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 "classical object oriented" language

But it IS "object oriented"

Objects yes, Classes no.

Classes are a blueprint to describe what an object can do.

In JS, Objects are not nearly so restricted.

Inheritance

Objects can have "inheritance" - where an object can use the properties/methods of another object.

If the code tries to access a value on the object, and the object doesn't have it defined for itself, it will check to see if its **prototype** has it.

Because the prototype is an object, when asked for this value, if it doesn't have it, it will check to see if **its** prototype has it.

This continues until an object doesn't have a prototype.

Prototype is a concept

Note that we are discussing a concept.

A prototype is an object.

A prototype can be accessed.

A prototype is NOT a property named `prototype`

Just like how an Object has properties, but not a property named `properties`

Using a prototype

Inheritance from a prototype is automatic when you try to use the property.

Many built-in functions are accessed this way.

```
const name = "amit";
name.newProperty = "someVal";
console.log(name.newProperty); // not inherited
console.log(name.toUpperCase()); // inherited
console.log(name.length); // inherited
```

Accessing

If you need to access the prototype object itself, there are three main ways:

- use `yourObject.__proto__` (DON'T DO THIS) - Legacy code
- use `Object.getPrototypeOf(yourObject)` - returns the prototype object
- Modify the source of the prototype - more on this later, such as with polyfills

Prototype Summary

- Prototypes are *objects*, not plans
    - This means the prototype can be modified after the fact, like any object
- The prototype of an object is not the `.prototype` property of that object

This is the most confusing topic

`this` is the hardest part of JS

- Similar, but different than other languages
- Makes English hard to use to talk about "this"

Essential Truth

`this` is a special variable name

- **refers to a new value each time you enter a function**

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

- usually a relevant object, but it can get confused.

DO NOT ASSUME `this` will be what you want

- you have to make it happen

Implicit Binding

By default, `this` is **bound** to a value "implicitly" when you enter a function

Uses the 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.implicit(); // 'meow'
```

When it works

Implicit binding works through copies and inheritance just fine:

```
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(); // What do you expect? Why?
```

When it doesn't work

BUT implicit binding has problems.

99% of the time this is when the function with `this` is used as a callback.

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

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

When the function is called, what is before the dot?

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

And Callbacks happen all the time!

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

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

Explicit Binding

When your function is used as a callback

- you can **explicitly bind** that function
- to the value of `this` that you want

If your function doesn't use `this`, you don't care

- this is becoming increasingly common

Explicit Binding via .bind()

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

- it returns a new function
- returns the function it is called on, but bound

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

inside `usesCallback()`

- `callback` will be the explicitly bound function
- so `this` will be `cat`
- even though no dot when `callback()` called

Explicit Bind via Fat Arrow

Unlike other functions, Fat Arrow functions do not redefine `this`

Technically this is not explicit binding, so much as not re-binding at all

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

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

Avoiding `this` old-school

In ancient times devs would bypass `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 the previous value of `this`

DON'T DO THIS. It's unnecessary and visually noisy.

Demonstration of old school way

```
const internet = {
  cats: [ 'Jorts', 'Jean' ],
  coolSite: 'tiktok',
  report: 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( internet.report() );
```

Additional notes

Other methods of setting the context (`this`) for a function exist

- such as `.call()` or `.apply()`
- these come up fairly rarely

`this` Summary

- `this` is a variable name that gets redefined when entering a function call
- Implicit binding is to "what is before the dot"
    - `this` can be a problem if the function is used as a callback
- Explicit binding is possible via `.bind()`
- A fat-arrow function can avoid the redefinition
- If you use a non-OOP programming style you can avoid `this` entirely
- Don't use work-arounds like `that` or `self`

Why Inheritance

Don't overuse Inheritance

- Modern best practices, even for OOP, favor **Composition** over **Inheritance**

Inheritance can provide common functionality

Inheritance can be a problem if you have 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 style, still works

Using `new` keyword on a function call:

- creates a new object
- calls the function with `this` set to the new object
- runs the function
- sets the prototype of the returned object to be the `prototype` property of the function
    - `.prototype` object, not prototype of the function itself

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.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, only defines starting state

```
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

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

Hoisting

"Hoisting"

- `var` variables
- `function` keyword functions not as a value
- JS engine treats as declared at top of function
    - not assigned, but declared
    - allows you to make references

Create global variables when run in the global scope

- not inside a function
- only a browser issue
- hoisted values create global variables