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 teach us the gotchas, our goal in-class is the main ideas

Javascript is to Java as Hamburger is to Ham ... that is Javascript has nothing to do with Java, that was purely a marketing move (Java was very popular/prominent at the time).
ESLinter is a static analysis tool that can flag certain errors and/or bad style (like removing fuzzy “lint”). We will make extensive use of ESLint throughout the semester.

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

- `0 == 0`
- `null == null`
- `true == true`

If one of those occurs in an innocent looking statement such as `x == 0`, the actual problem is very difficult to spot.

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

Google “eslint eeqeqq”
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.
Event loop is constantly spinning executing callbacks in response to events. So if the user clicks a link that adds a click handler to the queue, when the 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 callbacks to the queue). The browser is effectively single threaded. If you have ever observed the browser hang, that is JS code monopolizing that single thread preventing the event loop from advancing.

What is a callback? A callback is a function that is executed after other code has completed, i.e. when a network request has completed. But it is not just the "next" code, 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)
- Be able to hold on to state in a function (closures)
Functions as 1st class objects ... functions are a type in the language, can be created during execution, stored in variables/data structures, passed as arguments or returned. Not a formal definition...

We see example of creating anonymous functions, both pre-ES6 style using function keyword and ES6 arrow function (concise body). By “close”, we mean we have access to the the local variables that were in scope when the functions was defined.

Will print:
1
2
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. The callback function is closing over the variable current which was set to the time just before calling setTimeout.
What does the following code print?

```javascript
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?");
```

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>First?</td>
<td>First?</td>
<td>Time elapsed (ms): 100</td>
<td>None of the above</td>
</tr>
<tr>
<td>Time elapsed (ms): 100</td>
<td>First?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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.
What does the following code print?

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

We can simplify this slightly with the following by using additional arguments to `setTimeout`, which close over the values at the time of invocation.

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

Answer: B

For example, if we rewrote that code as follows, we would get B (as we would expect). Here we are creating a function that closes over the past parameter which is set to the value of current when setTimeout is invoked, that is behind the scenes there was the equivalent of past = current. Subsequent changes to current don’t affect that value of past.

We can simplify this slightly with the following by using additional arguments to `setTimeout`, which close over the values at the time of invocation.

```javascript
current = new Date('11 Feb 2019');
```
Also 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. When we close over constant values, either “const” values or arguments that won’t get changed, then closures are a very powerful tool and fundamental part of working with the Javascript event loop.
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).
How would you implement

```javascript
function 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;
};
```