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
What is the browser doing with its time?

What is happening during this time?

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
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);
console.log("First?")
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>First?</td>
<td>First?</td>
<td>Time elapsed (ms): 100</td>
</tr>
<tr>
<td>Time elapsed (ms): 100</td>
<td>First?</td>
<td></td>
</tr>
</tbody>
</table>

The time elapsed won’t exactly be 100. It will be larger but reasonably close.
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('12 Feb 2018')
console.log("First?")
```

First?
Time elapsed (ms): 100

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 when we assign to that new variable we are assigning the value at the moment of assignment (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(callback, delay[,param1[,param2...]]) delay in ms
setTimeout((past) => {
    console.log("Time elapsed (ms): " + (Date.now() - past))
}, 100, current);

console.log("First?")
```

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>First?</td>
<td>First?</td>
<td>Time elapsed (ms): 100</td>
</tr>
<tr>
<td></td>
<td>Time elapsed (ms): 31592310870</td>
<td>Time elapsed (ms): 100</td>
<td>First?</td>
</tr>
</tbody>
</table>

For example, if we rewrote that code as follows, using the feature that additional arguments to setTimeout are passed to the callback we would get B (as we would expect). When we invoke setTimeout, we are assigning current to a parameter. When we do so we close over a variable assigned the value of current at that moment (i.e. Date.now()). And that parameter is not affected by changing current.
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
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;
    }