Schedule

Today:
- More on callbacks
- Functional JavaScript
  - Currying
  - Closures
  - Anonymous functions

Next week: Servers!
- Monday: Querying servers
- Wed/Fri: Writing servers
Prereq: Command line

Sometime next week, we will need to start using the command line.

We will not be teaching how to use a command line interface. This was a prerequisite for the class through CS1U.

Please make sure you know how to:
- Navigate between directories in a command line
- Open / edit files via command-line
- Execute scripts via command-line
Callbacks
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.
Recall: Button example

Menu:
- Has an array of Buttons
- Also updates the <h1> with what was clicked

Button:
- Notifies Menu when clicked, so that Menu can update the <h1>

Solution with Custom Events
Custom Events: Menu listens for a 'button-clicked' event
```
class Button {
    constructor(containerElement, text) {
        this.containerElement = containerElement;
        this.text = text;

        this.onClick = this.onClick.bind(this);

        const button = document.createElement('button');
        button.textContent = text;
        button.addEventListener('click', this.onClick);
        this.containerElement.appendChild(button);
    }

    onClick() {
        const eventInfo = {
            buttonName: this.text
        };
        document.dispatchEvent(
            new CustomEvent('button-clicked', { detail: eventInfo }));
    }
}
```

Custom Events: Button **dispatches** a 'button-clicked' event, with information on what was clicked
How would we implement the same thing with callbacks?
Callback solution

**Button:**
- Takes a *function parameter* (callback) in the constructor
- Saves this parameter as a field
- Invokes the saved callback function when clicked

**Menu:**
- Passes `showButtonClicked` method as parameter in Button constructor

C was clicked
class Button {
    constructor(containerElement, text) {
        this.containerElement = containerElement;
        this.text = text;

        this.onClick = this.onClick.bind(this);

        const button = document.createElement('button');
        button.textContent = text;
        button.addEventListener('click', this.onClick);
        this.containerElement.appendChild(button);
    }

    onClick() {
        console.log('clicked: ' + this.text);
    }
}

**Callback Sender Strategy:** Add an onClickedCallback function parameter to the Button constructor, save it in field, and invoke it onClick.
class Button {
    constructor(containerElement, text, onClickedCallback) {
        this.containerElement = containerElement;
        this.text = text;
        this.onClickedCallback = onClickedCallback;

        this.on_click = this.on_click.bind(this);

        const button = document.createElement('button');
        button.textContent = text;
        button.addEventListener('click', this.on_click);
        this.containerElement.appendChild(button);
    }

    onClick() {
        this.onClickedCallback(this.text);
    }
}
Button constructor takes an `onClickedCallback` function parameter, which is saves in a field of the same name.
class Button {

    constructor(containerElement, text, onClickedCallback) {
        this.containerElement = containerElement;
        this.text = text;
        this.onClickedCallback = onClickedCallback;

        this.onclick = this.onclick.bind(this);

        const button = document.createElement('button');
        button.textContent = text;
        button.addEventListener('click', this.onclick);
        this.containerElement.appendChild(button);
    }

    onClick() {
        this.onClickedCallback(this.text);
    }
}

Invoke the saved callback function when clicked.
class Button {
    constructor(containerElement, text, onClickedCallback) {
        this.containerElement = containerElement;
        this.text = text;
        this.onClickedCallback = onClickedCallback;

        this.onClick = this.onClick.bind(this);

        const button = document.createElement('button');
        button.textContent = text;
        button.addEventListener('click', this.onClick);
        this.containerElement.appendChild(button);
    }

    onClick() {
        this.onClickedCallback(this.text);
    }
}

You can send whatever parameter(s) you'd like in the callback function.
Callback Receiver Strategy: Add a method to be called when a button is clicked and pass it to the constructor of Button
Add the `showButtonClicked` method, which should be called when the button is clicked.
class Menu {
    constructor() {
        this.buttonContainer = document.querySelector('#menu');
        this.statusBar = document.querySelector('#status-bar');
        
        this.showButtonClicked = this.showButtonClicked.bind(this);
        
        this.buttons = [
            new Button(this.buttonContainer, 'A', this.showButtonClicked),
            new Button(this.buttonContainer, 'B', this.showButtonClicked),
            new Button(this.buttonContainer, 'C', this.showButtonClicked)
        ];
    }

    showButtonClicked(buttonName) {
        this.statusBar.textContent = buttonName + ' was clicked';
    }
}
class Menu {
    constructor() {
        this.buttonContainer = document.querySelector('#menu');
        this.statusBar = document.querySelector('#status-bar');
        this.showButtonClicked = this.showButtonClicked.bind(this);
        this.buttons = [
            new Button(this.buttonContainer, 'A', this.showButtonClicked),
            new Button(this.buttonContainer, 'B', this.showButtonClicked),
            new Button(this.buttonContainer, 'C', this.showButtonClicked)
        ];
    }
    showButtonClicked(buttonName) {
        this.statusBar.textContent = buttonName + ' was clicked';
    }
}

Note that we still have to bind showButtonClicked, even though it won't be invoked as a result of a DOM event.
class Menu {
    constructor() {
        this.buttonContainer = document.querySelector('#menu');
        this.statusBar = document.querySelector('#status-bar');

        this.showButtonClicked = this.showButtonClicked.bind(this);

        this.buttons = [
            new Button(this.buttonContainer, 'A', this.showButtonClicked),
            new Button(this.buttonContainer, 'B', this.showButtonClicked),
            new Button(this.buttonContainer, 'C', this.showButtonClicked)
        ];
    }
}

showButtonClicked(buttonName) {
    this.statusBar.textContent = buttonName + ' was clicked';
}

Pass the `showButtonClicked` method to the constructor of Button
Button example solution

Solution with Callbacks
Q: Why did we have to bind `showButtonClicked`?
this in a method
this in different contexts

this in a constructor:
- this is set to the new object being created
	his in a function firing in response to a DOM event:
- this is set to the DOM element to which the event handler was attached

this being called as a method on an object:
- this is set to the that is calling the method, or the object on which the method is called.

(all values of this)
When Button is constructed, `showButtonClicked` is being saved in Button's `onClickedCallback` field.
Button is the object that ultimately calls the `showButtonClicked` function.
Without the call to `bind`, `this` in `showButtonClicked` is `Button`, and this will result in a JS error when we try to refer to `this.statusBar.textContent` (CodePen)
class Menu {
  constructor() {
    this.buttonContainer = document.querySelector('#menu');
    this.statusBar = document.querySelector('#status-bar');

    this.showButtonClicked = this.showButtonClicked.bind(this);

    this.buttons = [
      new Button(this.buttonContainer, 'A', this.showButtonClicked),
      new Button(this.buttonContainer, 'B', this.showButtonClicked),
      new Button(this.buttonContainer, 'C', this.showButtonClicked)
    ];
  }

  showButtonClicked(buttonName) {
    this.statusBar.textContent = buttonName + ' was clicked';
  }
}

But with the call to bind, this in showButtonClicked is the Menu, which is the behavior we want. (CodePen)
One more look at bind
Objects in JS

Objects in JavaScript are sets of property-value pairs:

const bear = {
  name: 'Ice Bear',
  hobbies: ['knitting', 'cooking', 'dancing']
};
Classes in JS

```javascript
class Playlist {
    constructor(name) {
        this.playlistName = name;
        this.songs = [];
    }

    addSong(songName) {
        this.songs.push(songName);
    }
}

const playlist = new Playlist('More Life');
playlist.addSong('Passionfruit');
```

Classes in JavaScript produce **objects** through `new`. *(CodePen)*
Classes in JS

```javascript
class Playlist {
    constructor(name) {
        this.playlistName = name;
        this.songs = [];
    }

    addSong(songName) {
        this.songs.push(songName);
    }
}

const playlist = new Playlist('More Life');
playlist.addSong('Passionfruit');
```

Q: Are the objects created from classes also sets of property-value pairs?
Classes and objects

const playlist = new Playlist('More Life');

A: Yes. The playlist object created by the constructor essentially* looks like this:

```javascript
{
  playlistName: 'More Life',
  songs: [],
  addSong: function(songName) {
    this.songs.push(songName);
  }
}
```

Technically addSong (and the constructor function) is defined in the prototype of the playlist object, but we haven't talked about prototypes and probably won't talk about prototypes until the end of the quarter.
Classes and objects

```javascript
const playlist = new Playlist('More Life');

{  
    playlistName: 'More Life',  
    songs: [],  
    addSong: function(songName) {  
        this.songs.push(songName);  
    }
}
```

In JavaScript, a **method** of an object is just a **property** whose value is of Function type.
Classes and objects

const playlist = new Playlist('More Life');

{ 
  playlistName: 'More Life',
  songs: [],
  addSong: function(songName) {
    this.songs.push(songName);
  }
}

In JavaScript, a **method** of an object is just a **property** whose value is of Function type.

And just like any other Object property, the value of that method can be changed.
Rewriting a function

```javascript
class Playlist {
    constructor(name) {
        this.playlistName = name;
        this.songs = [];
    }

    addSong(songName) {
        this.songs.push(songName);
    }
}

const playlist = new Playlist('More Life');
playlist.addSong = function(songName) {
    console.log("Nah");
};

playlist.addSong('Passionfruit');
console.log(playlist);
```

Q: What is the output of this code?
Rewriting a function

class Playlist {
    constructor(name) {
        this.playlistName = name;
        this.songs = [];
    }

    addSong(songName) {
        this.songs.push(songName);
    }
}

const playlist = new Playlist('More Life');
playlist.addSong = function(songName) {
    console.log("Nah");
};

playlist.addSong('Passionfruit');
console.log(playlist);
When would you ever want to rewrite the definition of a method?!
**bind in classes**

```javascript
constructor() {
    const someValue = this;
    this.onClick = this.onClick.bind(someValue);
}
```

The code in purple is saying:
- Make a copy of onClick, which will be the exact same as onClick except this in onClick is always set to the someValue
The code in purple is **rewriting the `onClick` property** of the object:

- Assign the value of the `onClick` property: set it to the new function returned by the call to `bind`
Practical Functional JavaScript
We are going to cover some topics that are fundamental to a programming paradigm called functional programming.

Pure functional programming is pretty extreme:
- Everything in your code is either a function or an expression
- There are no statements
- There is no state:
  - No variables, fields, objects, etc

Comes from the idea of treating a computer program as a mathematical function
Functional programming

This is a code snippet from Scheme, a functional programming language:

```
(define (sum row)
  (let loop ((row row) (result '()))
    (if (= (length row) 1)
        (reverse result)
        (loop (cdr row)
              (cons (+ (first row) (second row))
                    result)))))
```

Everything is a function or the result of a function call.
Practical FP in JS

Most software is **not** built using a pure functional programming paradigm, so we won't be covering it.

But there are some ideas from functional programming that are immensely useful:

- First-class functions (functions as objects)
- **Currying**
- Closures
- **Anonymous functions** / lambdas / function literals
Why FP matters

Why should we learn about this other programming paradigm?

- There are **ideas you can express more clearly** and concisely with functional programming.

- There are **problems you can solve much more easily** with functional programming.

- *(very practically)* You will see JavaScript code in the wild that uses functional programing and the code will be indecipherable if you don't learn it.

- *(very practically)* Functional programming is trendy and so useful that C++ and Java added support for a few critical FP concepts (lambdas/closures) in the past few years.
First-class functions

Functions in JavaScript are objects.
- They can be saved in variables
- They can be passed as parameters
- They have properties, like other objects
- They can be defined without an identifier

(This is also called having first-class functions, i.e. functions in JavaScript are "first-class" because they are treated like any other variable/object.)
Recall: Functions as parameters

We know that we can pass functions as parameters to other functions. We've already done this multiple times:

- The event handler parameter to `addEventListener`
- As a parameter for a constructor of a new object

`Array` objects also have several methods that take functions as parameters.
Example: `findIndex`

`list.findIndex(callback, thisArg)`:

Returns the index of an element.

`callback` is a function with the following parameters:
- `element`: The current element being processed.
- `index`: The index of the current element being processed in the array.
- `array`: The array `findIndex` was called upon.

`callback` is called for every element in the array, and returns true if found, false otherwise.

`thisArg` is the value of this in `callback`
Remove with for-loop

```javascript
// Removes the first song in the playlist that
// matches |songName|, case insensitive.
removeSong(songName) {
    for (let i = 0; i < this.songs.length; i++) {
        const song = this.songs[i];
        if (song.toLowerCase() === songName.toLowerCase()) {
            this.songs.shift(i, 1);
            break;
        }
    }
}
```

Let's say that we added a `removeSong` method to Playlist (CodePen)
// Removes the first song in the playlist that matches `songName`, case insensitive.
removeSong(songName) { 
    for (let i = 0; i < this.songs.length; i++) {
        const song = this.songs[i];
        if (song.toLowerCase() === songName.toLowerCase()) {
            this.songs.shift(i, 1);
            break;
        }
    }
}

How would we rewrite this using `findIndex`?

Starter CodePen
General approach

```javascript
doesSongTitleMatch(element, index, array) {
    // ... 
    // return true if the song title matches 
    // false otherwise
}

removeSong(songName) {
    const index = this.songs.findIndex(doesSongTitleMatch);
    this.songs.shift(index, 1);
}
```

We want to do something like this...
General approach

```javascript
function doesSongTitleMatch(element, index, array) {
  // how do we get songName?
  return element === songName; // DOESN'T WORK
}

function removeSong(songName) {
  const index = this.songs.findIndex(doesSongTitleMatch);
  this.songs.shift(index, 1);
}
```

But the problem is that we want to pass `songName` into the `doesSongTitleMatch` function somehow.
But the problem is that we want to pass `songName` into the `doesSongTitleMatch` function somehow.
Clunky solution: field

```javascript
doesSongTitleMatch(element, index, array) {
    // This works but is really gross.
    return element === this.removeSongNameParameter;
}

removeSong(songName) {
    this.removeSongNameParameter = songName;
    const index = this.songs.findIndex(this.doesSongTitleMatch, this);
    this.songs.shift(index, 1);
}
```

We could save the song parameter as a field, which the doesSongTitleMatch method can access…

(CodePen)
Clunky solution: field

```javascript
function doesSongTitleMatch(element, index, array) {
  // This works but is really gross.
  return element === this.removeSongNameParameter;
}

function removeSong(songName) {
  this.removeSongNameParameter = songName;
  const index = this.songs.findIndex(this.doesSongTitleMatch, this);
  this.songs.shift(index, 1);
}
```

But then you have this weird removeSongNameParameter field that is only valid in between these method calls. (CodePen)
Add a parameter?

```javascript
doesSongTitleMatch(element, index, array) {
  // How can we get `songName` here?
  return element === songName; // DOESN'T WORK
}

removeSong(songName) {
  const index = this.songs.findIndex(this.doesSongTitleMatch);
  this.songs.shift(index, 1);
}
```

We really want to pass the `songName` value from `removeSong` to `doesSongTitleMatch`...
Add a parameter?

```javascript
doesSongTitleMatch(element, index, array) {
    // How can we get `songName` here?
    return element === songName; // DOESN'T WORK
}

removeSong(songName) {
    const index = this.songs.findIndex(this.doesSongTitleMatch);
    this.songs.shift(index, 1);
}
```

But the callback for `findIndex` expects 3 specific parameters, and we can't somehow add `songName`. 
One solution: new function

We can do this (CodePen):

```javascript
createMatchFunction(songName) {
    const findIndexFunction = function (element, index, array) {
        return element.toLowerCase() === songName.toLowerCase();
    }
    return findIndexFunction;
}

removeSong(songName) {
    const matchFunction = this.createMatchFunction(songName);
    const index = this.songs.findIndex(matchFunction);
    this.songs.shift(index, 1);
}
```
One solution: new function

We can do this ([CodePen](https://codepen.io)):  

```javascript
createMatchFunction(songName) {
    const findIndexFunction = function (element, index, array) {
        return element.toLowerCase() === songName.toLowerCase();
    }
    return findIndexFunction;
}

removeSong(songName) {
    const matchFunction = this.createMatchFunction(songName);
    const index = this.songs.findIndex(matchFunction);
    this.songs.shift(index, 1);
}
```
Creating functions within functions
Functions that create functions

In JavaScript, we can create functions from within functions (CodePen).

```javascript
function printMessage(birthYear) {
  function getLabel(age) {
    if (age < 2) {
      return "baby";
    }
    if (age < 4) {
      return "toddler";
    }
    if (age < 13) {
      return "kid";
    }
    if (age < 20) {
      return "teenager";
    }
    return "grown-up";
  }
  const ageThisYear = 2017 - birthYear;
  const label = getLabel(ageThisYear);
  console.log('You are a ' + label + ' this year.);
}
printMessage(2005);
```
Functions that create functions

In JavaScript, we can create functions from within functions (CodePen).

A function declared within a function is also known as a closure.
Scope of closures

Functions declared with `function` (or `var`) have function scope.
- Can be referenced anywhere in the function after declaration

```
function printMessage(birthYear) {
    if (true) {
        function getLabel(age) {
            if (age < 2) {
                return "baby";
            }
            if (age < 4) {
                return "toddler";
            }
            if (age < 13) {
                return "kid";
            }
            if (age < 20) {
                return "teenager";
            }
            return "grown-up";
        }
        const ageThisYear = 2017 - birthYear;
        const label = getLabel(ageThisYear);
        console.log('You are a ' + label + ' this year.');
    }
}
```

This example works:

<table>
<thead>
<tr>
<th>Console</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;You are a kid this year.&quot;</td>
</tr>
</tbody>
</table>
Scope of closures

Functions declared with `function` (or `var`) have function scope.
- Cannot be referenced outside the function

```javascript
function printMessage(birthYear) {
    function getLabel(age) {
        if (age < 2) {
            return "baby";
        }
        if (age < 4) {
            return "toddler";
        }
        if (age < 13) {
            return "kid";
        }
        if (age < 20) {
            return "teenager";
        }
        return "grown-up";
    }
    const ageThisYear = 2017 - birthYear;
    const label = getLabel(ageThisYear);
    console.log('You are a ' + label + ' this year.');</n    
printMessage(2005);
const label = getLabel(8);```

This example doesn't work:
Scope of closures

Functions declared with `function` (or `var`) have function scope.
- Cannot be referenced outside the function

This example doesn't work:

```javascript
function printMessage(birthYear) {
    function getLabel(age) {
        if (age < 2) {
            return "baby";
        }
        if (age < 4) {
            return "toddler";
        }
        if (age < 13) {
            return "kid";
        }
        if (age < 20) {
            return "teenager";
        }
        return "grown-up";
    }
    const ageThisYear = 20;
    const label = getLabel(ageThisYear);
    console.log('You are a kid this year.');
}
printMessage(2005);
console.log('You are a kid this year.');
const label = getLabel();
```

Uncaught ReferenceError: getLabel is not defined at 72165567caf5acb78997480f59e315c6:59
Scope of closures

Functions declared with `const` or `let` have block scope
- Cannot be referenced outside of the block.

This example doesn't work:

```javascript
function printMessage(birthYear) {
  if (true) {
    const getLabel = function(age) {
      if (age < 2) {
        return "baby";
      }
      if (age < 4) {
        return "toddler";
      }
      if (age < 13) {
        return "kid";
      }
      if (age < 20) {
        return "teenager";
      }
      return "grown-up";
    }
    const ageThisYear = 2017 - birthYear;
    const label = getLabel(ageThisYear);
    console.log('You are a ' + label + ' this year.);
  }
}
```
Functions that return functions

In JavaScript, we can **return** new functions as well. (We kind of knew this already because `bind` returns a new function.)

```javascript
function makeHelloFunction(name) {
    const greeting = function() {
        console.log('Hello, ' + name);
    };  
    return greeting;
}

const helloWorld = makeHelloFunction('world');
const hello3 = makeHelloFunction('hello, hello');

helloWorld();
hello3();

CodePen
```
Functions that create functions

```javascript
function makeHelloFunction(name) {
    const greeting = function() {
        console.log('Hello, ' + name);
    }
    return greeting;
}

const helloWorld = makeHelloFunction('world');
const hello3 = makeHelloFunction('hello, hello');

helloWorld();
hello3();
```

<table>
<thead>
<tr>
<th>Hello, world</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hello, hello, hello</td>
</tr>
</tbody>
</table>
Closure: an inner function

```javascript
function makeHelloFunction(name) {
    const greeting = function() {
        console.log('Hello, ' + name);
    };
    return greeting;
}
```

- When you declare a function inside another function, the inner function is called a closure.
Within a closure, you can reference variables that were declared in the outer function, and those variables will not go away after the outer function returns.
Functions that create functions

```javascript
function makeHelloFunction(name) {
  const greeting = function() {
    console.log('Hello, ' + name);
  }
  return greeting;
}

const helloWorld = makeHelloFunction('world');
const hello3 = makeHelloFunction('hello, hello');

helloWorld();
hello3();
```

The scope of `greeting` is only in the `makeHelloFunction` function, as well as the scope of `name`...
Functions that create functions

```javascript
function makeHelloFunction(name) {
  const greeting = function() {
    console.log('Hello, ' + name);
  };
  return greeting;
}

const helloWorld = makeHelloFunction('world');
const hello3 = makeHelloFunction('hello, hello');

helloWorld();
hello3();
```

But the `makeHelloFunction` function returns a reference to the function, which is an object, so the function object doesn't go away
Functions that create functions

```javascript
function makeHelloFunction(name) {
  const greeting = function() {
    console.log('Hello, ' + name);
  }
  return greeting;
}

const helloWorld = makeHelloFunction('world');
const hello3 = makeHelloFunction('hello, hello');

helloWorld();
hello3();
```

And the function object keeps a reference to the name parameter, so that when the created function is called...
Functions that create functions

```javascript
function makeHelloFunction(name) {
    const greeting = function() {
        console.log('Hello, ' + name);
    }
    return greeting;
}

const helloWorld = makeHelloFunction('world');
const hello3 = makeHelloFunction('Hello, world');

helloWorld();
hello3();
```

... we see that the new function returned from `makeHelloFunction` still has access to the `name` variable.
Functions that create functions

```javascript
function makeHelloFunction(name) {
    const greeting = function() {
        console.log('Hello, ' + name);
    }
    return greeting;
}

const helloWorld = makeHelloFunction('world');
const hello3 = makeHelloFunction('hello, hello');

helloWorld();
hello3();
```

The idea of constructing a new function that is "partially instantiated" with arguments is called **currying**. (article)
Anonymous functions

We do not need to give an identifier to functions. When we define a function without an identifier, we call it an **anonymous function**

- Also known as a **function literal**, or a **lambda function**

```javascript
function makeHelloFunction(name) {
  const greeting = function() {
    console.log('Hello, ' + name);
  };
  return greeting;
}
```
Anonymous functions

We do not need to give an identifier to functions. When we define a function without an identifier, we call it an **anonymous function**.

- Also known as a **function literal**, or a **lambda function**

```javascript
function makeHelloFunction(name) {
  return function() {
    console.log('Hello, ' + name);
  };
}
```

[CodePen](https://codepen.io/your_username/your_codepen_url)
Back to our Playlist
We want to do something like this...
But the problem is that we want to pass `songName` into the `doesSongTitleMatch` function somehow.
Instantiating a function...

doesSongTitleMatch(element, index, array) {
    // how do we get songName?
    return element === songName; // DOESN'T WORK
}

removeSong(songName) {
    const index = this.songs.findIndex(doesSongTitleMatch);
    this.songs.shift(index, 1);
}

We want to create a version of doesSongTitleMatch, with a value assigned to songName.
Currying

We can do this (CodePen):

createMatchFunction(songName) {
  const findIndexFunction = function (element, index, array) {
    return element.toLowerCase() === songName.toLowerCase();
  }
  return findIndexFunction;
}

removeSong(songName) {
  const matchFunction = this.createMatchFunction(songName);
  const index = this.songs.findIndex(matchFunction);
  this.songs.shift(index, 1);
}
Currying

We've created a function whose signature matches what `findIndex` expects.

createMatchFunction(songName) {
  const findIndexFunction = function (element, index, array) {
    return element.toLowerCase() === songName.toLowerCase();
  }
  return findIndexFunction;
}

removeSong(songName) {
  const matchFunction = this.createMatchFunction(songName);
  const index = this.songs.findIndex(matchFunction);
  this.songs.shift(index, 1);
}
Currying

We're creating this function within an outer function that takes the `songName`.

```javascript
createMatchFunction(songName) {
  const findIndexFunction = function (element, index, array) {
    return element.toLowerCase() === songName.toLowerCase();
  }
  return findIndexFunction;
}
```

```javascript
removeSong(songName) {
  const matchFunction = this.createMatchFunction(songName);
  const index = this.songs.findIndex(matchFunction);
  this.songs.shift(index, 1);
}
```
Currying

This allows us to essentially construct a new `findIndexFunction`, with a set `songName` value. This is called **currying**.

```javascript
createMatchFunction(songName) {
    const findIndexFunction = function (element, index, array) {
        return element.toLowerCase() === songName.toLowerCase();
    }
    return findIndexFunction;
}

removeSong(songName) {
    const matchFunction = this.createMatchFunction(songName);
    const index = this.songs.findIndex(matchFunction);
    this.songs.shift(index, 1);
}
```
Cleaning up removeSong

We can also define the `findIndexFunction` directly in `removeSong`, instead of making a separate function to create one with the right parameters ([CodePen](https://codepen.io/)): 

```javascript
removeSong(songName) {
    const findIndexFunction = function (element, index, array) {
        return element.toLowerCase() === songName.toLowerCase();
    }
    const index = this.songs.findIndex(findIndexFunction);
    this.songs.shift(index, 1);
}
```
Cleaning up removeSong

We don't need to include the parameters we aren't using:

```javascript
removeSong(songName) {
    const findIndexFunction = function (element) {
        return element.toLowerCase() === songName.toLowerCase();
    }
    const index = this.songs.findIndex(findIndexFunction);
    this.songs.shift(index, 1);
}
```
Cleaning up removeSong

We can define the function directly in the `findIndex` parameter instead of saving it in a variable:

```javascript
removeSong(songName) {
    const index = this.songs.findIndex((element) => {
        return element.toLowerCase() === songName.toLowerCase();
    });
    this.songs.shift(index, 1);
}
```
Cleaning up `removeSong`

We can use the **arrow function** syntax for defining functions:

```javascript
removeSong(songName) {
  const index = this.songs.findIndex((element) => {
    return element.toLowerCase() === songName.toLowerCase();
  });
  this.songs.shift(index, 1);
}
```
Cleaning up `removeSong`

We can use the **concise version** of the **arrow function**:

- You can omit the parentheses if there is only one parameter
- You can omit the curly braces if there's only one statement in the function, and it's a return statement

```javascript
removeSong(songName) {
  const index = this.songs.findIndex(
    element => element.toLowerCase() === songName.toLowerCase()
  );
  this.songs.shift(index, 1);
}
```
removeSong before/after

```javascript
removeSong(songName) {
    for (let i = 0; i < this.songs.length; i++) {
        const song = this.songs[i];
        if (song.toLowerCase() === songName.toLowerCase()) {
            this.songs.shift(i, 1);
            break;
        }
    }
}
```

```javascript
removeSong(songName) {
    const index = this.songs.findIndex(
        element => element.toLowerCase() === songName.toLowerCase());
    this.songs.shift(index, 1);
}
```
# More Array functions

<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>list.forEach(function)</code></td>
<td>Executes the provided function once for each array element. (mdn)</td>
</tr>
<tr>
<td><code>list.filter(function)</code></td>
<td>Creates a new array with all elements that pass the test implemented by the provided function. (mdn)</td>
</tr>
<tr>
<td><code>list.every(function)</code></td>
<td>Tests whether all elements in the array pass the test implemented by the provided function. (mdn)</td>
</tr>
</tbody>
</table>

[All Array functions](#)
Gotchas and style notes
Recall: Present example

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

        this._openPresent = this._openPresent.bind(this);

        const image = document.createElement('img');
        image.src = OUTSIDE_IMAGE_URL;
        image.addEventListener('click', this._openPresent);
        this.containerElement.appendChild(image);
    }

    _openPresent(event) {
        const image = event.currentTarget;
        image.src = this.giftSrc;
    }
}
```

We implemented a Present class that had a separate `_openPresent` method.
What would happen if we defined the click event handler directly in the call to `addEventListener`? (CodePen)
class Present {
    constructor(containerElement, giftSrc) {
        this.containerElement = containerElement;
        this.giftSrc = giftSrc;

        const image = document.createElement('img');
        image.src = OUTSIDE_IMAGE_URL;
        image.addEventListener('click', function(event) {
            const image = event.currentTarget;
            image.src = this.giftSrc;
        });
        this.containerElement.appendChild(image);
    }
}
```javascript
class Present {
    constructor(containerElement, giftSrc) {
        this.containerElement = containerElement;
        this.giftSrc = giftSrc;

        const image = document.createElement('img');
        image.src = OUTSIDE_IMAGE_URL;
        image.addEventListener('click', (function(event) {
            const image = event.currentTarget;
            image.src = this.giftSrc;
        }).bind(this));
        this.containerElement.appendChild(image);
    }
}

Fixed CodePen
```
```javascript
class Present {
    constructor(containerElement, giftSrc) {
        this.containerElement = containerElement;
        this.giftSrc = giftSrc;

        const image = document.createElement('img');
        image.src = OUTSIDE_IMAGE_URL;
        image.addEventListener('click', (function(event) {
            const image = event.currentTarget;
            image.src = this.giftSrc;
        })[bind(this)]);
        this.containerElement.appendChild(image);
    }
}
```

[Fixed CodePen](https://codepen.io/username/pen/yourCodePenId)
class Present {
  constructor(containerElement, giftSrc) {
    this.containerElement = containerElement;
    this.giftSrc = giftSrc;

    const image = document.createElement('img');
    image.src = OUTSIDE_IMAGE_URL;
    image.addEventListener('click', event => {
      const image = event.currentTarget;
      image.src = this.giftSrc;
    });
    this.containerElement.appendChild(image);
  }
}

What would happen if we defined the click event handler like this, with the arrow function instead (CodePen)?
This works! Why?! 
(CodePen)

```javascript
image.addEventListener('click', event => {
    const image = event.currentTarget;
    image.src = this.giftSrc;
});
```
=> versus function

When you define a function using `function syntax`:

```javascript
const onClick = function() {
    const image = event.currentTarget;
    image.src = this.giftSrc;
};
```

This is will be dynamically assigned to a different value depending on how the function is called, like we've seen before (unless explicitly bound with `bind`)
=> versus function

When you define a function using arrow syntax:

```javascript
const onClick = event => {
    const image = event.currentTarget;
    image.src = this.giftSrc;
};
```

*this* is bound to the value of this in its enclosing context
Since we've used the arrow function in the constructor, the `this` in the enclosing context is the new Present object.
Which is better style?
class Present {
    constructor(containerElement, giftSrc) {
        this.containerElement = containerElement;
        this.giftSrc = giftSrc;

        this._openPresent = this._openPresent.bind(this);

        const image = document.createElement('img');
        image.src = OUTSIDE_IMAGE_URL;
        image.addEventListener('click', this._openPresent);
        this.containerElement.appendChild(image);
    }

    _openPresent(event) {
        const image = event.currentTarget;
        image.src = this.giftSrc;
    }
}

(A) Explicit event handler
class Present {
    constructor(containerElement, giftSrc) {
        this.containerElement = containerElement;
        this.giftSrc = giftSrc;

        const image = document.createElement('img');
        image.src = OUTSIDE_IMAGE_URL;
        image.addEventListener('click', event => {
            const image = event.currentTarget;
            image.src = this.giftSrc;
        });
        this.containerElement.appendChild(image);
    }
}
Callback style

Version A: Explicit event handler

- Pros:
  - Easier to read
  - More modular
  - Scales better to long functions, several event handlers

- Cons:
  - Because all class methods are public, it exposes the onClick function (which should be private)
Callback style

Version A: Explicit event handler

- Pros:
  - Easier to read
  - More modular
  - Scales better to long functions, several event handlers

- Cons:
  - Because all class methods are public, it exposes the onClick function (which should be private)
  - Need to bind explicitly
Callback style

Version A: Explicit event handler

- Pros:
  - Easier to read
  - More modular
  - Scales better to long functions, several event handlers

- Cons:
  - Because all class methods are public, it exposes the onClick function (which should be private)
  - Need to bind explicitly

This is the style I recommend and the preferred style for CS193X
Callback style

```
image.addEventListener('click', event => {
    const image = event.currentTarget;
    image.src = this.gifSrc;
});
```

Version B: Inline event handler

- Pros:
  - Does not expose the event handler: function is privately encapsulated
- Cons:
  - Constructor logic has unrelated logic inside of it
  - Will get messy with lots of event handlers, long event handlers
Callback style

Version B: Inline event handler

- Pros:
  - Does not expose the event handler: function is privately encapsulated
- Cons:
  - Constructor logic has unrelated logic inside of it
  - Will get messy with lots of event handlers

Some people strongly prefer this style because of the encapsulation aspect (but I don't recommend it).
Advanced closures

```javascript
function createFunction() {
    let x = 0;

    function inner() {
        x++;
        let y = 0;
        y++;

        console.log('x is: ' + x + ', ' + 'y is: ' + y);
    }
    return inner;
}

const functionOne = createFunction();
functionOne();
functionOne();
functionOne();
functionOne();
```

What's the output of this program? (CodePen)
Advanced closures

```javascript
function createFunction() {
  let x = 0;

  function inner() {
    x++;
    let y = 0;
    y++;

    console.log('x is: ' + x + ', ' + 'y is: ' + y);
  }

  return inner;
}

const functionOne = createFunction();
functionOne();
functionOne();
functionOne();
functionOne();
```
Closures

Within a closure, you can reference variables that were declared in the outer function, and those variables will not go away after the outer function returns.

```javascript
function createFunction() {
  let x = 0;

  function inner() {
    x++;
    let y = 0;
    y++;
    console.log('x is: ' + x + ', ' + 'y is: ' + y);
  }
  return inner;
}
```
Closures

```javascript
function createFunction() {
    let x = 0;

    function inner() {
        x++;
        let y = 0;
        y++;
        console.log('x is: ' + x + ', ' + 'y is: ' + y);
    }
    return inner;
}
```

The variable is not copied to the inner function; the inner function has a **reference** to the variable in the outer scope.

- **See this iconic StackOverflow post** to learn more
Closures

```javascript
function createFunction() {
  let x = 0;

  function inner() {
    x++;
    let y = 0;
    y++;

    console.log('x is: ' + x + ', ' + 'y is: ' + y);
  }

  return inner;
}
```

tl;dr: Be careful with closures! For now, we are not going to be modifying outer function variables in the closure.
Review: ES6 classes

- ES6 classes mostly work the way you expect

- **this in a constructor**: refers to the new object being created

- **this outside a constructor**: refers to a different value depending on how the function is called
  - In response to a DOM event, this is the element that the event handler was tied to
  - When called in a method, this is the object that the method is called from

- **bind**: sets the value of this for a function so it does not change depending on the context
Review: Functional JavaScript

- Functions in JavaScript are first-class citizens:
  - Objects that can be passed as parameters
  - Can be created within functions:
    - Inner functions are called closures
  - Can be created without being saved to a variable
    - These are called anonymous functions, or function literals, or lambdas
  - Can be created and returned from functions
    - Constructing a new function that references part of the outer function's parameters is called currying