 1st class objects

Function "closes" over local

[click] Recall that "functions as 1st class objects" means functions are a type in the language, can be created during execution, stored in variables/data structures, passed as arguments or returned. For a more formal definition you would need to take a functional programming class.

[click] Here we see an example of creating anonymous functions using the ES6 arrow function (including concise body), and the pre ES6-syntax. This function "closes" over the variable local. By "close", we mean we have access to the variables that were in scope *when* the functions was defined, even if those variables are no longer in scope when it executes.

More formally: "Closure is when a function is able to remember and access its lexical scope even when that function is executing outside its lexical scope."

What will this print?

2
1

[click] Why? We are creating a function that closes over 1, e.g., the function shown here, and another that closes over 2. We then print the values returned by those closures but starting with 2.

<https://github.com/getify/You-Dont-Know-JS/blob/master/scope%20%26%20closures/ch5.md>

What does the following code print?

```
let current=Date.now(); // Time in ms since epoch
// setTimeout(callback, delay[,param1[,param2...]])
// setTimeout returns immediately, and then invokes the
// callback after delay in milliseconds.
setTimeout(() => {
  console.log("Time elapsed (ms): " + (Date.now() - current));
}, 100);
console.log("First?");
```

| A | B | C | D |
|--------|----------------------------------|----------------------------------|-------------------------|
| First? | First? Time elapsed (ms): 100 | Time elapsed (ms): 100 First? | None of the above |

Context: setTimeout returns immediately, and then invokes the callback after delay in milliseconds.

Answer: B

Although the second print command is "later" in the code, it executes first because the callback does not execute until after 100ms has elapsed. In the meantime, execution moves onto the next line, printing of "First?" The time elapsed won't exactly be 100. It will be larger but reasonably close. Why? The callback goes into the queue after the timeout but may not be executed exactly at that moment. The callback function closes over the variable current which was set to the time just before calling setTimeout.

What does the following code print?

```
let current=Date.now(); // Time in ms since epoch
// setTimeout(callback, delay[,param1[,param2...]]) delay in ms
setTimeout(() => {
  console.log("Time elapsed (ms): " + (Date.now() - current));
}, 100);
current = new Date("11 Feb 2019");
console.log("First?");
```

Function "closes" over current
variable which is still in scope after
setTimeout

| A | B | C | D |
|--------|----------------------------------|----------------------------------|-------------------------|
| First? | First? Time elapsed (ms): 100 | Time elapsed (ms): 100 First? | None of the above |

Answer: D

What happened? The implication of our discussion was that the callback function "closed" over current. And that is the case, but it closes over the *variable* not the *value* of that variable. Here the same variable is in scope when we create the closure and when we modify current after setTimeout. Most of the situations in which we use closures we are creating new variables (e.g., as function arguments) and thus it appears we are closing over both the variable and the current value. But in reality, we only close over the variable.

What does the following code print?

```
let current=Date.now(); // Time in ms since epoch
// setTimeout(callback, delay[,param1[,param2...]]) delay in ms
setTimeout(((past) => (() => {
  console.log("Time elapsed (ms): " + (Date.now() - past))
}))(current), 100);
current = new Date("11 Feb 2019");
console.log("First?");
```

| A | B | C | D |
|--|----------------------------------|----------------------------------|-------------------------|
| First? Time elapsed (ms): 31592310870 | First? Time elapsed (ms): 100 | Time elapsed (ms): 100 First? | None of the above |

Answer: B

For example, if we rewrote that code as follows, we would get B (as we would expect). Here we are closing over `current` as the argument `past`, when we create the callback. That is behind the scenes we are doing something like `past=current` and then closing over `past`.

A simpler way to implement this in practice would be to use the additional arguments to `setTimeout`. It closes over those arguments and pass them to the supplied callback.

```
setTimeout((past) => {
  console.log("Time elapsed (ms): " + (Date.now() -
past))
}, 100, current);
```


What did we actually do there...

```
let current=Date.now(); // Time in ms since epoch

// Create callback provided to setTimeout
const createCallback = (past) => {
  return () => {
    console.log("Time elapsed (ms): " + (Date.now() - past))
  };
};
const callback = createCallback(current);

setTimeout(callback, 100);
current = new Date("11 Feb 2019");
console.log("First?");
```

This is the equivalent code to what we saw before, but likely a little easier to reason about. Note that the role of `createCallback` is exactly that. We are creating a callback that closes over the `past` parameter. Thus it is not affected by the reassignment to `current` below.

Your take on closures?

“Closures are often avoided because it’s hard to think about a value that can be mutated over time.”

-Dan Abramov

That is exactly the issue in the 2nd version of the problem!

When we close over constant variables, e.g., const variables or parameters that won’t change, - the typical case - then closures are more straightforward (and a key tool when working with JS).

From Dan Abramov (one of the key React developers, and a name we will encounter repeatedly)

“Closures are often avoided because it’s hard to think about a value that can be mutated over time.”

That is exactly the issue we saw in the 2nd problem. We were closing over a mutable variable. Closures are a very powerful tool and fundamental part of working with the JavaScript event loop. We can’t really avoid them. While we want to understand what is going on the second example, we would rather not create that kind of tricky situation in the first place! Returning to our defensive driving analogy, let’s proactively make choices to minimize the chance something goes awry. When we close over constant values, either “const” values or arguments that behave like const references, we will find closures are easier to reason about.

<https://overreacted.io/how-are-function-components-different-from-classes/>

The rest of JS?

- We are not going to spend a lot of time “just” learning JS

Today we focused on features that support interactive applications in the browser

In the future, we will talk more about Promises and the `async/await` keywords

- We will pick up the rest “along the way”

Check out the links on the website for common, but perhaps unfamiliar, syntax

Pay attention to messages from ESLint (really read the rule documentation and its motivation)