JavaScript Functionals

6.170 Recitation 2
Functionals
First Class Functions
High-order Functions

- Assign to variables
- Pass as arguments
- Return as results
Good Practices

- **let**: declare block-scope variables
- **const**: declare block-scope variables that cannot be reassigned
List Functionals

```javascript
> a = [1,2,3]
\[::-] ▶️ (3) [1, 2, 3]

> a.forEach((num, idx) => {
  console.log(num);
  a[idx] = idx;
});

1
2
3

\[::-] undefined

> a
\[::-] ▶️ (3) [0, 1, 2]
```
List Functionals

- **map**
  - `list.map(function(item, idx) {...})`
  - `list.map((item, idx) => {...})`
  - `array → array (new items, same length)`

- **reduce**
  - `list.reduce(function(accumulator, item, idx) {...}, initialValue)`
  - `list.reduce((accumulator, item, idx) => {…}, initialValue)`
  - `array → any (same type as accumulator & initialValue)`

- **filter**
  - `list.filter(function(item, idx) {...})`
  - `list.filter((item, idx) => {...})`
  - `array → array (same items, same length or less)`
Exercise

https://jsfiddle.net/alicezjin/s35utk19/

Given tasks and their durations, calculate the total time for all tasks that took more than 2 hours to complete.
Advantages to Iteration Abstraction

- Easier to reuse an iteration pattern.
- Reference items rather than indices (fewer mistakes!).
- Custom iterators: e.g., "for each neighbor"
Exercise

https://jsfiddle.net/alicezjin/6z7wmdvr/

Connect 4: check if the game is over
Additional Resources

http://people.csail.mit.edu/dnj/teaching/6170/javascript-live/

https://github.com/hariharsubramanyam/common_misuses_of_functionals
Closures

function Counter() {
  let c = 0;
  return {
    get: function() { return c; },
    up: function() { c++; },
    down: function() { c--; }
  }
}

count = Counter();
count.up();
count.get(); // returns 1

count.get(); // what happens?

console.log(c); // what happens?
function Counter() {
    let c = 0;
    return {
        get: function() { return c; },
        up: function() { c++; },
        down: function() { c--; }
    }
}

count = Counter();
count.up();
count.get(); // returns 1
Exercise

Write (+ draw scope diagram)

```javascript
const makeTimedFn = function(msg, fn) {};
```

It should return a function that, when called, does the following:

1. Calls fn()
2. Measures how long fn() takes to run (in ms)
3. Prints out “<msg>: <duration>”
// Assume we can access syncWithDatabase
const timedSync = makeTimedFn(
  "Sync time (ms): ",
  syncWithDatabase
);

// Print something like "Sync time (ms): 1240"
timedSync();
Exercise

Write (+ draw scope diagram)

```javascript
const makeTimedFn = function(msg, fn) {};
```

It should return a function that, when called, does the following:

1. Calls `fn()`
2. Measures how long `fn()` takes to run (in ms)
3. Prints out "<msg>: <duration>"

Hint

```javascript
const timeInMs = (new Date()).getTime();
console.log("Hello world! ", timeInMs);
```
Solution

```javascript
const makeTimedFn = function(msg, fn) {
    const timedFn = function() {
        const startMs = (new Date()).getTime();
        fn();
        const endMs = (new Date()).getTime();
        const elapsed = endMs - startMs;
        console.log(msg + " " + elapsed);
    }
    return timedFn;
};
```
Exercise

Modify `makeTimedFn` so it prints the average execution time on each invocation

```javascript
const makeTimedFn = function(msg, fn) {
    const timedFn = function() {
        const startMs = (new Date()).getTime();
        fn();
        const endMs = (new Date()).getTime();
        const elapsed = endMs - startMs;
        console.log(`$${msg} ${elapsed}`);
    }
    return timedFn;
};
```
const makeTimedFn = function(msg, fn) {
  let avgTime = 0;
  let numRuns = 0;
  const timedFn = function() {
    const startMs = (new Date()).getTime();
    fn();
    const endMs = (new Date()).getTime();
    const elapsed = endMs - startMs;
    numRuns++;
    avgTime = (avgTime * (numRuns-1)) / numRuns + elapsed / numRuns
    console.log(msg + " " + avgTime);
  }
  return timedFn;
};