CS193X: Web Programming Fundamentals

Spring 2017

Victoria Kirst
(vrk@stanford.edu)

Course logistics

Remember how we said the following on Day 1?

"This is the first ever offering of CS193X, meaning:

- **Everything is subject to change.**"

→ **We're making some changes to the schedule!**

Grades

Homework: ~~*60%*~~ **65%**

~~*Mini-HWs: 5%*~~

Final Project: 35%

- **We're dropping Mini-Homeworks**: Too much hassle for everyone. We're totally ignoring the first mini-HW you turned in for HW1. Might try again next year.

CS193X Structure

"Homework 0" + ~~*6 homeworks*~~ 5 homeworks

- Each homework will be a standalone web page or a very small standalone web app
- ~~*Each homework with have a multiple choice "mini-homework" attached to it*~~

1 final project

- Choice of open-ended OR structured
  - Basically you can do HW6 for your final project
- ~2 week in scope; **individual** project; no groups

0 exams

- No final, no midterm, no exams

Yes, another HW extension

Tentative schedule for the rest of the quarter:

~~Fri May 5~~

Mon, May 8:

- HW3 due -- Moved from this Friday to next Monday!
- HW4 goes out

Wed, May 17:

- HW4 due
- HW5 goes out

Tentative schedule

Tentative schedule for the rest of the quarter:

Fri, May 26

HW5 due

Final Project goes out

Wed, June 7:

Last day of lecture!

Mon, June 12

Final project due EOD: No late submissions

Disclaimer

This is the plan for the rest of the quarter.

However, there's still a lot of quarter left!

Everything I just said is still subject to change.

Classes in JavaScript

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 methods

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

As far as I can tell, private methods aren't in the language only because they are still figuring out the spec for it. (They will figure out private fields first.)

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.

Public fields

```
class ClassName {

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

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

You cannot define private fields… yet.

(Again, there are plans to add add private fields to ES once the spec is finalized.)

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

Example: Present

Let's create a `Present` class inspired by our present example from last week.

Starter / Finished

How to design classes

You may be wondering:

- How do I decide what classes to write?
- How do I decide what methods to add to my class?

Disclaimer

This is not a software engineering class, and this is not an object-oriented design class.

As such, we will not grade your OO design skills.

However, this also means we won't spend too much time explaining *how* to break down your app into well-composed objects.

(It takes practice and experience to get good at this.)

One general strategy

"Component-based" approach: Use classes to add functionality to HTML elements ("components")

Each component:

- Has exactly one container element / root element
- Handles attaching/removing event listeners
- Can own references to child components / child elements

(Similar strategy to ReactJS, Custom Elements, many other libraries/frameworks/APIs before them)

Container element

One pattern:

```
<div id="present-container"></div>

const element =
  document.querySelector('#present-container');
const present = new Present(element);
// Immediately renders the present
```

Container element

A similar pattern:

```
<div id="present-container"></div>

const element =
  document.querySelector('#present-container');
const present = new Present();
// Renders with explicit call
present.renderTo(element);
```

Web: Almost total freedom

Unlike most app platforms (i.e. Android or iOS), you have almost total freedom over exactly how to organize your code

Pros:

- Lots of control!

Cons:

- Lots and lots and lots of decisions to make

Web: Almost total freedom

Unlike most app platforms (i.e. Android or iOS), you have almost total freedom over exactly how to organize your code

Pros:

- Lots of control!

Cons:

- Lots and lots and lots of decisions to make
- This is why **Web Frameworks** are so common: A web framework just make a bunch of software engineer decisions for you ahead of time (+provides starter code)

Don't forget `this`

```
// Create image and append to container.
const image = document.createElement('img');
image.src = 'https://s3-us-west-2.amazonaws.com/s.cdpn.io/1083533/gift-icon.png';
image.addEventListener('click', this._openPresent);
```

If the event handler function you are passing to `addEventListener` is a method in a class, you must pass "`this.`***functionName***" (finished)

"Private" with _

A somewhat common JavaScript coding convention is to add an underscore to the beginning or end of private method names:

```
_openPresent() {
   ...
}
```

I'll be doing this in this class for clarity, but note that it's frowned upon by some.

Solution: Present

CodePen finished

`this` in event handler

```
class Present {
  constructor(containerElement) {
    this.containerElement = containerElement;

    // Create image and append to container.
    const image = document.createElement('img');
    image.src = 'https://s3-us-west-2.amazonaws.com/s.cdpn.io/1083533/gift-icon.png';
    image.addEventListener('click', this._openPresent);
    this.containerElement.append(image);
  }

  _openPresent(event) {
    const image = event.currentTarget;
    image.src = 'https://media.giphy.com/media/27ppQUOxe7KlG/giphy.gif';
    image.removeEventListener('click', this._openPresent);
  }
}
```

Right now we access the image we create in the constructor in `_openPresent` via `event.currentTarget`.

`this` in event handler

```
  _openPresent(event) {
    this.image.src = 'https://media.giphy.com/media/27ppQUOxe7KlG/giphy.gif';
    this.image.removeEventListener('click', this._openPresent);
  }
}
```

Q: What if we make the `image` a field and access it `_openPresent` via `this.image` instead of `event.currentTarget`?

`this` in event handler

Error message!

CodePen / Debug

What's going on?

JavaScript `this`

The `this` keyword in JavaScript is **dynamically assigned**, or in other words: `this` means different things in different contexts (mdn list)

- In our constructor, `this` refers to the instance
- When called in an event handler, `this` refers to… the element that the event handler was attached to (mdn).

`this` in event handler

```
  _openPresent(event) {
    this.image.src = 'https://media.giphy.com/media/27ppQUOxe7KlG/giphy.gif';
    this.image.removeEventListener('click', this._openPresent);
  }
}
```

That means `this` refers to the `<img>` element, not the instance variable of the class...

...which is why we get this error message.

Solution: `bind`

To make `this` always refer to the instance object for a method in the class (i.e. to get `this` to behave as you'd expect), you can add the following line of code in the `constructor`:

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

```
class Present {
  constructor(containerElement) {
    this.containerElement = containerElement;

    // Bind event listeners.
    this._openPresent = this._openPresent.bind(this);
```

Solution: `bind`

Now `this` in the `_openPresent` method refers to the instance object (CodePen / Debug):

```
_openPresent(event) {
  this.image.src = 'https://media.giphy.com/media/27ppQUOxe7KlG/giphy.gif';
  this.image.removeEventListener('click', this._openPresent);
}
```

Moral of the story:

Don't forget to `bind()` event listeners in your constructor!!

```
class Present {
  constructor(containerElement) {
    this.containerElement = containerElement;

    // Bind event listeners.
    this._openPresent = this._openPresent.bind(this);
```

One more time:

Don't forget to `bind()` event listeners in your constructor!!

Communicating between classes

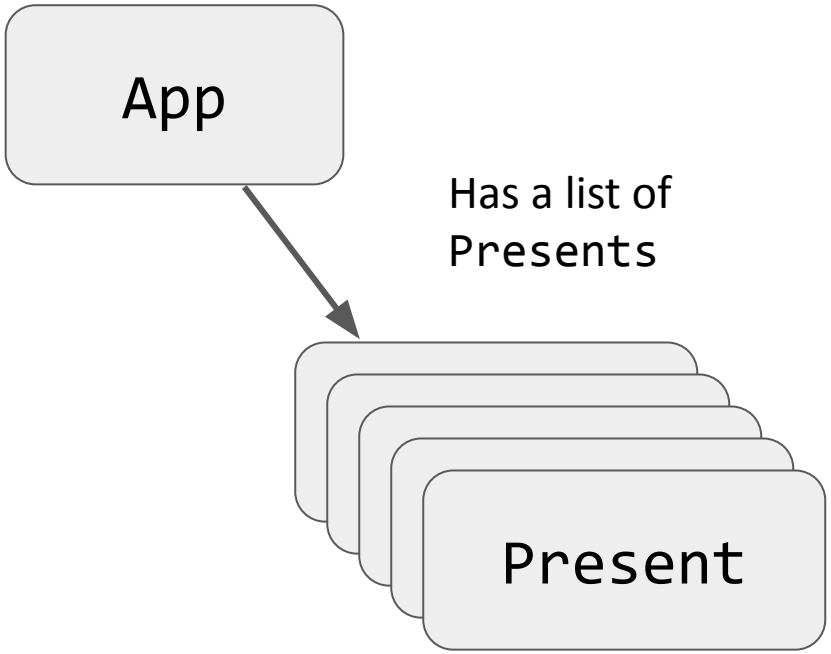
Multiple classes

Let's say that we have multiple presents now (CodePen):

Multiple classes

And we have implemented this with two classes:

- App: Represents the entire page
  - `Present`: Represents a single present



CodePen

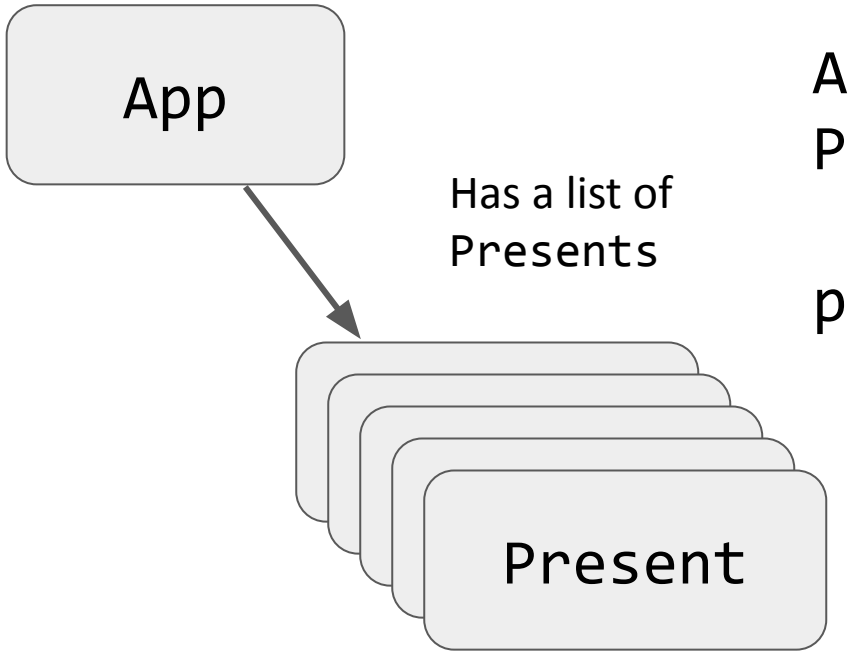
Communicating btwn classes

What if we want to change the **title** when all present have been opened? (CodePen)

Enjoy your presents!

Communication btwn classes

Communicating from App → `Present` is easy, since App has a list of the `Present` objects.

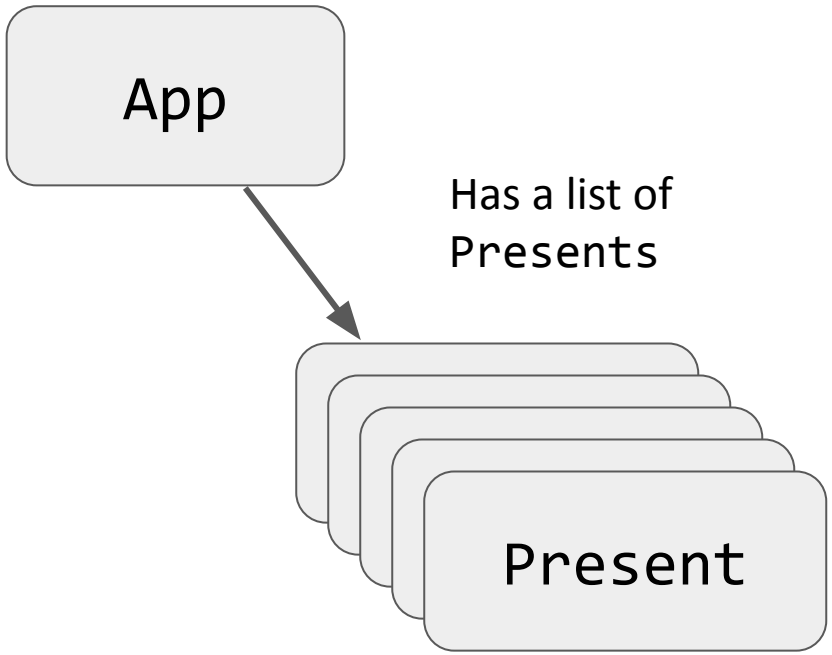


App can just call methods on `Present`:

```
present.doWhatever();
```

Communication btwn classes

However, communicating `Present` → `App` is not as easy, because `Presents` do not have a reference to App



Communicating btwn classes

You have three general approaches:

1. Add a reference to App in Photo

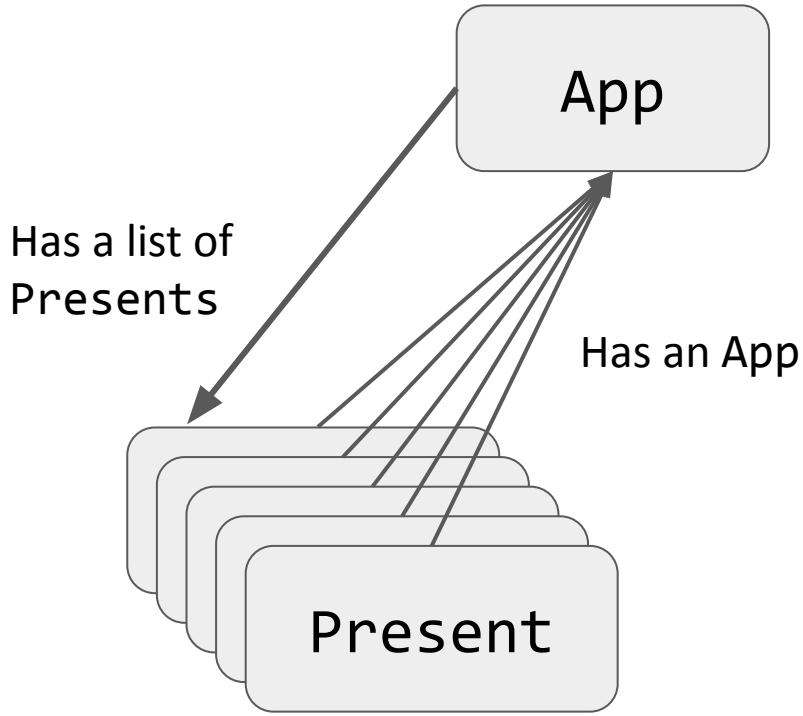
   This is poor software engineering, though we will allow it on the homework because this is not an OO design class
2. Fire a custom event

   OK (don't forget to bind)
3. Add onOpened "callback function" to Present

   Best option (don't forget to bind)

Terrible style: Presents own App

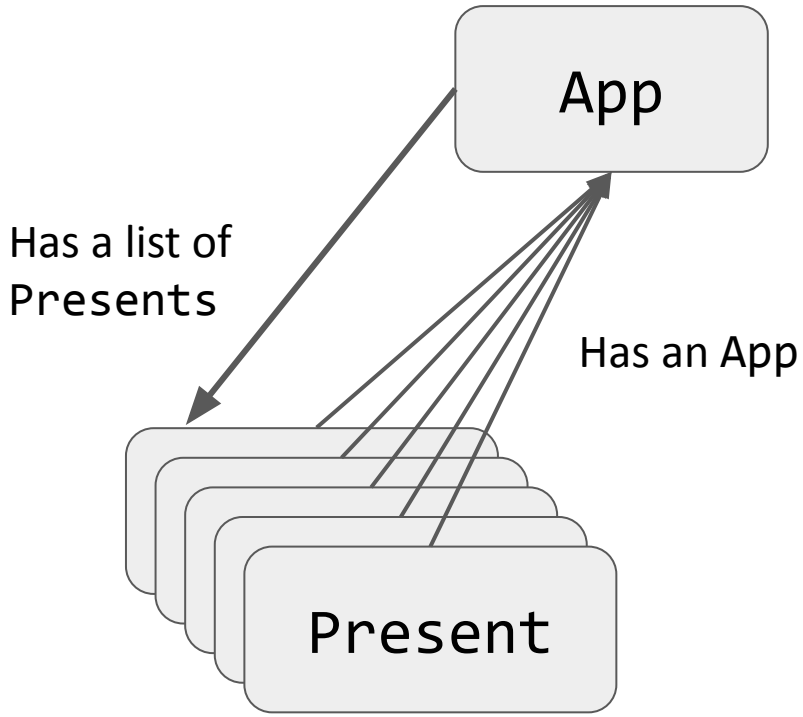
A naive fix is to just give `Present` a reference to App in its constructor: CodePen



(Please don't do this.)

Terrible style: Presents own App

This is the easiest workaround, but **it's terrible software engineering.**



- Logically doesn't make sense: a `Present` doesn't have an App
- Gives `Present` way too much access to App
- Especially bad in JS with no private fields/ methods yet

Custom events

Custom Events

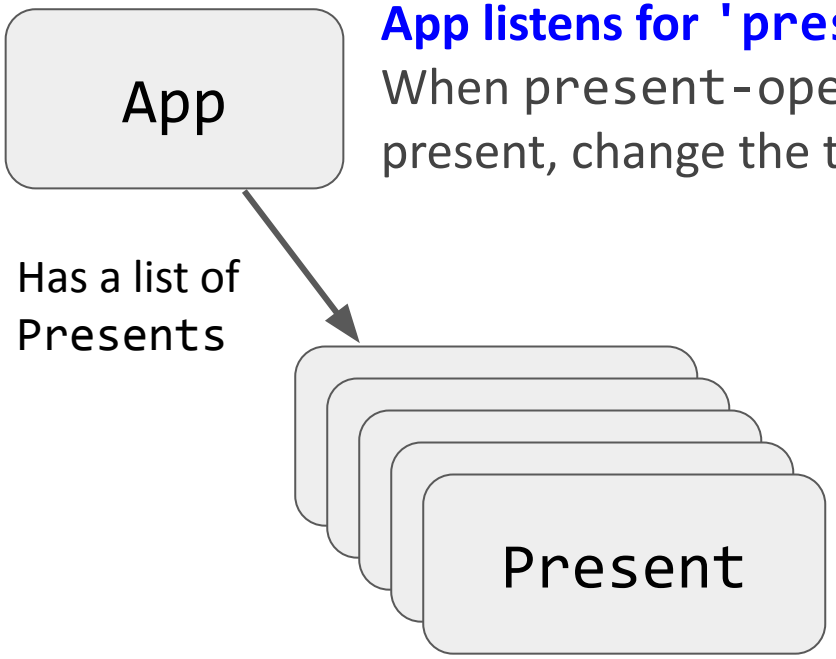
You can listen to and dispatch Custom Events to communicate between classes (mdn):

```
const event = new CustomEvent(
    eventNameString, optionalParameterObject);
element.addEventListener(eventNameString);
element.dispatchEvent(eventNameString);
```

However, CustomEvent **can only be listened to / dispatched on HTML elements,** and not on arbitrary class instances.

Custom Events: Present example

Let's have the App listen for the `'present-open'` event...

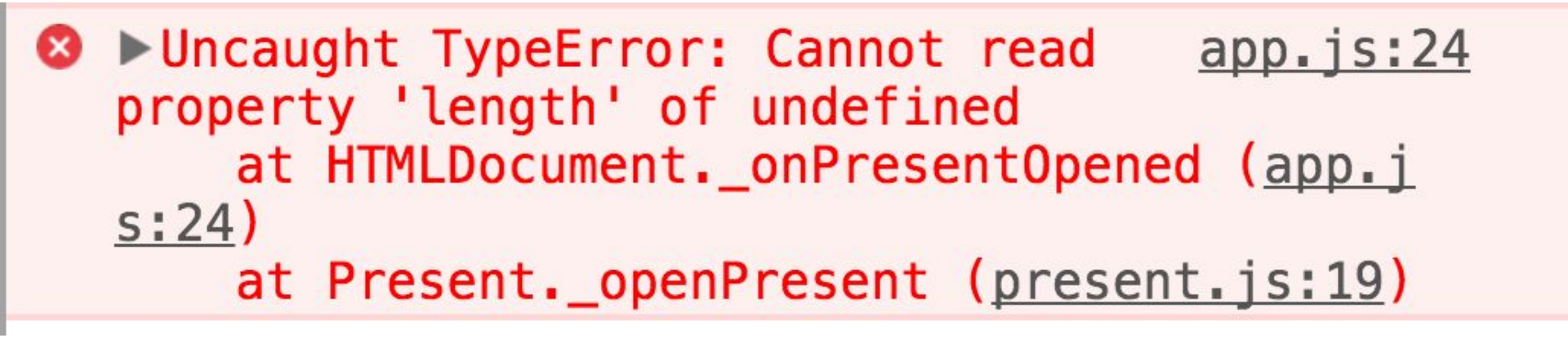


App listens for `'present-open'`
When `present-open` has fired for each present, change the title.

Each present fires `'present-open'` when clicked

[CodePen attempt]

`this` in event handler



Our first attempt at solution results in errors again!

([CodePen attempt])

Solution: `bind`

To make `this` always refer to the instance object for a method in the class (i.e. to get `this` to behave as you'd expect), you can add the following line of code in the `constructor`:

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

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

CodePen solution

First-class functions

Recall: `addEventListener`

Over the last few weeks, we've been using **functions** as a parameter to `addEventListener`:

```
dragon.addEventListener(
    'pointerdown', onDragStart);

image.addEventListener(
    'click', this._openPresent);
```

First-class functions

JavaScript is a language that supports **first-class functions**, i.e. functions are treated like variables of type Function:

- Can be passed as parameters
- Can be saved in variables
- Can be defined without a name / indentifier
  - Also called an **anonymous function**
  - Also called a **lambda function**
  - Also called a **function literal value**

Function variables

You can declare a function in several ways:

```
function myFunction(params) {
}

const myFunction = function(params) {
};

const myFunction = (params) => {
};
```

Function variables

```
function myFunction(params) {
}

const myFunction = function(params) {
};

const myFunction = (params) => {
};
```

Functions are invoked in the same way, regardless of how they were declared:

```
myFunction();
```

Simple, contrived example

```
function greetings(greeterFunction) {
  greeterFunction();
}

const worldGreeting = function() {
  console.log('hello world');
};

const hawaiianGreeting = () => {
  console.log('aloha');
};

greetings(worldGreeting);
greetings(hawaiianGreeting);
```

CodePen

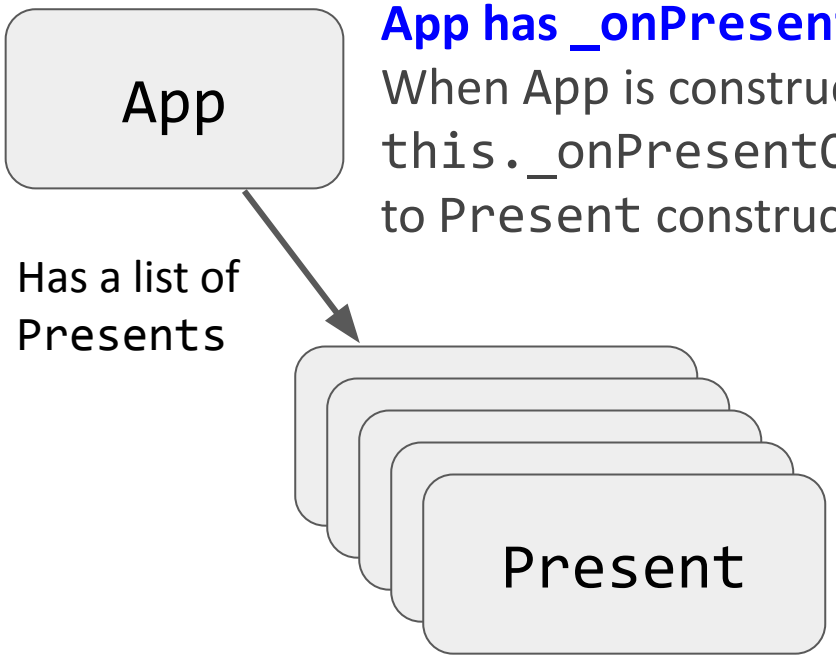
A real example: Callbacks

Another way we can communicate between classes is through callback functions:

- **Callback**: A function that's passed as a parameter to another function, usually in response to something.

Callback: Present example

Let's have `Presents` communicate with App via callback `parameter`: (CodePen attempt)



App has `_onPresentOpened` method

When App is constructing `Presents`, pass its `this._onPresentOpened` method as parameter to `Present` constructor

Each `Present` saves an `onOpenCallback` parameter in the constructor

When the present is opened, fire the callback

`this` in event handler

Say, it's another error in our event handler...

Solution: `bind`

Unless explicitly bound, "`this`" refers to the object that owns the method being called.

To make `this` always refer to the instance object for a method in the class (i.e. to get `this` to behave as you'd expect), you can add the following line of code in the `constructor`:

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

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

CodePen solution

Object-oriented photo album

Let's look at an object-oriented version of the photo album:

CodePen / Debug