Functional programming and classes

Michael Chang

Spring 2020

Plan for today

JS detour: functional programming

- Anonymous functions, closures, callbacks

Intro to JS classes

- Constructor, methods, fields
- Public vs. private
- Callbacks for communication

Aside: using the debugger

If time, JS modules

- Separating code into files
- import and export

Aside: global scope

Browser's global variables

- `window` (this actually contains the globals)
  - `window.foo` makes `foo` global, even inside functions
- `document`
- Variables (incl. functions) declared at top level
- References to elements by ID
  - Best practice: My understanding is that querySelector is more accepted

Functional programming in JS

JavaScript has first-class functions

- Functions are objects like any other
- Can be created, assigned, reassigned, stored
- Similar to Python, unlike C
  - C++ and Java have some recent support

Functional programming

- In contrast to imperative programming
- Create, chain, and compose functions
- Fairly popular style (e.g. React)

Functional programming in JS

We've already seen some of this

- We create functions with = operator
  - Note: `function` keyword is a more imperative style
- We passed a function to `addEventListener`
  - Referred to as a "callback function"
  - The browser "calls us back" by calling our function when the event fires

Example

Imperative

```
const timesTwo = (arr) => {
  let result = [];
  for (let elem of arr)
    result.push(elem * 2);
  return result;
};
```

Example

Functional

```
const timesTwoElem = (num) => {
  return num * 2;
};
const timesTwo = (arr) => {
  return arr.map(timesTwoElem);
};
```

Functional JS syntax

A few syntactic shorthands

Can drop parens around argument if only one arg

Can drop braces and "return" if only one expression

Instead of

```
const timesTwoElem = (num) => {
  return num * 2;
};
```

We can write

```
const timesTwoElem = num => num * 2;
```

Functional JS syntax

Anonymous functions (lambdas)

Don't need to assign to variable and give name

Instead of

```
const timesTwoElem = num => num * 2;
const timesTwo = (arr) => {
  return arr.map(timesTwoElem);
};
```

We can write

```
const timesTwo = arr =>
    arr.map(num => num * 2);
```

Useful JS functions

`arr.map(fn)`

Call fn on each element, return new array with results

`arr.sort([cmp])`

Sort array, cmp is called with two elems and returns negative, zero, or positive for <, =, or >

`Object.fromEntries(arr)`

Turn an array of pairs into an Object

More useful JS functions

Definition: predicate function

Returns true or false, typically takes one argument

Arg "satisfies" or "matches" predicate if it returns true

`arr.filter(pred)`

Return array with only elems satisfying pred

`arr.find(pred)`

Return first elem satisfying pred

`arr.every(pred)`

Returns true if every element satisfies pred

`arr.some(pred)`

Return true if at least one elem satisfies pred

Closures

Functions can be nested

Nested functions have access to outer function vars

Example

```
const graduating = (applied, units) => {
  const canGraduate = student =>
    units[student] >= 180 &&
       meetsReqs(student);
  return applied.filter(canGraduate);
};
```

Style: functional vs. imperative

Many strong opinions

- Some buy in heavily to functional paradigm
- Others stick more to OO or procedural style

Some of it is ubiquitous

- You'll see a fair amount of map, some filter
- Closures get thrown around quite a bit too

Use what you know

- For your projects, use what you're comfortable with
- When working with others / on larger projects, best follow similar/consistent style

Next up

JS detour: functional programming

- Anonymous functions, closures, callbacks

Intro to JS classes

- Constructor, methods, fields
- Public vs. private
- Callbacks for communication

Aside: using the debugger

If time, JS modules

- Separating code into files
- import and export

Using what we just learned

```
const Counter = (start = 0) => {
  let count = start;
  const value = () => count;
  const add = (n = 1) => { count += n; };
  return { value, add };
};
let c = Counter();
c.add(5);
console.log(c.value());
let c2 = Counter();
console.log(c2.value());
```

Using what we just learned

One way to create a class-like object

- Return "methods", contains "instance variables"
- Some projects/styles use this
- But we have real classes, with more features
- Best practice: use real classes

JS class syntax

```
class Counter {
  constructor(start = 0) {
    this._count = start;
  }
  value() { return this._count; }
  add(n = 1) { this._count += n; }
}

let c = new Counter(10);
c.add(5);
console.log(c.value());
```

JS classes

MDN class syntax

`constructor`

Special method name, called by `new`

Methods

Define in class body

Fields (instance variables)

Accessed through `this`

Initialize in constructor (or method)

Can add/delete dynamically

JS classes

Visibility

- Everything is "public" (like Python)
- Convention: prefix with _ for "private" members
  - Don't access fields/methods starting with _ from outside

this keyword

Problem

`elem.addEventListener(..., this._method);`

When `_method` is called, `this` is `undefined`!

Cause (summary)

this gets its value at time of call

```
obj.foo() -> this === obj
foo() -> this === undefined
let bar = obj.foo; // Not a call, just assigns the fn
bar(); -> this === undefined
```

this keyword

Problem

```
elem.addEventListener(..., this._method);
```

When `_method` is called, `this` is `undefined`!

Solution

```
elem.addEventListener(...,
    this._method.bind(this));
```

bind() "locks" the value of this

Another solution

In constructor:

```
this._method = this._method.bind(this);
```

Do this for all event handlers

Not needed for methods called normally

this keyword

Problem

`elem.addEventListener(..., this._method);`

When `_method` is called, `this` is `undefined`!

Solution

```
elem.addEventListener(...,
    this._method.bind(this));
```

bind() "locks" the value of this

Another solution (a bit cleaner)

In constructor:

```
this._method = this._method.bind(this);
```

Do this for all event handlers

Not needed for methods called normally

Style tips for classes

Bind event handlers in constructor

- Avoid repetition or forgetting

Methods vs. lambdas

- For short functions used once, lambdas can be nice
- Core functionality of the class should probably go in a method

Encapsulation

- Keep instance variables "private" when you can
- But trivial getters/setters are probably unnecessary

Communication

- Use callbacks or references to pass data between classes
  - But each class should define a clean abstraction

Summary

Today

- Functional programming
- Classes

Next time

- Modules
- JSON and data representation
- Reading data from a source (file, API)