JavaScript/Programming Tips, Debugging, Best Practices

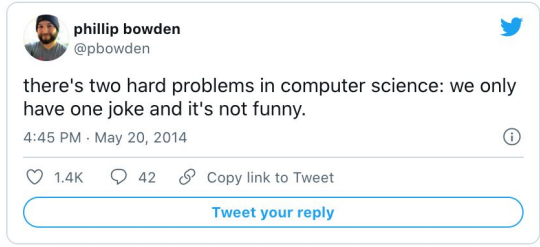
CUNY Tech Prep

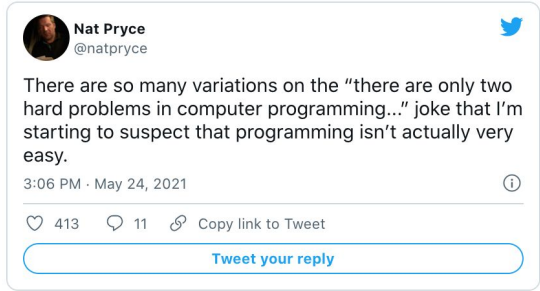
Rewrite working code for readability

- Practice writing better code with clearer structure
  a. Write code to solve a problem
  b. Treat code as a DRAFT
  c. Revise your code's structure and names for readability
     - *Assume you have to teach/present this solution to your team*
- If you're nesting loops, consider extracting the nested block into a function
- If you're getting lost in complex logic, tackle subproblems
  a. Use comments to separate the steps
  b. Or for very long code use functions

Naming things (hardest problem in CS)

- `let temp = 42;` // 😟
- `let temp2 = 23;` // 😰
- Naming things is difficult
  - Start by using meaningful words
  - Longer variable names are OK
  - `letter > char > c`
  - `word > w`
  - `event > ev > e`
- One letter variables are OK for common uses
  - Indexing, math equations, common uses
- Shorter names are easier to type, but take longer to read
  - Most of the time at work we're reading or explaining code, not writing





Printing output (*console.log() - cout >> - print() - System.out.println()*)

- **console.log()** is *typically* output for the programmer
- **`return`** - Users of your code require the output to be returned
  - Users can be a person, a script, or other application
  - Returning output allows for reusability of the code
- Using too many console.log() calls will get you lost
  - Use markers that will help you locate the messages
    - `console.log('isWordValid() line 34: ', validEntries)`

Looping

- If you know that your algorithm/solution has to access:
  - Every element in an array, object, string
  - Access is in order (forward or reverse)
  - Then use **`for-of`** loop or **`.forEach()`**, or **`.map()`**
  - Also look into: **`.filter(), .reduce()`**
- If you will scan the array with multiple indexes, non-linear indexes:
  - Back and forth, multiple passes, random-access
  - Then use a **`while`** loop
  - For-loops will not work and lead you to confusion

Nested loops

- 2-3 levels are OK and depends on the problem
- Writing functions may be easier to reason about them
- Use separate indexes for each loop
    - Always understand if nested indexes depend on each other or not

```
for(let i = 0; i < someLength; i++) {
 for(let j = 0; j < someLength; j++) {
   // ...
 }
}
```

```
for(let i = 0; i < someLength; i++) {
 for(let j = i; j < someLength; j++) {
   // ...
 }
}
```

```
for(let i = 0; i < someLength; i++) {
 for(let j = 0; j < i; j++) {
   // ...
 }
}
```

Callback hell

- Just like loops, code with nested callbacks are harder to understand
- http://callbackhell.com/
- Create functions with names that you can use and reason about independently
- Use Promises or Async/Await
    - (we'll learn about these in the Fall semester)

debugger

- **`debugger;`**
  - Using this statement will start the javascript debugger
  - Debugger sets a breakpoint
  - Let's you step through code line-by-line and watch variables
- Frontend (Browser) debugging:
  - https://www.w3schools.com/js/js_debugging.asp
- Backend (Node) debugging:
  - **`node inspect codeFile.js`**
  - Best done with a VSCode plugin
  - https://nodejs.org/dist/latest-v14.x/docs/api/debugger.html

Using Built-in Data Structures in JavaScript

Stacks

- We use arrays directly
- Only add items using push method
- Only remove items using pop method
- LIFO: Last in first out
- The top of the stack is the last element of the array

Methods to know:

- `let stack = []`
- `stack.push()`
  - Add operation, O(1) runtime
  - But, depending on underlying implementation, this may take more time if memory has to be moved or allocated
- `stack.pop()`
  - Remove operation, O(1) runtime
- `stack[stack.length - 1]`
  - Top of the stack, O(1) runtime

Queues

- We use arrays directly
- Only add items using push method
- Only remove items using shift method
- FIFO: First in first out
- The front of the queue is the first element of the array
- The end of the queue is the last element of the array

Methods to know:

- `let queue = []`
- `queue.push()`
  - Add operation, O(1) runtime
  - But, depending on underlying implementation, this may take more time if memory has to be moved or allocated
- `queue.shift()`
  - Remove operation, O(n) runtime
- `queue[0]`
  - Front of the queue, O(1) runtime
- `queue[queue.length - 1]`
  - end of the queue, O(1) runtime

Maps (Dictionaries/HashTables**)**

- Traditionally we have used Objects for this purpose
- There is a Map() object in ES6+
  - API is cleaner for this purpose. Look into it
  - There are some differences
- It is an efficient way to store and look up keys and their associated values
- Each key in the map is unique
  - All of the keys are a set

Methods to know:

- `let myMap = {}`
- `myMap[key]`
  - Get key operation, O(1) runtime
- `myMap[key] = value`
  - Set key operation, O(1) runtime
- `Object.keys(myMap)`
  - Returns an array of all keys in the map
- `Object.entries(myMap)`
  - Returns an array of all [key, value] entries

Sets

- 3 ways to implement Sets, each has tradeoff
  - Use an array
  - Use an object's keys
  - Use Set() object in ES6+
- Array implementation will make Get operations O(n)
- Object keys implementation is faster but arrays and Set are more flexible
  - This implementation is similar to previous slide where all values are true or some other ignored value

Methods to know:

- `let mySet = []`
- `mySet.includes(entry)`
  - Returns true or false if entry exists
  - O(n) runtime, linear search for entry
- `mySet.push(entry)`
  - Add an entry (make sure it's not a duplicate)
  - O(n) runtime (have to call .includes() first to check for existence)
- `mySet.length`
  - Size of set, O(1) runtime
- `mySet`
  - Array of all entries in set

Sorting

- JavaScript arrays have a built-in sorting method
- It is an *in-place* sort, which means your array will be modified
- ***Default behavior turns everything into strings***
  - Default sorting of numbers would give you:
    - [1, 10, 100, 2, 23, 3, 4, 5]
- `compareFunction(a, b)`
  - Optionally takes a callback to compare items in the array.
- Useful and flexible when you need to compare entries that are objects
- ***Runtime is dependent on implementor***
  - browser, node
- For specific runtimes or implementation you will have to write sorting from scratch

Methods to know:

- `let items = []`
- `items.sort()`
  - Sort based on items string value
- `items.sort(compareFunction)`
  - Sort based on compare function

- If `compareFunction(a, b)` returns a value > than 0, sort `b` before `a`.
- If `compareFunction(a, b)` returns a value < than 0, sort `a` before `b`.
- If `compareFunction(a, b)` returns 0, `a` and `b` are considered equal.

Useful links

- General runtime complexity (always depends on implementation details)
  - https://www.bigocheatsheet.com/
- JavaScript documentation
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript
- For ES6 and newer features (not standardized yet)
  - https://exploringjs.com/impatient-js/toc.html
- JavaScript Data Structures and Algorithms
  - ***Video of DS&A implementations in JS*** https://www.youtube.com/watch?v=t2CEgPsws3U
  - https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures/
  - Visualizations: https://www.cs.usfca.edu/~galles/visualization/Algorithms.html