Classes in JavaScript

Amateur JavaScript

So far the JavaScript code we've been writing has looked like this:

- Mostly all in one file
- All global functions
- Global variables to save state between events

It would be nice to write code in a **modular** way...

```
//
// Album view functions
//
let currentIndex = null;
function onThumbnailClick(event) {
  currentIndex = event.currentTarget.dataset.index;
  const image = createImage(event.currentTarget.src);
  showFullsizeImage(image);
  document.body.classList.add('no-scroll');
  modalView.style.top = window.pageYOffset + 'px';
  modalView.classList.remove('hidden');
}

//
// Photo view functions
//
function createImage(src) {
  const image = document.createElement('img');
  image.src = src;
  return image;
}

function showFullsizeImage(image) {
  modalView.innerHTML = '';

  image.addEventListener('pointerdown', startDrag);
  image.addEventListener('pointermove', duringDrag);
  image.addEventListener('pointerup', endDrag);
  image.addEventListener('pointercancel', endDrag);
  modalView.appendChild(image);
}

let originX = null;
function startDrag(event) {
  event.preventDefault();
  // Needed so clicking on picture doesn't cause modal dialog to close.
  event.stopPropagation();

  originX = event.clientX;
  event.target.setPointerCapture(event.pointerId);
}

function duringDrag(event) {
  if (originX) {
    const currentX = event.clientX;
    const delta = currentX - originX;
    const element = event.currentTarget;
    element.style.transform = 'translateX(' + delta + 'px)';
  }
}

function endDrag(event) {
  if (!originX) {
    return;
  }

  const currentX = event.clientX;
  const delta = currentX - originX;
  originX = null;

  let nextIndex = currentIndex;
  if (delta < 0) {
    nextIndex++;
  } else {
    nextIndex--;
  }

  if (nextIndex < 0 || nextIndex == PHOTO_LIST.length) {
    event.currentTarget.style.transform = '';
    return;
  }
```

ES6 classes

We can define **classes** in JavaScript using a syntax that is similar to Java or C++:

```
class ClassName {
  constructor(params) {
    ...
  }
  methodName() {
    ...
  }
  methodName() {
    ...
  }
}
```

These are often called "**ES6 classes**" or "**ES2015 classes**" because they were introduced in the EcmaScript 6 standard, the 2015 release

- Recall that EcmaScript is the standard; JavaScript is an implementation of the EcmaScript standard

Wait a minute…

Wasn't JavaScript created in 1995?

And classes were introduced… 20 years later in 2015?

Q: Was it seriously not possible to create classes in JavaScript before 2015?!

Objects in JavaScript

In JavaScript, there are several ways to create blueprints for objects. Two broad approaches:

1. Functional
    a. This approach has existed since the creation of the JavaScript
    b. Weird syntax for people used to languages like Java, C++, Python
    c. Doesn't quite behave the same way as objects in Java, C++, Python
2. Classical
    a. This is the approach that just got added to the language in 2015
    b. Actually just "syntactic sugar" over the functional objects in JavaScript, so still a little weird
    c. But syntax is much more approachable

Public methods

```
class ClassName {
  constructor(params) {
    ...
  }
  methodName() {
    ...
  }
  methodName() {
    ...
  }
}
```

`constructor` is optional.

Parameters for the constructor and methods are defined in the same they are for global functions.

You do not use the `function` keyword to define methods.

Public methods

```
class ClassName {
  constructor(params) {
    ...
  }
  methodOne() {
    this.methodTwo();
  }
  methodTwo() {
    ...
  }
}
```

Within the class, you must always refer to other methods in the class with the **`this.`** prefix.

Public methods

```
class ClassName {
  constructor(params) {
    ...
  }
  methodName() {
    ...
  }
  methodName() {
    ...
  }
}
```

All methods are **public**, and you **cannot** specify private methods… yet.

Public fields

```
class ClassName {

  constructor(params) {
    this.fieldName = fieldValue;
    this.fieldName = fieldValue;
  }

  methodName() {
    this.fieldName = fieldValue;
  }
}
```

Define public fields by setting **`this.`*fieldName*** in the constructor… or in any other function.

(This is slightly hacky underneath the covers and there is a draft to add public fields properly to ES.)

Public fields

```
class ClassName {
  constructor(params) {
    this.someField = someParam;
  }
  methodName() {
    const someValue = this.someField;
  }
}
```

Within the class, you must always refer to fields with the `this.` prefix.

Instantiation

Create new objects using the new keyword:

```
class SomeClass {
  ...
  someMethod() { … }
}

const x = new SomeClass();
const y = new SomeClass();
y.someMethod();
```

Why classes?

Why are we even doing this?

Why do we need to use classes when web programming?

Why can't we just keep doing things the way we've been doing things, with global functions and global variables?

Why classes?

A: All kinds of reasons

- For a sufficiently small task, globals variables, functions, etc. are fine
- But for a larger website, your code will be hard to understand **and** easy to break if you do not organize it
- Using classes and object-oriented design is the most common strategy for organizing code

E.g. in the global scope, it's hard to know at a variable called "name" would be referring to, and any function could accidentally write to it.

- But when defined in a Student class, it's inherently clearer what "name" means, and it's harder to accidentally write that value

Organizing code

Well-engineered software is well-organized software:

- Software engineering is all about knowing
  1. What to change
  2. Where to change it
- You can read an existing codebase better if it is well-organized
  - *"Why do I need to read a codebase?"* Because you need to modify the codebase to add features and fix bugs

Other problems with globals

Having a bunch of loose variables in the global scope is asking for trouble

- Much easier to hack
  - Can access via extension or Web Console
  - Can override behaviors
- Global scope gets polluted
  - What if you have two functions with the same name? One definition is overridden without error
- Very easy to modify the wrong state variable

All these things are much easier to avoid with classes