Modern Application Development - II

JavaScript Collections

- Arrays
- Synchronous Iteration
- Multi-dimensional
- Maps, Sets, …
- Destructuring
- Generators

Basic Arrays

- Collection of objects of any type
    - Can even be mixed type (numbers, strings, objects, functions…)
- Element access
- Length
- Holes
- Iteration

Iteration

- Go over all elements in a collection
- Concepts:
    - **Iterable**: an object whose contents can be accessed sequentially
    - **Iterator**: pointer to the next element
- Iterable objects:
    - Array
    - String
    - Map
    - Set
    - Browser DOM - tree structure
- Objects: object.keys(), object.entries() - helper functions

Iterations and Transformations

- Functions that take functions as input
- `map, filter, find`
  - Apply a callback function over each element of array
- Elements of functional programming: create a transformation chain

Callback: important concept - function passed in to another function, to be called back for some purpose

Other Collections

- Maps: proper dictionaries instead of objects
- WeakMaps
- Sets

More advanced topics - use only if needed

Destructuring

- Simple syntax to split an array into multiple variables
- Easier to pass and collect arguments etc.
- Also possible for objects

Generators

- Functions that `yield` values one at a time
- Computed iterables
- Dynamically generate iterators
- Advanced topic - skip for now

Modularity

- Modules
- Objects
  - Prototype based inheritance

Modules

- Collect related functions, objects, values together
- “export” values for use by other scripts
- “import” values from other scripts, packages

Ways of implementing

- script - direct include script inside browser
- CommonJS - introduced for server side modules
  - synchronous load: server blocks till module loaded
- AMD - asynchronous module definition
  - browser side modules

ECMAScript 6 and above:

- ES6 Modules
  - Both servers and browsers
  - Asynchronous load

npm

- Node Package Manager
- Node:
  - command line interface for JS
  - Mainly used for backend code, can also be used for testing
- npm can also be used to package modules for frontend
  - “Bundle” managers - webpack, rollup etc.

Objects

- Everything is an object …
- Object literals
  - Assign values to named parameters in object
- Object methods
  - Assign functions that can be called on object
- Special variable **`this`**
- Function methods
  - call(), apply(), bind()
- Object.keys(), values(), entries()
  - use as dictionary
  - iterators

Prototype based inheritance

- Object can have a “prototype”
- Automatically get properties of parent
- Single inheritance track

Class

- Better syntax - still prototype based inheritance
- constructor must explicitly call super()
- Multiple inheritance or Mixins
    - Complex to implement - out of scope here

Asynchrony

- Asynchronous calls
- Asynchronous Iteration
- Basic ideas of Promises
  - async/await

Function calls

- Function is like a “branch”
    - but must save present state so we can return
- Call stack:
    - Keep track of chain of functions called up to now
    - Pop back up out of stack

- main() on stack - current - calls f()
- f() goes on stack - calls g()
- g() goes on stack - calls h()
- h() goes on stack - executes
- return from h -> pop into g
- return from g -> pop into f
- return from f -> pop into main

Call Stack

Explanatory video:

https://vimeo.com/96425312

Visualizing the call stack:

http://latentflip.com/loupe/

Event Loop and Task Queue

- Task Queue: store next task to execute
  - Tasks are pushed into queue by events (clicks, input, network etc.)
- Event loop:
  - Wait for call stack to become empty
  - Pop task out of queue and push it onto stack, start executing
- Run-to-completion
  - Guarantee from JavaScript runtime
  - Each task will run to completion before next task is invoked

Blocking the browser

https://exploringjs.com/impatient-js/ch_async-js.html#how-to-avoid-blocking-the-javascript-process

Why callbacks?

- Long running code
  - Will block execution till it finishes!
- Push long running code into a separate “thread” or “task”
  - Let main code proceed
  - Call back when completed

Example: reading files - synchronous

```
const fs = require('fs')

try {
 const data = fs.readFileSync('/Users/joe/test.txt', 'utf8')
 console.log(data)
} catch (err) {
 console.error(err)
}
```

Example: reading files - asynchronous

```
const fs = require('fs')

fs.readFile('/Users/joe/test.txt', 'utf8' , (err, data) => {
 if (err) {
   console.error(err)
   return
 }
 console.log(data)
})
```

Asynchronous Code

- Very powerful - allows JS to have high performance even though it is single threaded
- Can be difficult to comprehend
  - Focus on using async libraries and functions before writing your own
- Promises, async function calls etc.
  - Important and useful concepts
  - Deferred for now

JSON

JSON

- Object notation - for serialization, communication
- Notation is frozen
  - Means even problem cases will remain (trailing “,” etc could be useful but will not be used)
- Usage through JSON API

JSON API

- Global namespace object `JSON`
- Main methods:
  - `JSON.stringify()`
  - `JSON.parse()`