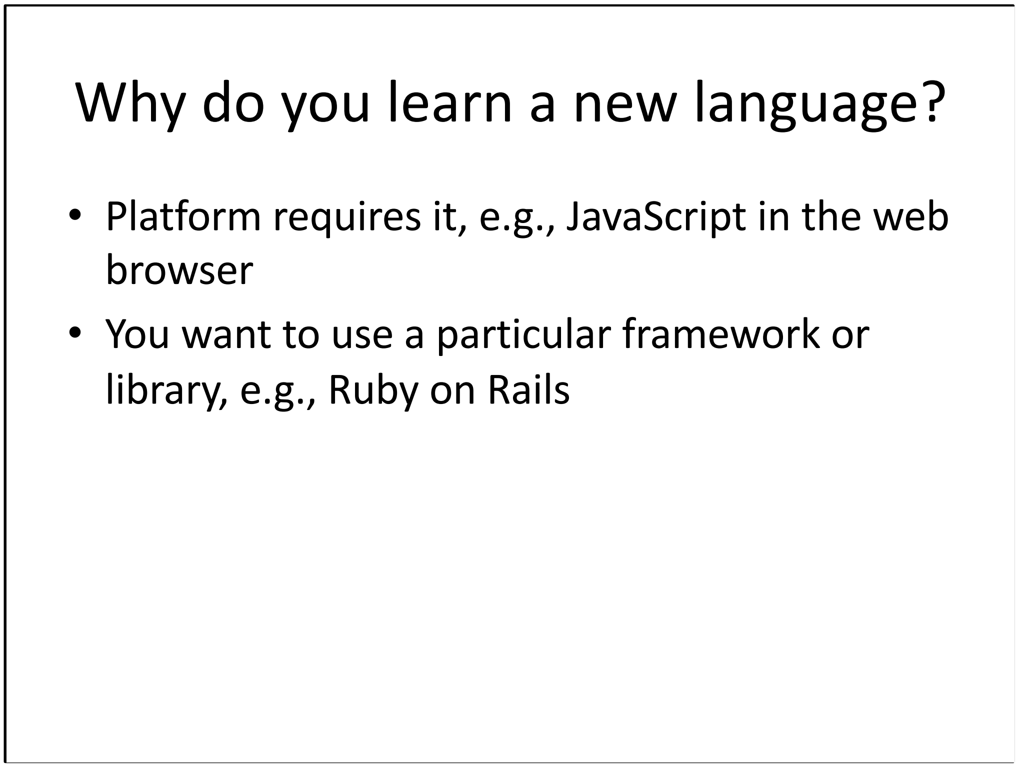


Why do you learn a new language?

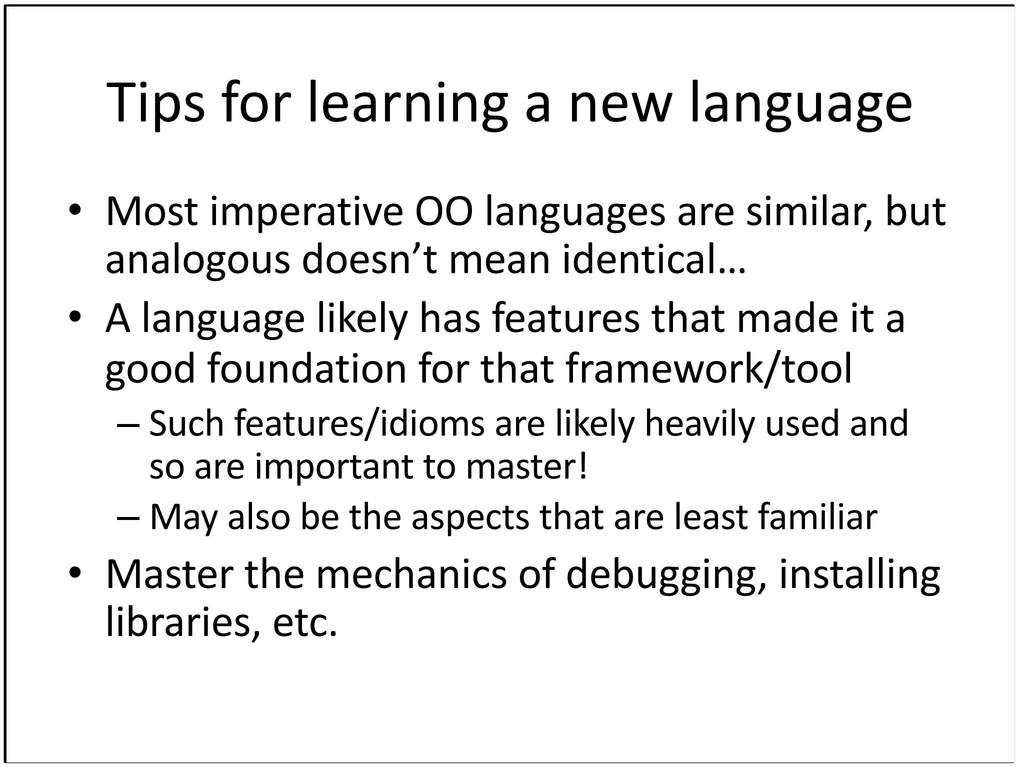
- Platform requires it, e.g., JavaScript in the web browser
- You want to use a particular framework or library, e.g., Ruby on Rails

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 (it is the only language natively supported across all browsers). 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 really 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 the not the only choice, many (e.g., the Saas book authors) argue one should use different languages/frameworks on the back-end.

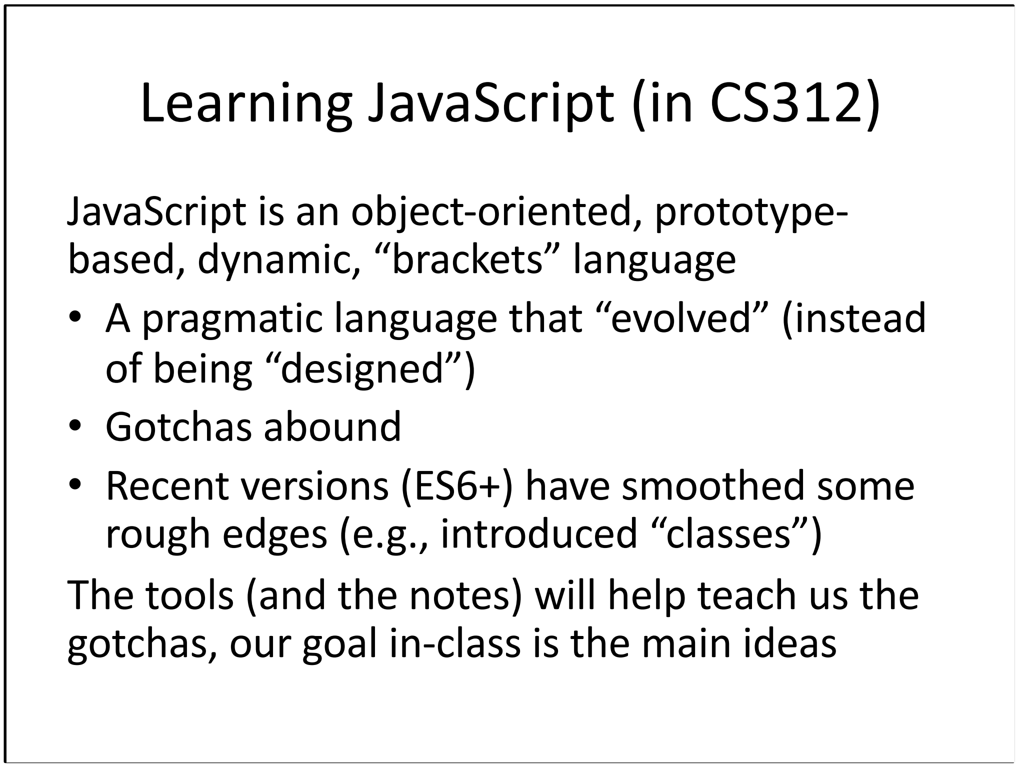


Tips for learning a new language

- 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.

Mastering a language is more than learning its syntax. In practical/professional usage, it also (maybe even more so) learning about an entire ecosystem. How you create projects, install/use libraries, test, etc. We will spend as much time (if not more) on the latter!



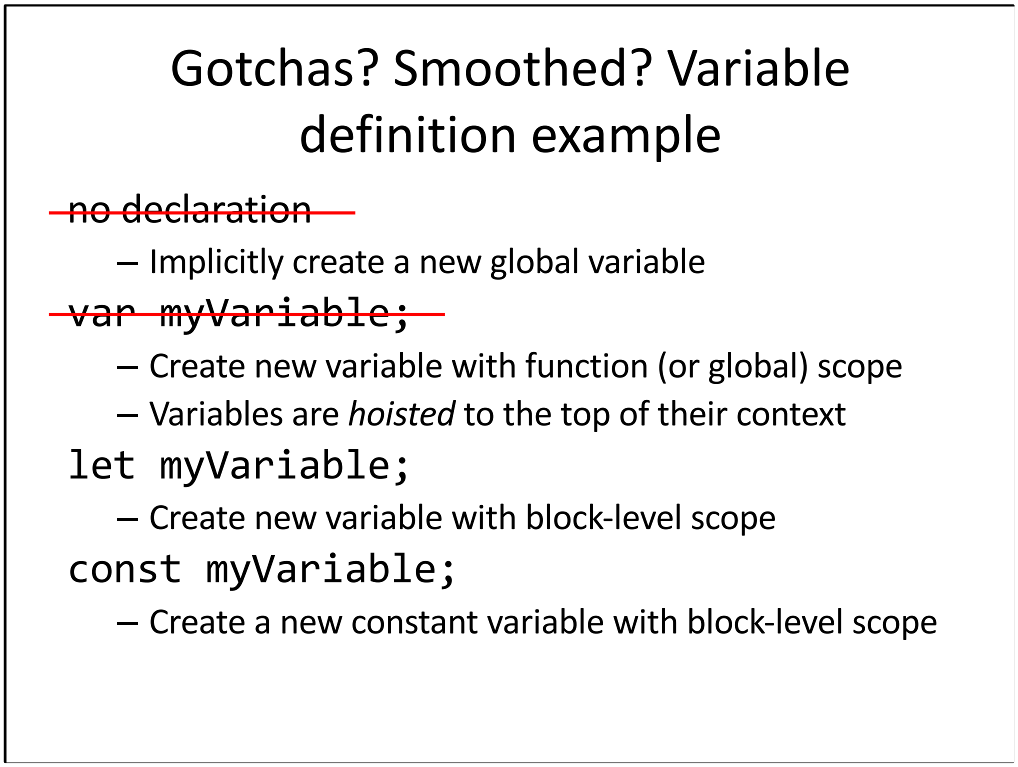
Learning JavaScript (in CS312)

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;`

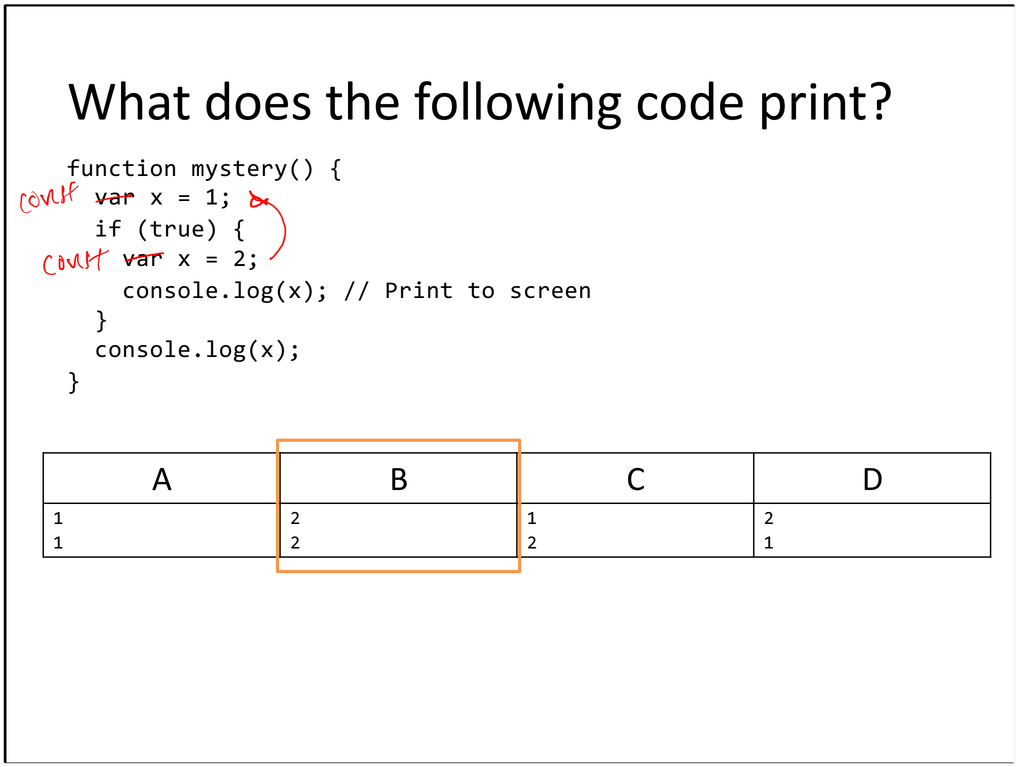
- Create new variable with block-level scope

`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.

This is also a reminder that the examples, etc. we find online may date from earlier, “gotcha” era. Don’t copy that outdated code! With JS we need to pay attention to the dates on Stack Overflow, etc.



What does the following code print?

```
function mystery() {
const ~~var~~ x = 1; ~~x~~
  if (true) {
 const ~~var~~ x = 2;
    console.log(x); // Print to screen
  }
  console.log(x);
}
```

A	B	C	D
1 1	2 2	1 2	2 1

Answer: B

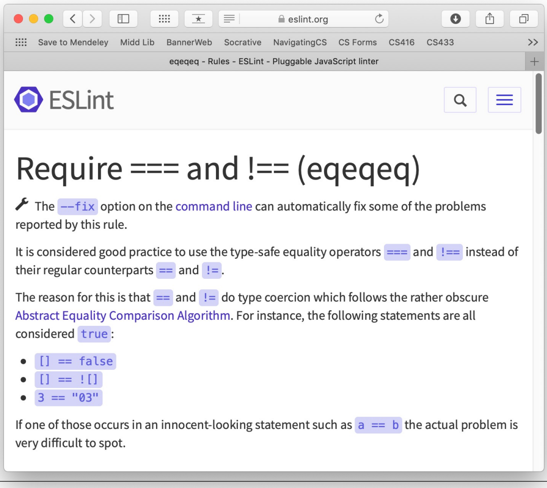
Ugh. Note from the previous discussion, var ”hoists” a variable to top of the function scope, i.e., there is only one x here and it has the value of 2. If we place `var` with `let` we will get the more expected behavior of 2 then 1.

https://sentry.io/answers/difference-between-let-and-var-in-javascript/

Tools and gotchas

```
$ pnpm run lint
index.js
  74:16  error  Expected '===' and instead saw '=='  eqeqeq
✖ 1 problem (1 error, 0 warnings)
```

Search for “eslint eqeqeq”



Require === and !== (eqeqeq)

The `--fix` option on the command line can automatically fix some of the problems reported by this rule.

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. For instance, the following statements are all considered `true`:

- `[] == false`
- `[] == ![]`
- `3 == "03"`

If one of those occurs in an innocent-looking statement such as `a == b` the actual problem is very difficult to spot.

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.

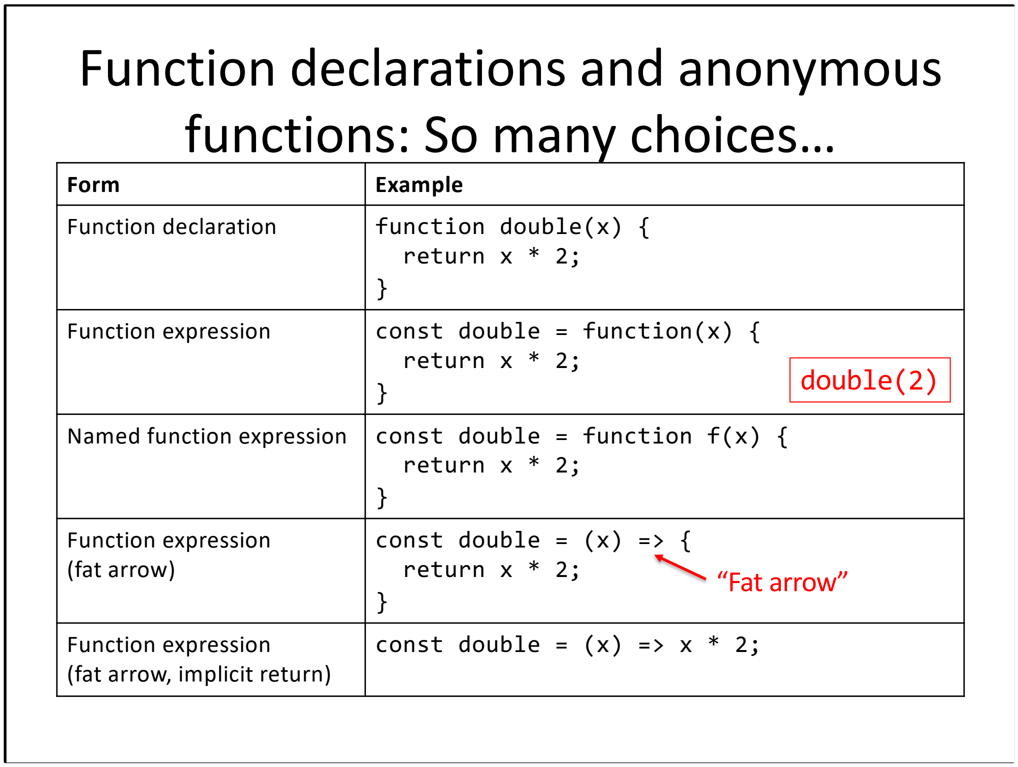
As example, == for equality in JS will do some surprising forms of type coercion. === (triple equals) behaves in a more consistent and expected way; it should be used instead. ESLint would flag the use == in our code.

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).

For example, our lint configurations would flag the previous example with the no-var rule, specifically “Unexpected var, use let or const instead.”



Function declarations and anonymous functions: So many choices…

Form	**Example**

Function declaration	`function double(x) {`
	`  return x * 2;`
	`}`

Function expression	`const double = function(x) {`
	`  return x * 2;`
	`}`
	double(2)

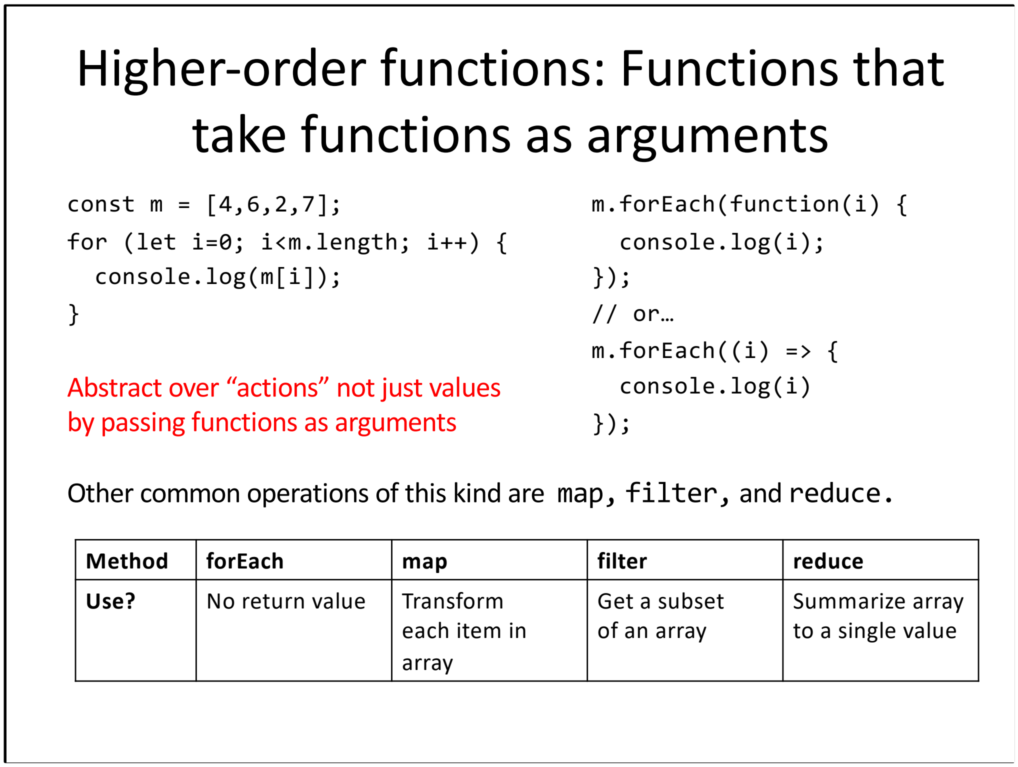
Named function expression	`const double = function f(x) {`
	`  return x * 2;`
	`}`

Function expression
(fat arrow)
	`const double = (x) => {`
	`  return x * 2;`
	`}`
	“Fat arrow”

Function expression (fat arrow, implicit return)	`const double = (x) => x * 2;`

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.

These examples also highlight that in JS functions are just objects and we can do object-related things with them, e.g., assign to variable, attach properties. This is one of our first "key features" that is driven by/enables the interactive use in the browser. In the latter 4 examples note we are assigning a function to a variable. We can use that variable to call the function, e.g., `double(2)`, just as if we had created it with that name (i.e., the first example). Functions created without a name, e.g., specifically examples 2, 4, 5, are called anonymous functions. We commonly create and use anonymous functions as arguments to other functions, i.e., to pass a set of operations to be performed by other code…



Higher-order functions: Functions that take functions as arguments

```
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

Other common operations of this kind are `map`, `filter`, and `reduce`.

Method	forEach	map	filter	reduce
Use?	No return value	Transform each item in array	Get a subset of an array	Summarize array to a single value

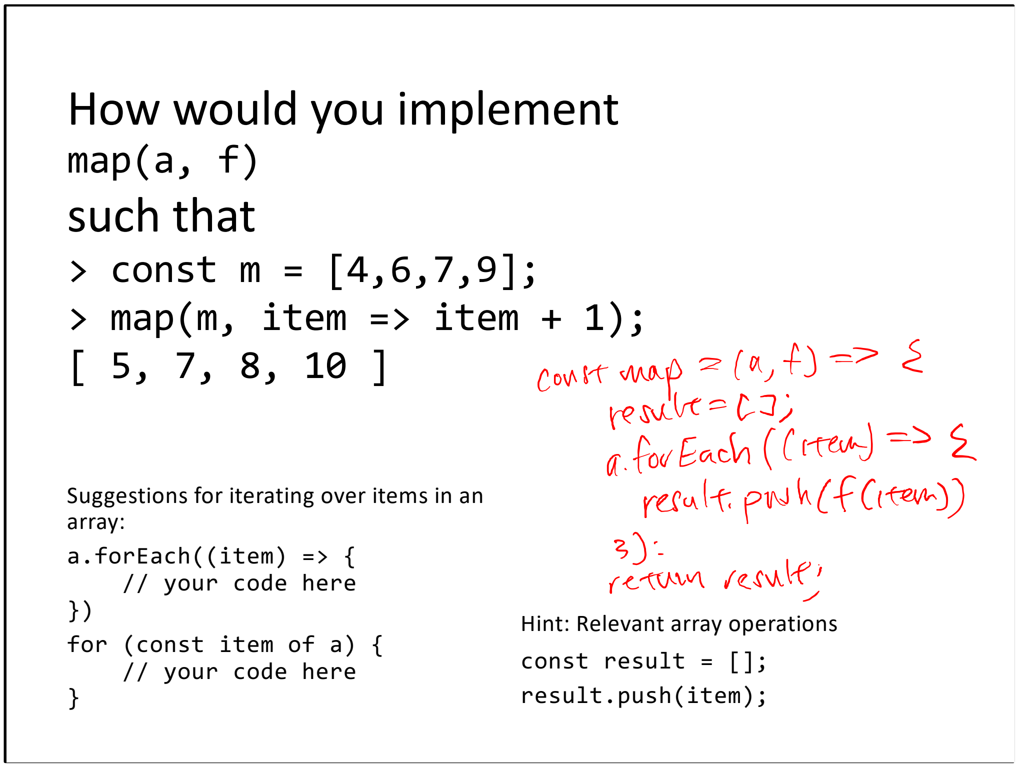
Higher order functions are functions that take other functions as arguments. Frequently in JS, we will use higher-order function to abstract over actions. What do we mean by abstracting over actions? Instead of a writing a loop that prints an array, or a function that filters data with specific (and fixed) predicate and applying that function to arbitrary data, we are writing a generic function for iterating through an array or filtering an array, or …These generic functions can be applied to arbitrary data *and* implement arbitrary actions, e.g., arbitrary predicates, by supplying a different functions as an argument. This loop, for example, iterates over a specific array (m) and performs a specific operation (printing) <click> We could implement it instead with `forEach`. The `forEach` method on arrays implements that part of iterating over an array that is "the same every time". We implement different operation by changing the function we provide to `forEach`…

We generally prefer these high-order approaches over loops. Why? Easier to reason about and the compiler/runtime to optimize. <click>

Common operations are … All of these methods are applied to an array, i.e., the receiver of the method call is an array of elements. The choice of method 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 ]
```

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

Suggestions for iterating over items in an array:

```
a.forEach((item) => {
    // your code here
})
for (const item of a) {
    // your code here
}
```

Hint: Relevant array operations

```
const result = [];
result.push(item);
```

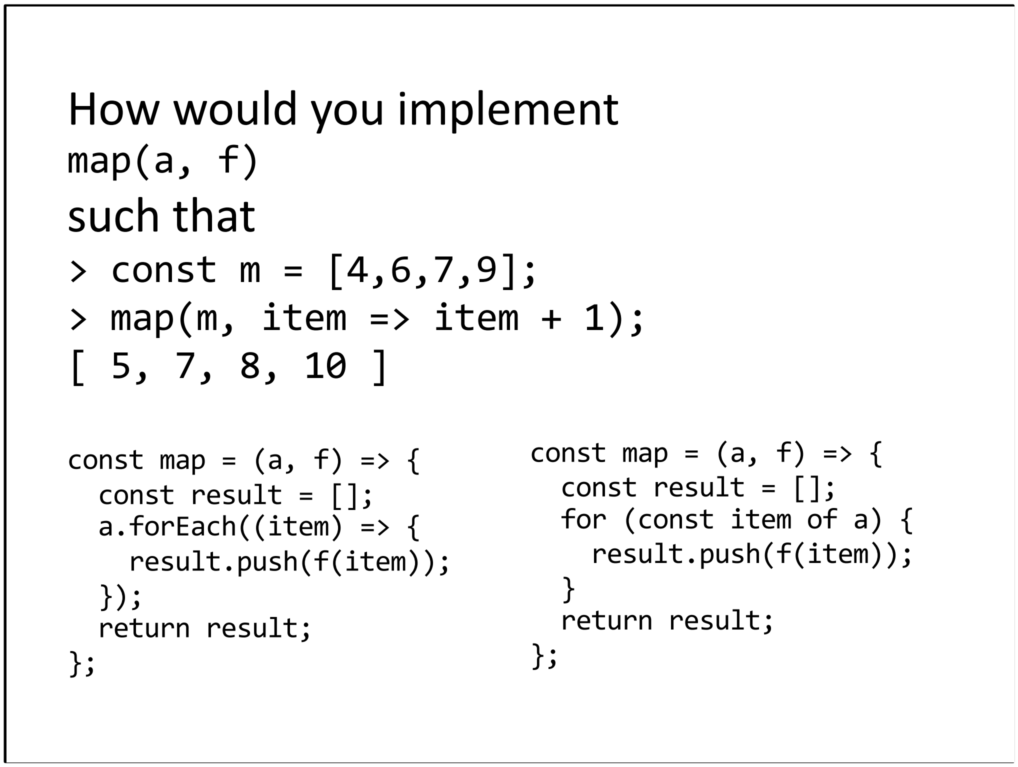
Let’s remind ourselves what `map` is doing… How could we implement our own map function to do the following? At the heart, what do we need? Some way of iterating over the items and building up a new array <click>…

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

Historically we might have does something like,

```
for (var index = 0; index < a.length; index++) {
```

or now, let index. However, I am going to advocate we use forEach. Opinion warning… Why forEach instead of a familiar for loop-based approaches? And specifically, the modern for-of construct. … The forEach is concise, and as we will see shortly can avoid to tricky issues with the loop variable in asynchronous code. There are subtle and important differences between for-of and the similar but not identical for-in. We want to use the former. Using `forEach` by default can be a form of “defensive programming” that tries to minimize the surface area for bugs like mixing up for-of and for-in (like “defensive driving”). We would only use a for loop if needed to exit early (e.g., break, etc.) or use some other imperative feature of loops.



How would you implement

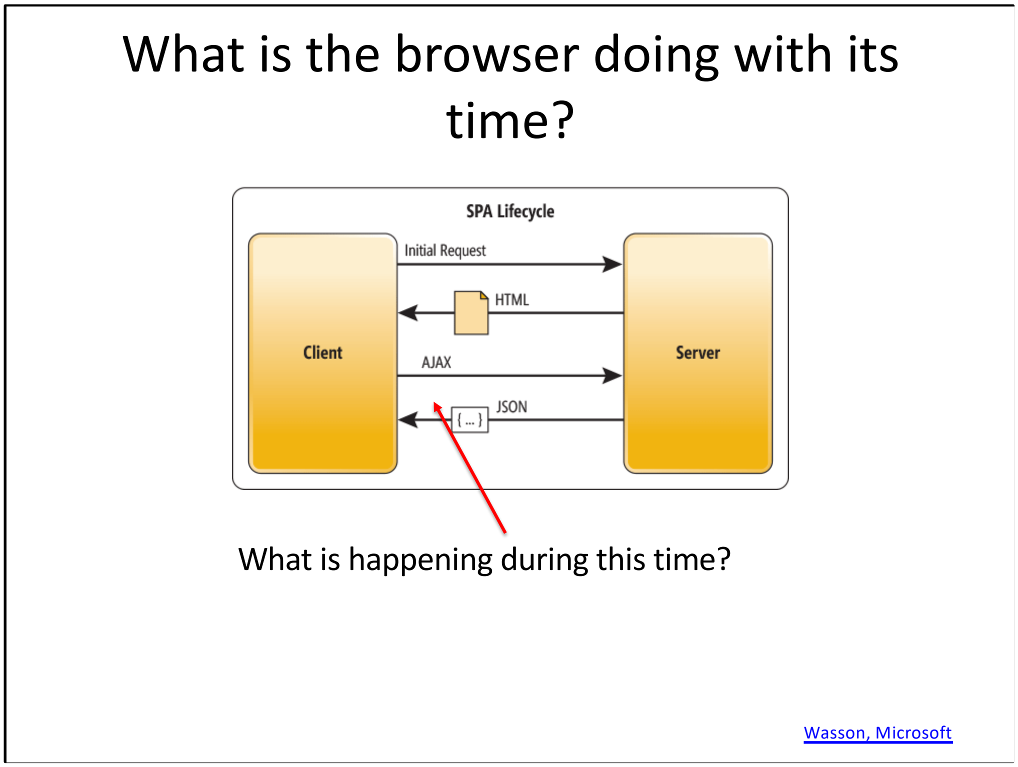
```
map(a, f)
```

such that

```
> const m = [4,6,7,9];
> map(m, item => item + 1);
[ 5, 7, 8, 10 ]
```

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

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



What is the browser doing with its time?

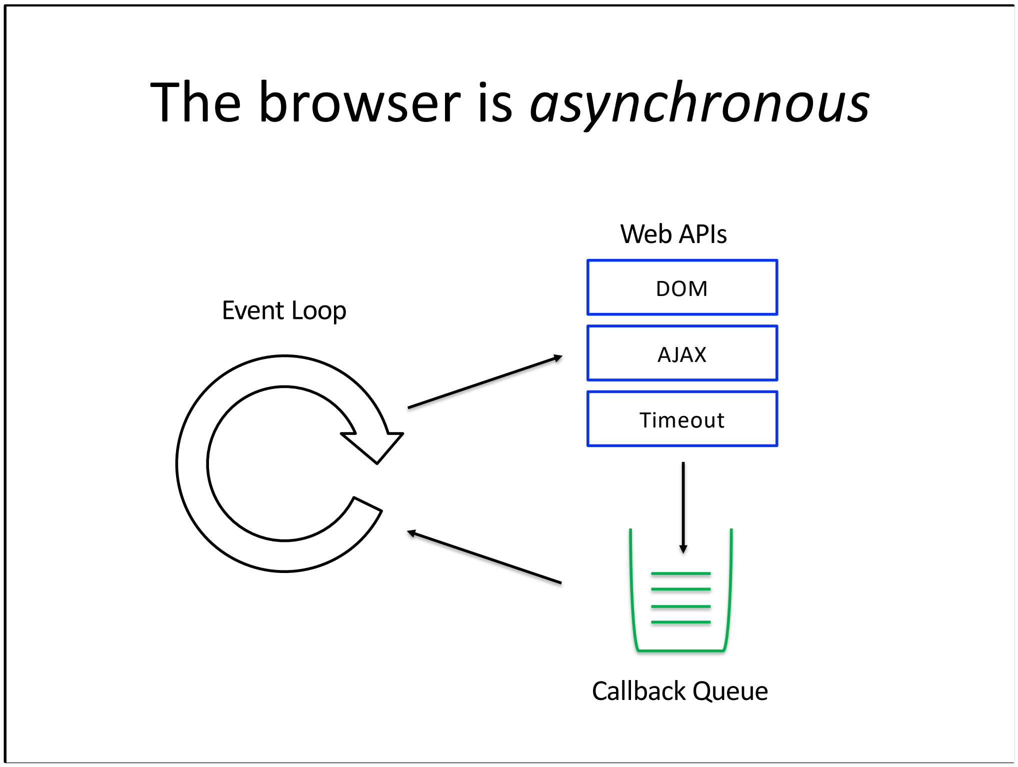
What is happening during this time?

Wasson, Microsoft

How is all this relevant in the browser and how do these features enable the kind of kind of code we want to write in a web application? <click> Let's think about what browser doing while we wait for this network request to complete?

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. As you might imagine, providing a function as an argument is a key enabler to doing so.

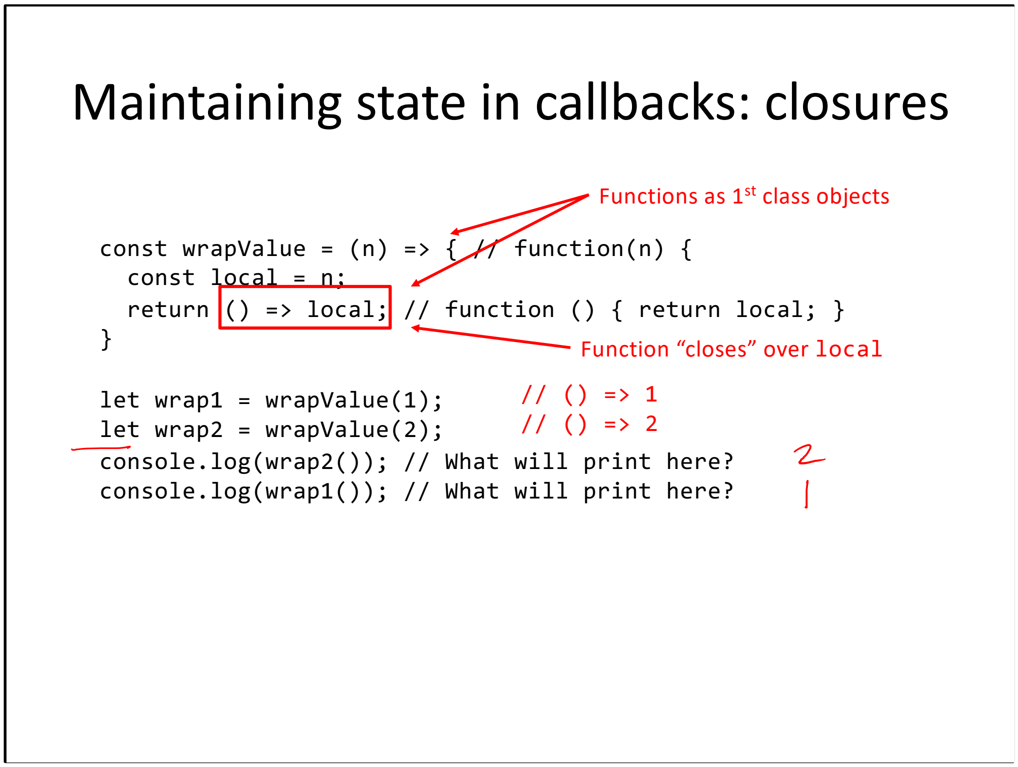


The browser is *asynchronous*

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 unknown 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 with our higher-order function examples
- Be able to hold on to state in a function (i.e., closures)



Maintaining state in callbacks: closures

Functions as 1st class objects

```
const wrapValue = (n) => { // function(n) {
  const local = n;
  return () => local; // function () { return local; }
}
```

Function “closes” over local

```
let wrap1 = wrapValue(1);      // () => 1
let wrap2 = wrapValue(2);      // () => 2
console.log(wrap2()); // What will print here?     2
console.log(wrap1()); // What will print here?     1
```

[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 an arrow function (including concise body). When we execute `wrapValue, we will return this newly created function and `local` will go out of scope (it is only defined within `wrapValue`). But the function we are creating `() => local` “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 we might say: a “closure is when a function is able to remember and access its lexical scope even when that function is executing outside its lexical scope.”

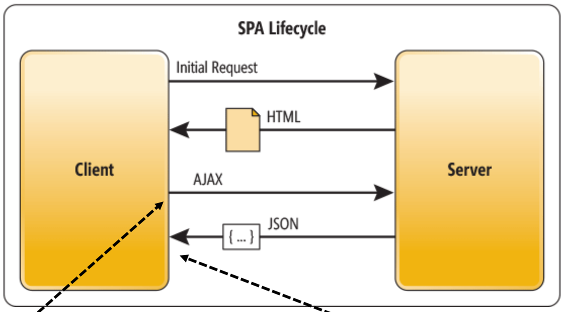
What will this print?
2
1

[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

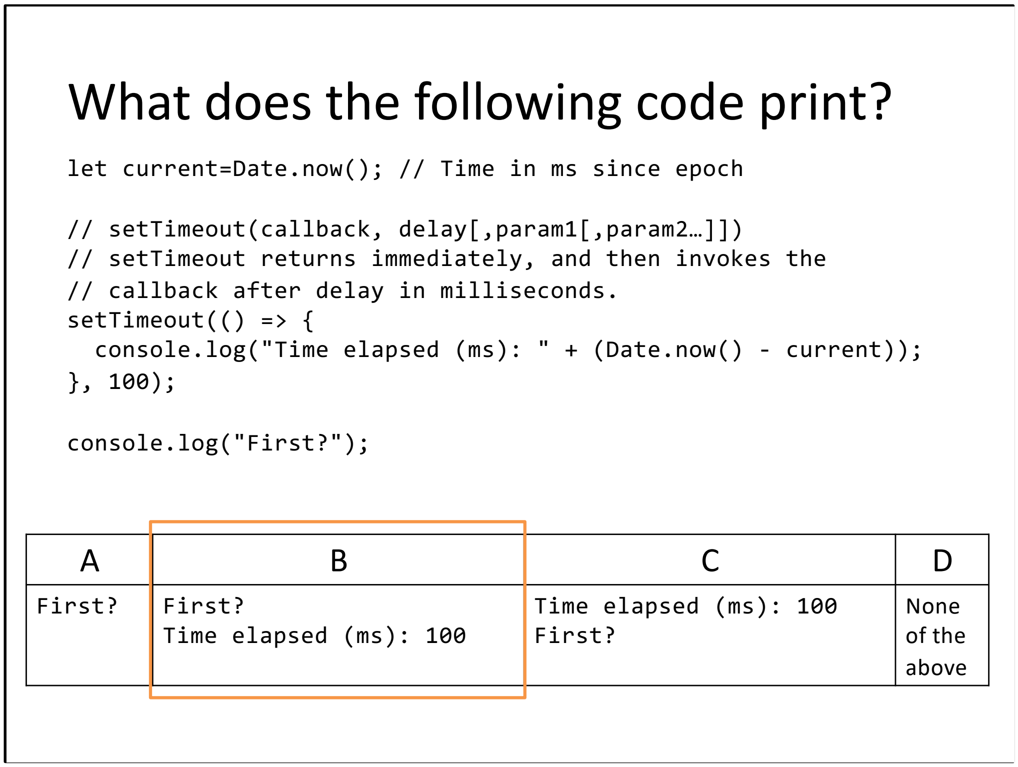
Why is this useful in the browser?



Presumably, we know what we want do with the response when we define/launch the request.

Closure helps us easily maintain that state until the response is available without keeping those values in scope.

Wasson, Microsoft



What does the following code print?

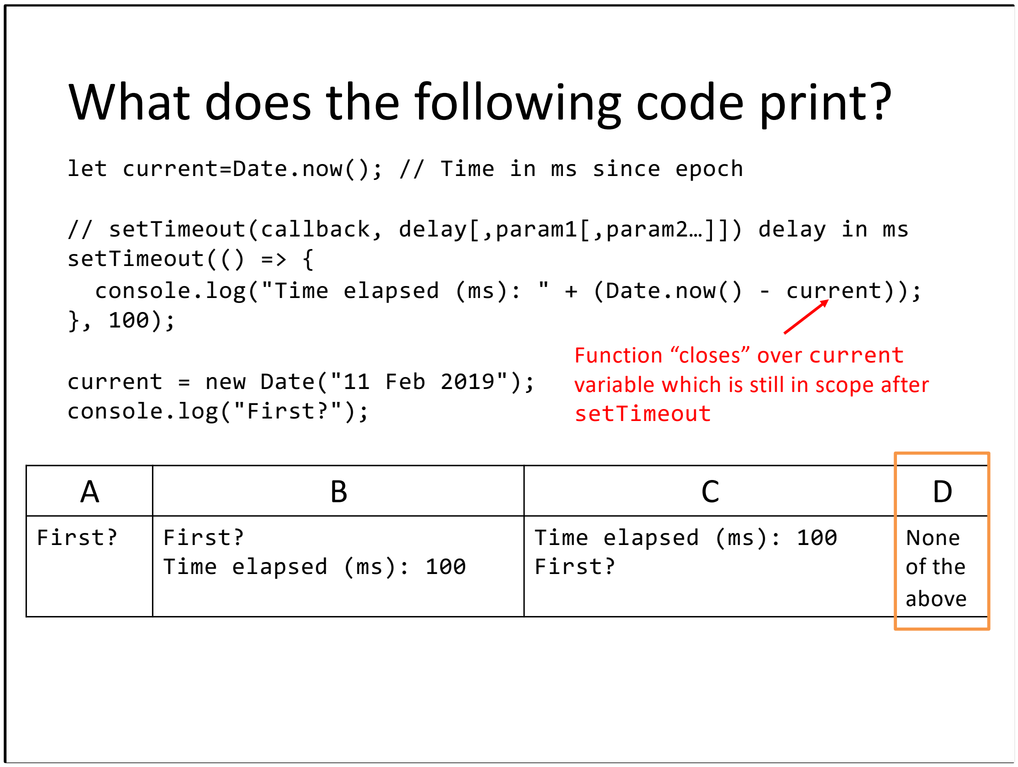
```
let current=Date.now(); // Time in ms since epoch

// 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?");
```

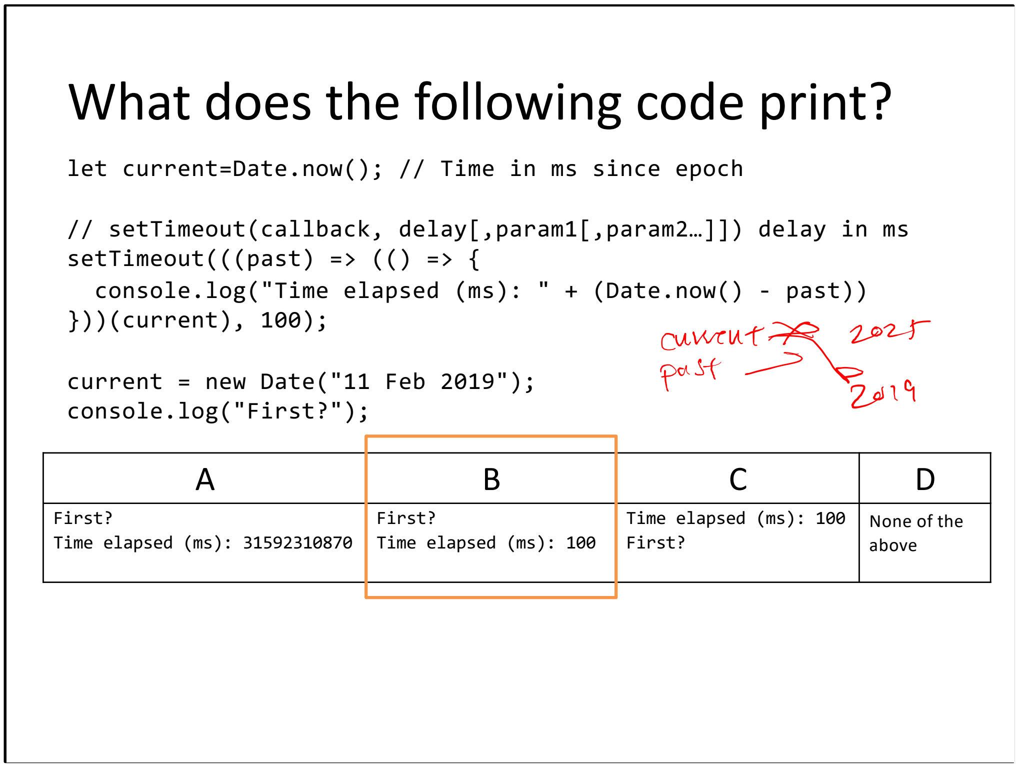
Function “closes” over `current` variable which is still in scope after `setTimeout`

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?");
```

current 2025
past
2019

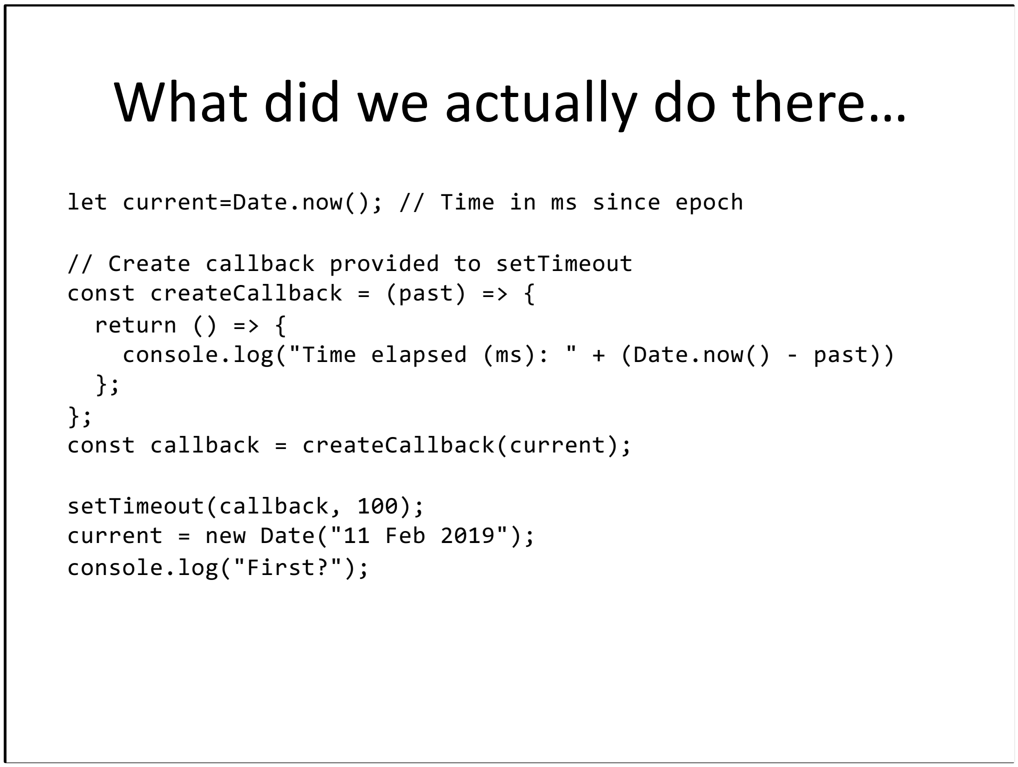
A	B	C	D
First? Time elapsed (ms): 31592310870	First? Time elapsed (ms): 100	Time elapsed (ms): 100 First?	None of the above

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.

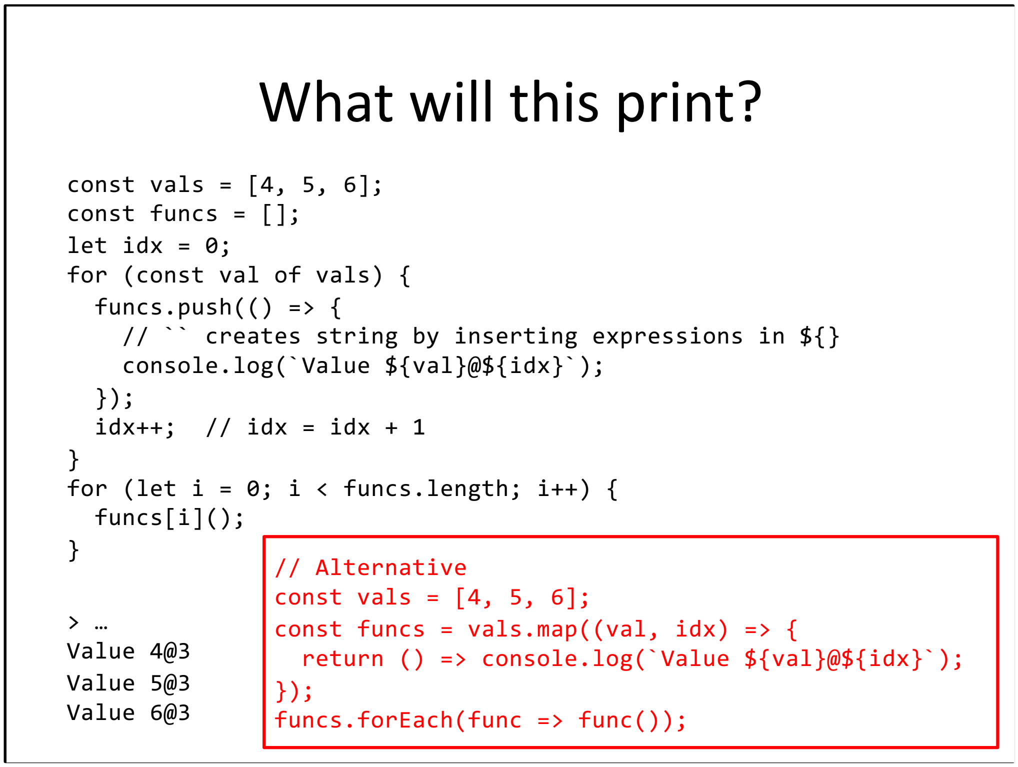
```
setTimeout((past) => {
  console.log("Time elapsed (ms): " + (Date.now() -
past))
}, 100, current);
```



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 no affected by the reassignment to current below.



What will this print?

```
const vals = [4, 5, 6];
const funcs = [];
let idx = 0;
for (const val of vals) {
  funcs.push(() => {
    // `` creates string by inserting expressions in ${}
    console.log(`Value ${val}@${idx}`);
  });
  idx++;  // idx = idx + 1
}
for (let i = 0; i < funcs.length; i++) {
  funcs[i]();
}
```

```
> …
Value 4@3
Value 5@3
Value 6@3
```

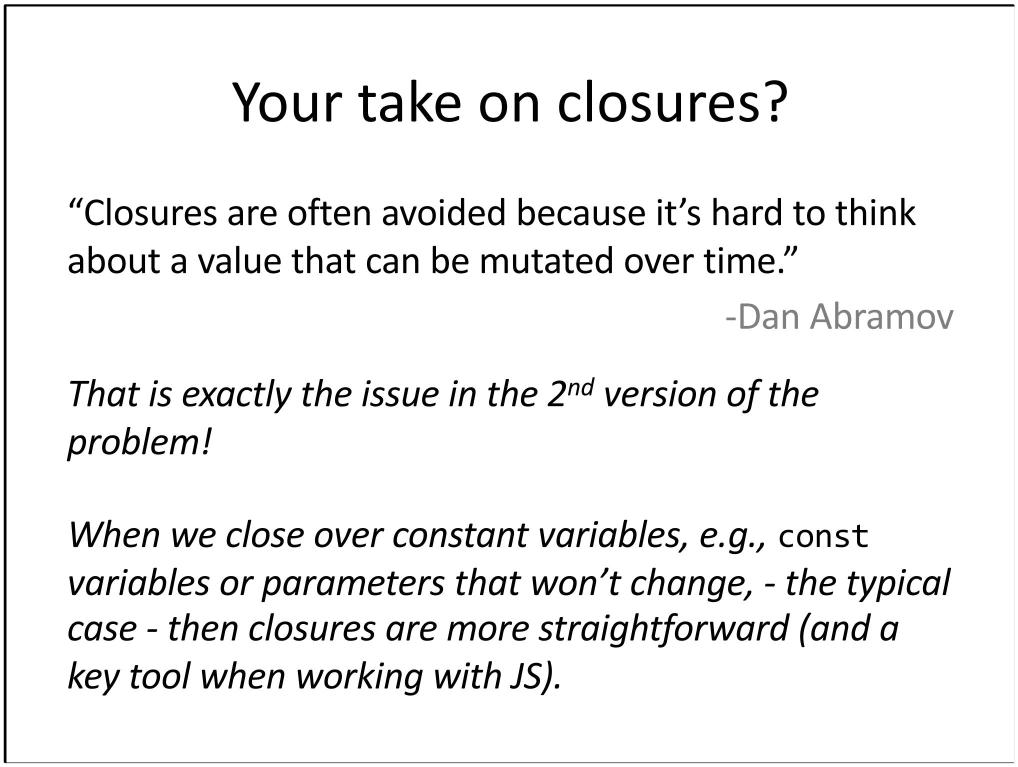
```
// Alternative
const vals = [4, 5, 6];
const funcs = vals.map((val, idx) => {
  return () => console.log(`Value ${val}@${idx}`);
});
funcs.forEach(func => func());
```

Surprised? What happened? Recall we close over variables not values. Thus, all the callbacks we create close over the same `idx` variable! We don’t have the same issue with `val`, because we each iteration of the loop is defining a new `val` variable! Could we improve this?

We could make the first loop look like the second, e.g., for (let i = 0; i < funcs.length; i++), thus closing over a new variable `i` each iteration. But even that is subtle… defining a variable with let in a for loop is a special case, it behaves as though it was defined in the loop body, thus each iteration is a new variable, but initialized with the value from the previous iteration. Ugh. How could our functional tools, e.g., map, make this clearer? That first loop is really transforming an array into another array. A good tool for map. We want to the value and its index. Map supports that through additional optional arguments to the function argument, e.g., …

```
const vals = [4, 5, 6];
const funcs = vals.map((val, idx) => {
  return () => console.log(`Value ${val}@${idx}`);
});
funcs.forEach(func => func());
```

Should we just know we can use map with those additional arguments. No. Not without looking at the documentation. Instead, we want to cultivate the sense that such a thing may be possible (after all it is common to need a value and its index) and so we should look to see if there are relevant tools in JS for that task, instead of immediately turning to a loop because that is how we would do it in another language.



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!

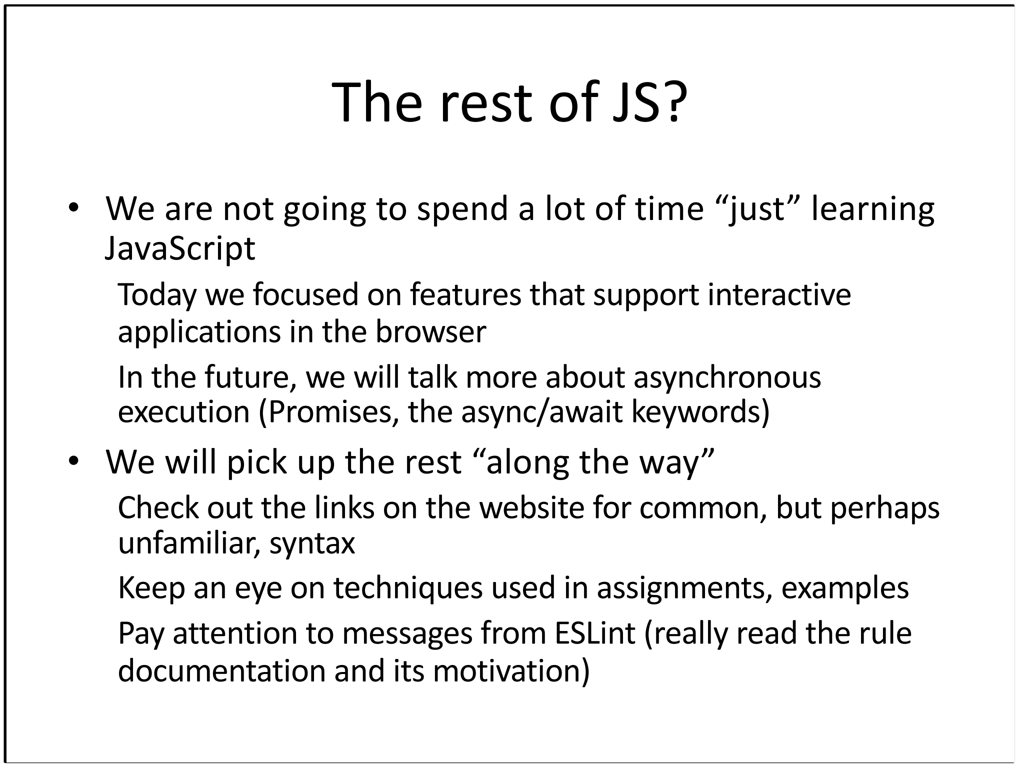
When we close over constant variables, e.g., `const` *variables or parameters that won't change, - the typical case - then closures are more straightforward (and a key tool when working with JS).*

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 JavaScript
  - Today we focused on features that support interactive applications in the browser
  - In the future, we will talk more about asynchronous execution (Promises, 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
  - Keep an eye on techniques used in assignments, examples
  - Pay attention to messages from ESLint (really read the rule documentation and its motivation)