Additional Javascript

- This is not a Javascript course
    - We are using it to learn more about web dev
- We learn a lot, but still a minimal intro
- These slides cover a handful of bonus topics
    - Not used in class, but useful for interviews

Prototypes

- JS is *NOT* a "classically object oriented" language
  - But it IS "object oriented"
- Objects yes, Classes no

Outside of JS:

- A Class is a blueprint defining an object

Within JS:

- Classes are a starting point for an object
- A **prototype**

Inheritance

- Objects can "inherit" properties
    - JS does it slightly differently
    - Uses the properties of another object

When code accesses a non-existent object property

- Object uses that property from its **prototype**
- Prototype is also an object
    - If it lacks the property, uses *its* prototype
- Continues until property found or no prototype
    - Similar to lexical scoping
    - But not based on scope

Prototype is a concept not a property

This involves some common confusion

- A prototype is an object
- A prototype can be accessed
- A prototype is NOT a property named `prototype`

An Object has "properties"

- Not a property named `properties`
- At least not because of the concept

Using a prototype

Inheritance from a prototype is automatic

Many built-in functions are accessed this way

```
const name = "amit"; // Strings are Objects!
name.newProperty = "someVal";

// These all work
console.log(name.newProperty); // not inherited
console.log(name.toUpperCase()); // inherited
console.log(name.length); // inherited
```

Accessing the Prototype (if you need to)

3 main ways:

- `yourObject.__proto__` (DON'T DO THIS)
    - Legacy code
- `Object.getPrototypeOf(yourObject)`
    - returns the prototype object
- Modify origin of prototype (SPECIAL CASES ONLY)
    - To **polyfill**
        - Add features that aren't yet in JS Engine
    - To **monkey patch**
        - Inject wrapper around existing function

Prototype Summary

- Prototypes are *objects*, not plans
  - Prototype can change after object created
- An Object's prototype not `.prototype` property
  - My emphasis will soon make sense

This is the most confusing topic

`this` is the hardest part of JS

- Deceptively similar to other languages
- Differences can surprise when it seemed the same
- English struggles to talk about `this`

Because of this (*see?!*), often an interview question

Essential Truth

`this` is a special variable name

- a new value **each time** you **enter a function**
    - Except for arrow functions (more soon)

The object the `this` variable refers to is the **context**

- Hopefully a useful object

DO NOT ASSUME `this` will be what you want

- You have to make it happen
- You must understand why it gets which value

Implicit Binding

- `this` **implicitly bound** when you enter a function
- Uses value **before the dot** in the function call

```
const cat = {
  sound: 'meow',
  speak: function() {
    console.log( cat.sound );
  },
  implicit: function() {
    console.log( this.sound );
  }
};

cat.speak();  // 'meow', `cat` is `cat`
cat.implicit(); // 'meow' `this` is `cat
```

`cat.implicit()` had `cat` **before the dot**

- `this` was **implicitly bound** to `cat`

When it works

Implicit binding works with copies and inheritance:

```
const cat = {
  sound: 'meow',
  speak: function() {
    console.log( this.sound );
  }
};
const feline = { sound: 'purr' };
feline.speak = cat.speak; // copy assignment, not calling

cat.speak(); // meow
feline.speak(); // meow or purr?  Why?
```

Implicit Binding has limitations

Imagine a function using `this` called as a callback

```
function usesCallback( callback ) {
  callback();
}

usesCallback( cat.speak ); // passing, not calling
```

`callback()` has no dot! What is value of `this`?

Result is different with/without `'use strict';`

- Without `use strict`: `this` is global object
- With `use strict`: `this` is `undefined`

Callbacks used very frequently!

```
const internetCats = {
  cats: [ 'Jorts', 'Jean' ],
  coolSite: 'rogue illegal tiktok',
  reportSites: function() {
    const html = this.cats.map( function(name) {
      return `<li>${name} uses ${this.coolSite}</li>`;
    }).join('');
    return html;
  },
};

console.log( internetCats.reportSites() );
```

Explicit Binding

Functions can be **explicitly bound**

- Forces `this` to given value

Implicit binding works fine unless:

- Your function is used as a callback
- *and* your function uses `this`
  - Much code doesn't use `this`
  - But some code will
  - Devs need to know when to worry about `this`
    - English worked out this time

Explicit Binding via .bind()

`.bind()` is a method on the prototype of all functions

- Returns a new function
- A bound version of the function itself
    - Original function itself **not bound**

```
usesCallback( cat.speak.bind(cat) );
```

Inside `usesCallback()`

- `callback` will be the explicitly bound function
- Inside `callback()` call
    - `this` will be `cat`
- Regardless of dot/no dot when `callback()` called

Avoiding Explicit Binding via Fat Arrow

Entering an arrow functions does not redefine `this`

Not explicit binding, it can avoid the need

```javascript
const internet = {
  cats: [ 'Jorts', 'Jean' ],
  coolSite: 'rogue illegal tiktok',
  report: function() {  // Need to keep as function keyword!
    const html = this.cats.map( name => {  // arrow here
      return `<li>${name} uses ${this.coolSite}</li>`;
    }).join('');
    return html;
  },
};

console.log( internet.report() );
```

Why did `report` need to use the `function` keyword?

Avoiding `this` old-school (DON'T DO)

In ancient times devs would bypass the `this` problem

- By copying the value of `this` into another variable

Usually called `self` or `that`

- Before defining an inline function as a callback

Entering the new function

- `this` would be redefined
- `self` would keep previous value of `this`

DON'T DO THIS (or `that`)

- Now unnecessary and visually noisy

Demonstration of old school way

```
const internetCats = {
  cats: [ 'Jorts', 'Jean' ],
  coolSite: 'MySpace', // I said this was old!
  reportSites: function() {
    const self = this; // self unchanged below
    const html = this.cats.map( function(name) {
      return `<li>${name} uses ${self.coolSite}</li>`;
    }).join('');
    return html;
  },
};

console.log( internetCats.reportSites() );
```

Additional notes

More ways to setting the context (`this`) for a call exist

- Such as `.call()` or `.apply()` (See MDN if curious)
    - Allow you to call a passed function
    - And say what value to use as context (`this`)
- These come up fairly rarely

`this` Summary

- `this` is a special variable name
    - Gets redefined when entering a function call
        - Except when entering arrow functions
- `this` **implicitly bound** to "what is before the dot"
- Implicit `this` a problem when function is callback
- **Explicit binding** is possible via `.bind()`
- Arrow functions do not redefine `this`
- Non-OOP programming can avoid `this` entirely
- Don't use work-arounds like `that` or `self`

Why Inheritance

Don't overuse Inheritance

- Modern best practices, including OOP:
    - Favor **Composition** over **Inheritance**

Inheritance can provide common functionality

Inheritance a problem when many instances

- But then need to change *half* of them
- We change code more than we write new

How to create Inheritance

JS has 4 ways to create inheritance

Really 4 ways to create a **prototype**

- Constructor Function
- Object.create
- ES6 classes
- Brute Force Prototype Assignment

Constructor Function - Older code, but checks out

Using `new` keyword on a function call:

- Creates a new object
- Calls the function with `this` set to the new object
- Sets the prototype of the returned object
  - To the `prototype` property of the function
  - function `.prototype` prototype of new object
  - But prototype isn't `.prototype` of new object

Such functions are MixedCase, not camelCase

- By convention, not code-enforced

Constructor Function Demo

```
const Cat = function(name) {  // MixedCase function name
  this.name = name;           // `this` is the new object
};

// Cat.prototype is just an empty object by default
// Is not the prototype of Cat
Cat.prototype.beNice = function() {
  console.log(`${this.name} silently maintains eye contact`);
};

const jorts = new Cat('Jorts');
jorts.beNice();
```

- `jorts.beNice` IS inherited
- `jorts.name` is NOT inherited
    - Is property of the object itself

Object.create - for the Functional Programmers

`Object.create()` gives you a new object

- New object's prototype set to passed object
- No initialization code runs (no constructor)
- Popular among functional programmers (FP)

```
const cat = {
  beNice: function() {
    console.log(`${this.name} maintains eye contact`);
  }
};
const jorts = Object.create(cat);
jorts.name = 'Jorts';
jorts.beNice();
```

ES6 Classes

- Use `new` on a **class** call
- Was hotly debated, now reaction is meh
- More comfortable for those from other languages
- Can mislead, defines starting state, not limits

```
class Cat {
  constructor(name) {
    this.name = name;
  }
  beNice() {
    console.log(`${this.name} pretends not to hear`);
  }
}

const jorts = new Cat('Jorts');
jorts.beNice();
```

Brute Force - Set the prototype directly

Usually a bad idea (messy/unclear)

- Listed for educational purposes!
- Use any of the other methods instead

```
const cat = {
  beNice: function() {
    console.log(`${this.name} says 'No'`);
  }
};
const jorts = { name: 'Jorts' };
Object.setPrototypeOf(jorts, cat);
jorts.beNice();
```

Hoisting

"Hoisting"

- `function` keyword functions (not as a value)
    - JS treats as if **defined** at top of function
    - Allows you to use and refer to functions
- `var` variables
    - JS treats as if **declared** at top of function
        - Declared, but not **assigned**
- In global scope, creates global variables
    - Only a browser issue, not backend