

# Interactive Web Programming

1st semester of 2021

Murilo Camargos  
(**[murilo.filho@fgv.br](mailto:murilo.filho@fgv.br)**)

Heavily based on [Victoria Kirst](#) slides

# Today's schedule

## Today

- Mobile events
- Simple CSS animations
- Classes and objects in JavaScript
- `this` keyword and `bind`

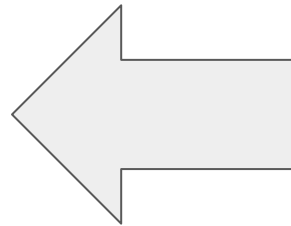
## Next week:

- Custom events
- `this` keyword and `bind`
- First-class functions
- Callbacks and functional programming

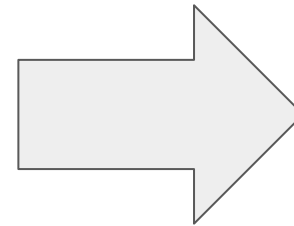
# Custom swipe events

- There are no gesture events in JavaScript (yet).
- That means there is no "Left Swipe" or "Right Swipe" event we can listen to. (Note that [drag](#) does not do what we want, nor does it work on mobile)

To get this behavior, we must implement it ourselves.



Next



Previous

# transform

[transform](#) is a strange but powerful CSS property that allow you to translate, rotate, scale, or skew an element.

<code>transform: translate(<i>x</i>, <i>y</i>)</code>	Moves element relative to its natural position by <i>x</i> and <i>y</i>
<code>transform: translateX(<i>x</i>)</code>	Moves element relative to its natural position horizontally by <i>x</i>
<code>transform: translateY(<i>y</i>)</code>	Moves element relative to its natural position vertically by <i>y</i>
<code>transform: rotate(<i>deg</i>)</code>	Rotates the element clockwise by <i>deg</i>
<code>transform: rotate(10deg) translate(5px, 10px);</code>	Rotates an element 10 degrees clockwise, moves it 5px down, 10px right

## [Examples](#)

# translate vs position

Can't you use relative or absolute positioning to get the same effect as translate? What's the difference?

- translate is much faster
- translate is optimized for animations

See comparison ([article](#)):

- [Absolute positioning](#) (click "10 more macbooks")
- [transform: translate](#) (click "10 more macbooks")

# style attribute

The `style` attribute has **higher precedence** than any CSS property.

To undo a style set via the `style` attribute, you can set it to the empty string:

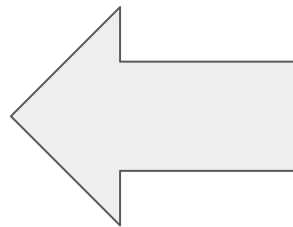
```
element.style.transform = '';
```

Now the element will be styled according to any rules in the CSS file(s).

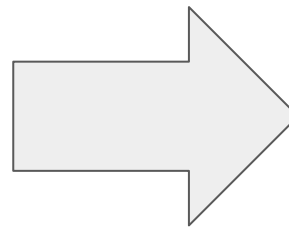
# Softening the edges

This is mostly a perception issue. We can make the UI **feel** a little smoother if we added some animations.

- The image should **slide in from the left** if we are going to the previous picture
- The image should **slide in from the right** if we are going to the next picture



Next



Previous

# CSS animations



# CSS animations syntax

```
@keyframes animation-name {  
  from {  
    CSS styles  
  }  
  to {  
    CSS styles  
  }  
}
```

[Examples](#)

Then set the following CSS property:

```
animation: animation-name duration;
```

# Example: Fade in

```
#album-view img {  
  animation: fadein 0.5s;  
}
```

```
@keyframes fadein {  
  from {  
    opacity: 0;  
  }  
  to {  
    opacity: 1;  
  }  
}
```

# CSS animations events

You can listen to animation events ([mdn](#)):

- **animationstart**: fires at the beginning of the animation
- **animationend**: fires at the end of the animation

```
const image = document.querySelector('img');  
image.addEventListener('animationstart', onStart);  
image.addEventListener('animationend', onEnd);  
  
image.classList.add('fade-grow');
```

[CodePen example](#)

# CSS animations

There are all kinds of customizations ([mdn](#)):

- Set multiple keyframes
- Set keyframes by percentage
- Make animations repeat
- Make animations alternate
- Change the timing function

Also note that not all CSS is animatable: [see list](#)

[Fancy CodePen example](#)

(credit [CSS tricks](#) -- check out their article for more details)

# CSS transitions

You can also set a **CSS transition** on an element, which controls the animation speed of a changing CSS property ([mdn](#))

```
transition: Ns;
```

[CodePen example](#)

Finished result:

[photo-mobile-finished.html](#)

# 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...

```
1 //
2 // Album view functions
3 //
4 let currentIndex = null;
5 function onThumbnailClick(event) {
6   currentIndex = event.currentTarget.dataset.index;
7   const image = createImage(event.currentTarget.src);
8   showFullSizeImage(image);
9   document.body.classList.add('no-scroll');
10  modalView.style.top = window.pageYOffset + 'px';
11  modalView.classList.remove('hidden');
12 }
13
14 //
15 // Photo view functions
16 //
17 function createImage(src) {
18   const image = document.createElement('img');
19   image.src = src;
20   return image;
21 }
22
23 function showFullSizeImage(image) {
24   modalView.innerHTML = '';
25
26   image.addEventListener('pointerdown', startDrag);
27   image.addEventListener('pointermove', duringDrag);
28   image.addEventListener('pointerup', endDrag);
29   image.addEventListener('pointercancel', endDrag);
30   modalView.appendChild(image);
31 }
32
33 let originX = null;
34 function startDrag(event) {
35   event.preventDefault();
36   // Needed so clicking on picture doesn't cause modal dialog to close.
37   event.stopPropagation();
38
39   originX = event.clientX;
40   event.target.setPointerCapture(event.pointerId);
41 }
42
43 function duringDrag(event) {
44   if (originX) {
45     const currentX = event.clientX;
46     const delta = currentX - originX;
47     const element = event.currentTarget;
48     element.style.transform = 'translateX(' + delta + 'px)';
49   }
50 }
51
52 function endDrag(event) {
53   if (!originX) {
54     return;
55   }
56
57   const currentX = event.clientX;
58   const delta = currentX - originX;
59   originX = null;
60
61   let nextIndex = currentIndex;
62   if (delta < 0) {
63     nextIndex++;
64   } else {
65     nextIndex--;
66   }
67
68   if (nextIndex < 0 || nextIndex == PHOTO_LIST.length) {
69     event.currentTarget.style.transform = '';
70     return;
71   }
72 }
```



# 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

# 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

**This approach is quite controversial.**

# Class controversy

"There is one thing I am certain is a bad part, a very terribly bad part, and that is the new class syntax [in JavaScript]... [T]he people who are using `class` will go to their graves never knowing how miserable they were." ([source](#))

-- Douglas Crockford, author of *JavaScript: The Good Parts*; prominent speaker on JavaScript; member of [TC39](#) (committee that makes ES decisions)



# Functional approach: next week!

## **Today:**

- We will check out ES6 classes.

## **Next week:**

- We will explore "functional JavaScript," allowing us to understand a way to create object factories without classes.

## **In this class:**

- We will use ES6 classes because the syntax is significantly simpler.

Back to classes!

# Public methods

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

constructor is optional.

Parameters for the constructor and methods are defined in the same way 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.

**It is stage 3 already!**

# 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, global 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.



[Solution](#)

# 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"`

# "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/27ppQU0xe7KlG/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

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

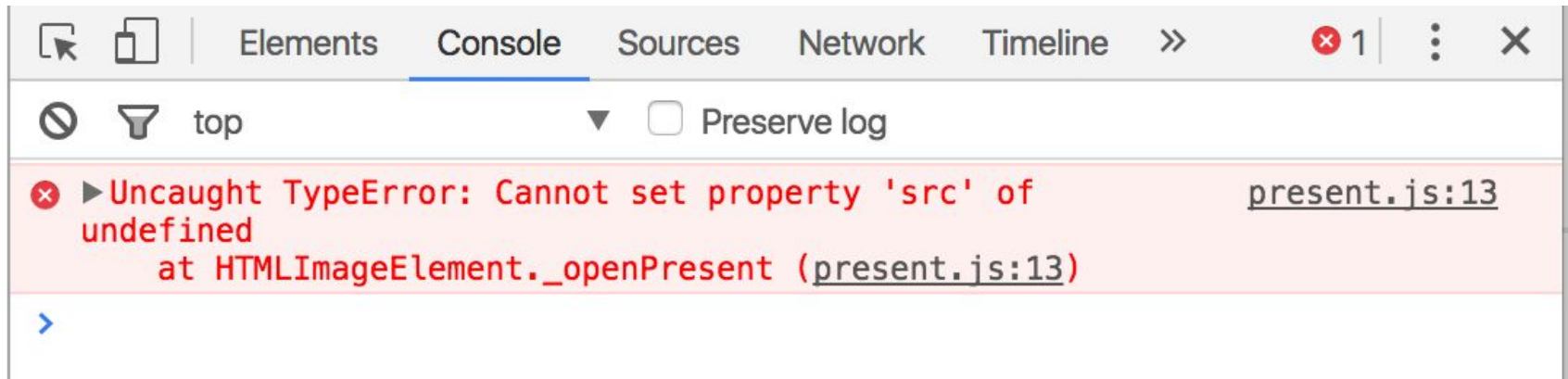
    // Create image and append to container.
    this.image = document.createElement('img');
    this.image.src = 'https://s3-us-west-2.amazonaws.com/s.cdpn.io/1083533/gift-icon.png';
    this.image.addEventListener('click', this._openPresent);
    this.containerElement.append(this.image);
  }

  _openPresent(event) {
    this.image.src = 'https://media.giphy.com/media/27ppQU0xe7KlG/giphy.gif';
    this.image.removeEventListener('click', this._openPresent);
  }
}
```

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](#)

What's going on?

# JavaScript `this`

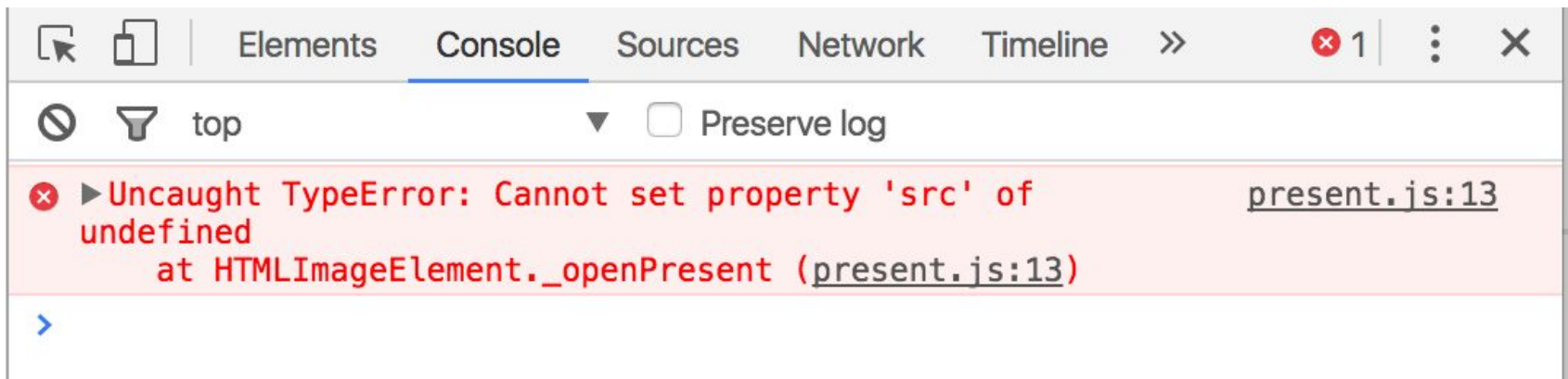
The `this` keyword in JavaScript is **dynamically assigned**, or in other words: `this` means different things in different contexts ([mdn list](#))

- 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/27ppQU0xe7KlG/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](#)):

```
_openPresent(event) {  
  this.image.src = 'https://media.giphy.com/media/27ppQU0xe7KlG/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](#)):

**Click a present to open it:**

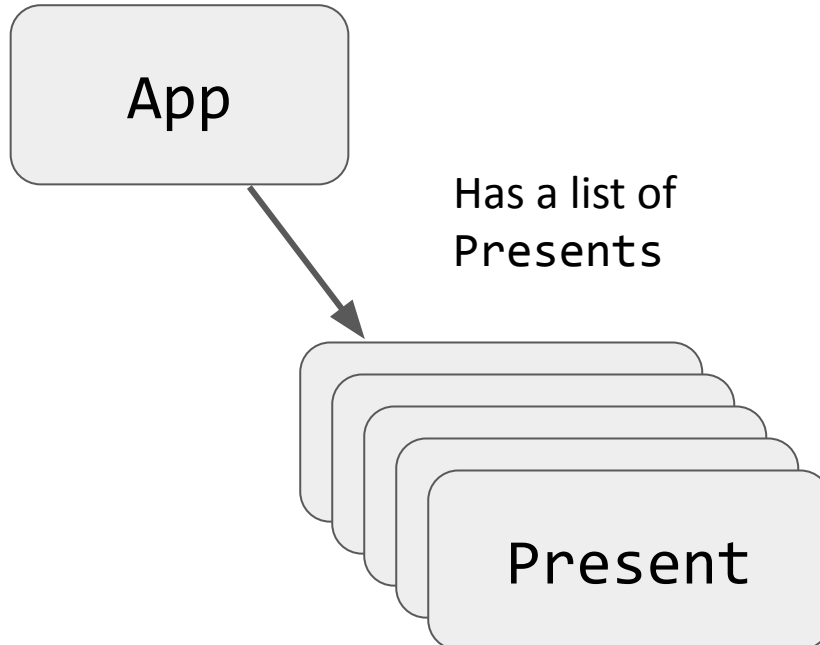




# Multiple classes

And we have implemented this with two classes:

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



[CodePen](#)

# Communicating btwn classes

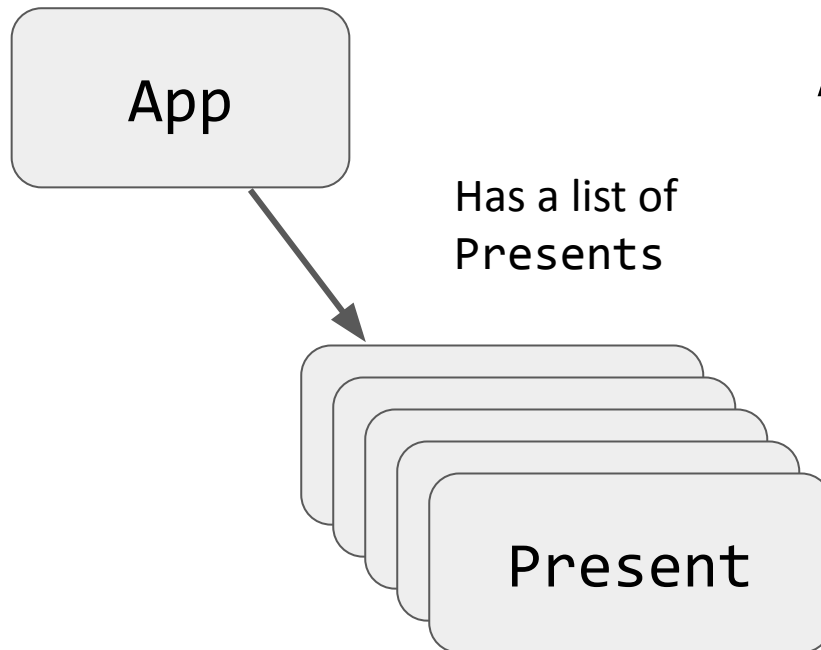
What if we want to change the **title** when all present have been opened? ([CodePen](#))

**Enjoy your presents!**



# Communication btwn classes

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

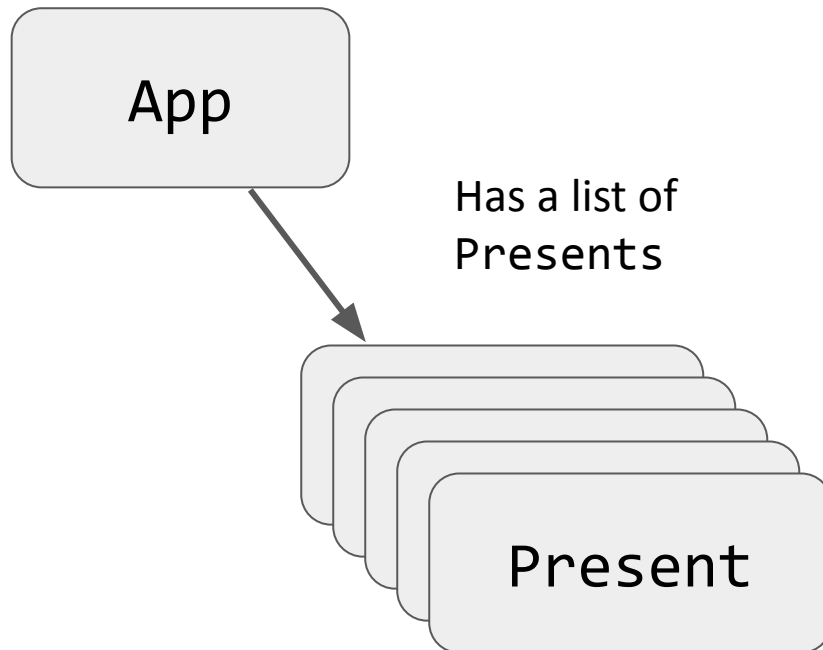


App can just call methods on Present:

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

# Communication btwn classes

However, communicating Present  $\rightarrow$  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

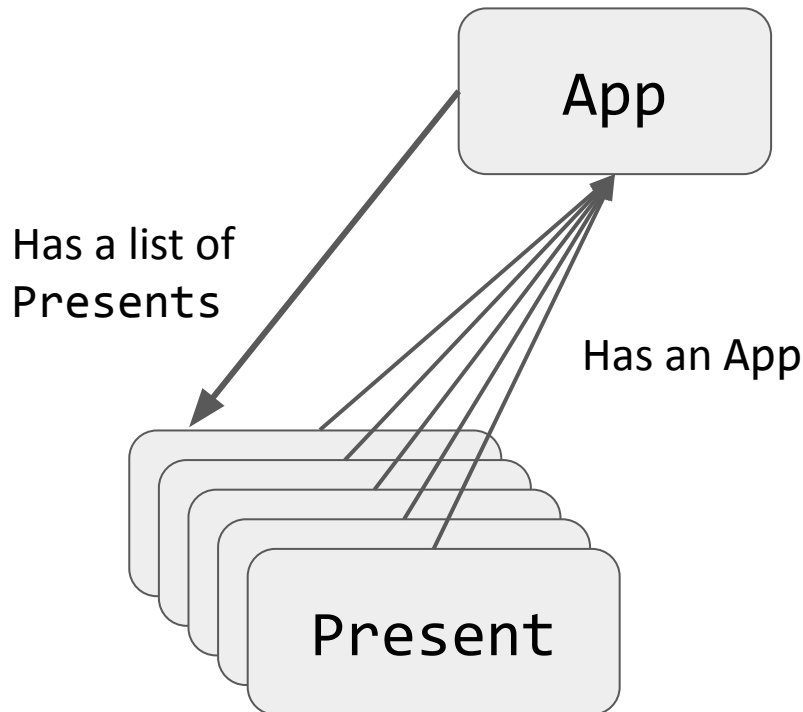
**OK (don't forget to bind)**

3. Add onOpened "callback function" to Present

**Best option (don't forget to bind)**

# Terrible style: Presents own App

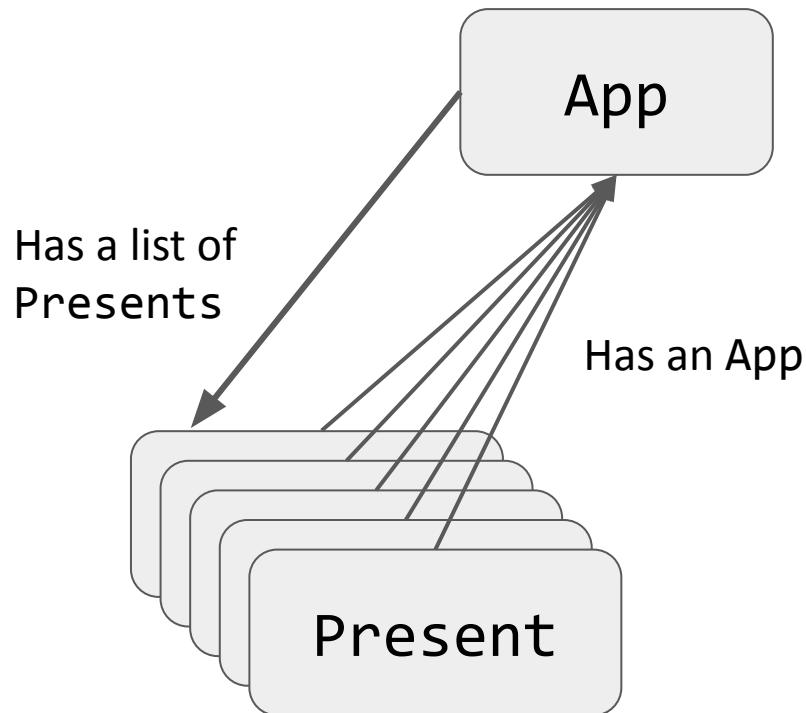
A naive fix is to just give Present a reference to App in its constructor: [CodePen](#)



**(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

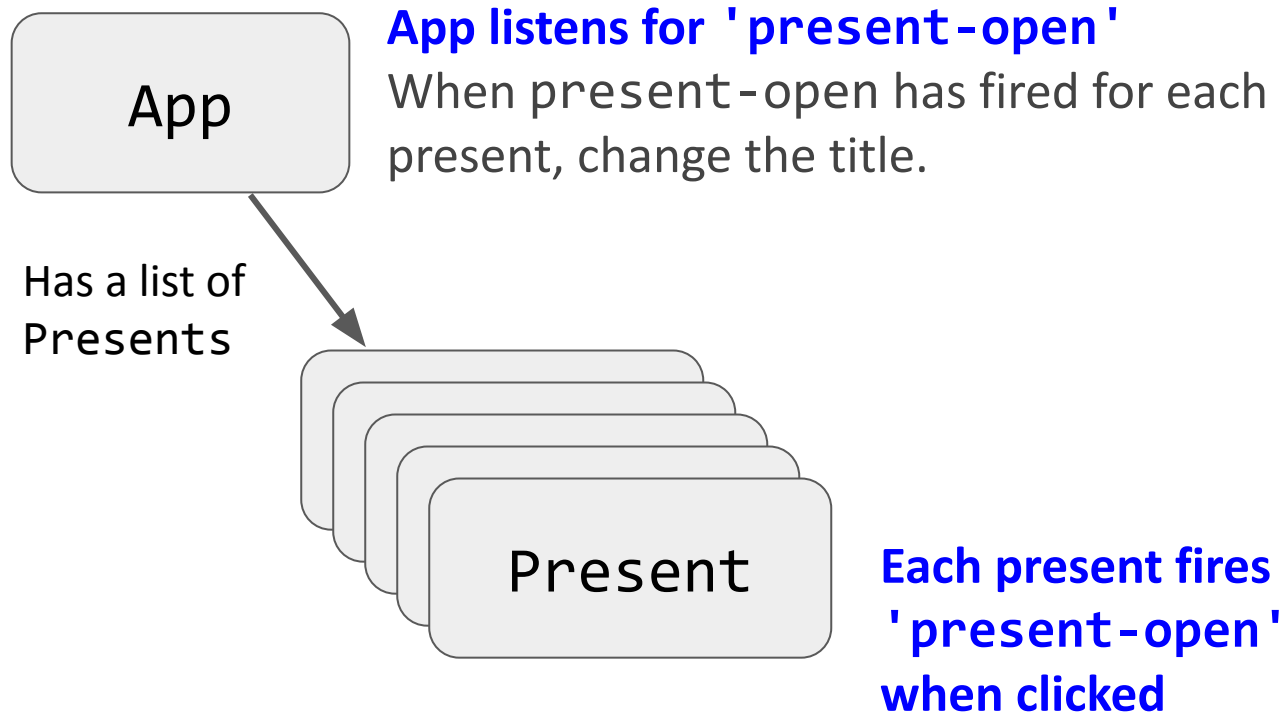
You can listen to and dispatch Custom Events to communicate between classes ([mdn](#)):

```
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...



[CodePen attempt](#)

# this in event handler

```
✖ ▶ Uncaught TypeError: Cannot read property 'length' of undefined      app.js:24  
    at HTMLDocument._onPresentOpened (app.js:24)  
    at Present._openPresent (present.js:19)
```

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`:

```
image.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 / identifier
  - 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](#)

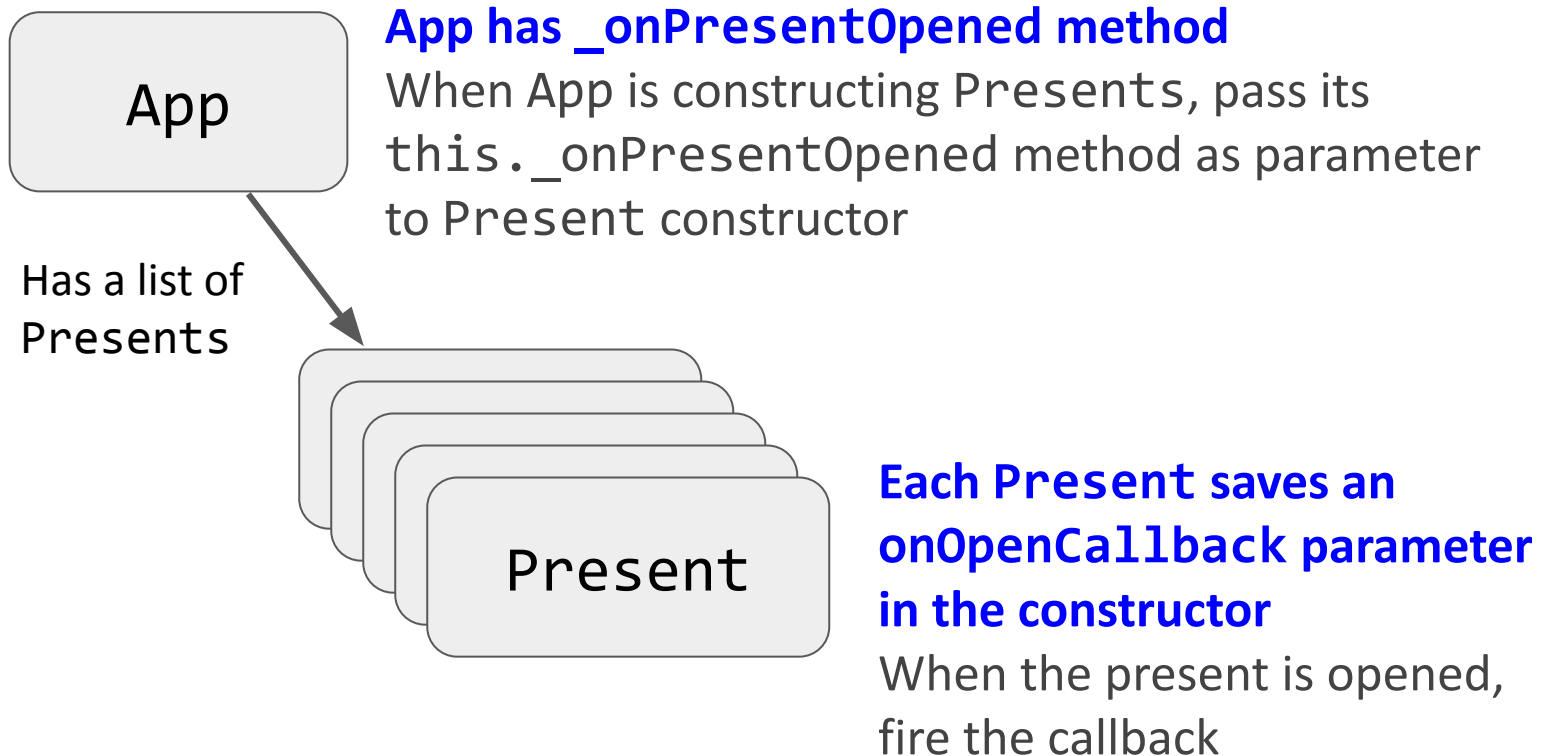
# A real example: Callbacks

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](#))



# this in event handler

```
✘ ▶ Uncaught TypeError: Cannot read property 'length' of undefined      app.js:21  
    at Present._onPresentOpened [as onOpenCallback] (app.js:21)  
    at Present._openPresent (present.js:20)
```

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);
```

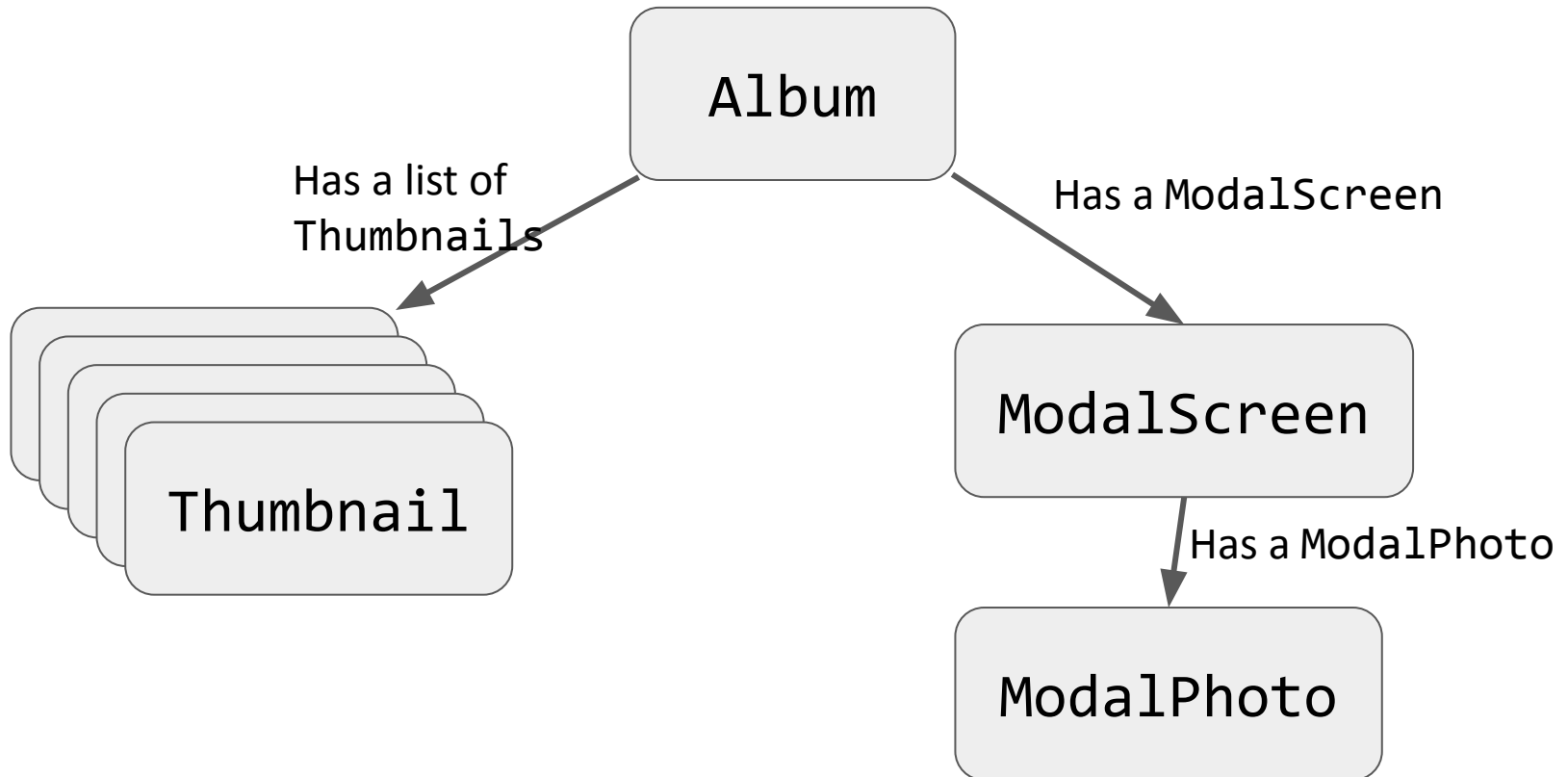
```
this._onPresentOpened = this._onPresentOpened.bind(this);
```

[CodePen solution](#)

# Object-oriented photo album

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

[CodePen](#)



More next time!