JavaScript Classes

Relatively new (2015)

How we used to write code

```
function Dog() {
  this.bark = function () {
    return 'bark'
  }
  this.eat = function () {
    return 'mmmmmm'
  }
}
```

```
const riley = new Dog()
const roxy = new Dog()
const rover = new Dog()

riley.bark()
roxy.eat()
rover.bark()
```

JavaScript is an Object Prototypical language

- Works on prototypes (the attributes of an object)

Along came classes

- Allows for a more common syntax
- Familiar to Object Oriented developers from other languages (C++, C#, Java, Python, Ruby, etc)

Redefining Dogs

Let's redefine our Dog

```
class Dog {
  bark() {
    return 'bark'
  }

  eat() {
    return 'mmmm'
  }
}
```

Public field declarations

- Do we have `properties` as we do in C#?
- Yes, called `public field declarations`

```
class Dog {
  name = 'Not Named'

  bark() {
    return `${this.name} says bark!`
  }

  eat() {
    return 'mmmm'
  }
}
```

Usage

```
const newDog = new Dog()
newDog.bark() // Not Named says bark!
newDog.name // Not Named
newDog.name = 'Fluffy'
newDog.bark() // Fluffy says bark!
newDog.name // Fluffy
```

Notice the use of `this`

Inside a function of a class, `this` refers to the `current object` and must be used to distinguish a local variable `name` versus the field `this.name`

NOTE: The idea of `this` *in JavaScript can be perplexing and we'll return to it later.*

Constructors

Like C#, JavaScript classes also have constructors. We can allow the constructor to accept arguments and use them to fill in our public fields (and other data).

```
class Dog {
  name = 'Not Named'

  constructor(newName) {
    this.name = newName
  }

  bark() {
    return `${this.name} says bark!`
  }

  eat() {
    return 'mmmm'
  }
}
```

Now when we create a new dog we must give it a proper name.

```
const myPal = new Dog('Fluffy')

myPal.bark() // Fluffy says bark!
myPal.name // Fluffy
```

Subclasses

Again, like C# we have the idea of subclasses.

NOTE: This idea is used heavily in React as we will see.

```
class LoudDog extends Dog {
  bark() {
    return `${this.name.toUpperCase()} SAYS BARK!!!!!`
  }

  yell() {
    return 'I am a loud dog, so I yell!'
  }
}
```

Instantiating subclasses

```
const jack = new LoudDog('Jack')
jack.bark()
```

Constructors in subclasses and super

Subclasses can also have constructors. To ensure the *parent* constructor is called, we use super

```
class LoudDog extends Dog {
  constructor(name) {
    super(name.toUpperCase())
  }

  bark() {
    return `${this.name} SAYS BARK!!!!!`
  }

  yell() {
    return 'I am a loud dog, so I yell!'
  }
}
```

```
const barkeyMcBarkson = new LoudDog('Barkey McBarkson')
barkeyMcBarkson.name // 'BARKEY MCBARKSON'
```

Arrow function methods

There is another way to define methods for a class, to use the public field definition syntax.

```
class Dog {
  name = 'Not Named'

  constructor(newName) {
    this.name = newName
  }

  greet = () => {
    return `Hello I am ${this.name}`
  }

  bark() {
    return `${this.name} says bark!`
  }

  eat() {
    return 'mmmm'
  }
}
```

Understanding `this`

`this` in JavaScript (and thus TypeScript) is different.

- Understanding `this` is challenging
- It is often a *gotcha* interview question
- Easiest way to remember a good answer is:
  - `this` is always the object that called a function
  - *OR* if the function is an arrow function, it is the object in scope when the function was defined

Example time

```
const objectOne = {
  theIdentifier: 'object number one',
  someMethod() {
    console.log(this.theIdentifier)
    console.log(this)
  }
}

objectOne.someMethod()
```

See that this would log `object number one` and `objectOne` as the object

```
const detachedMethod = objectOne.someMethod
detachedMethod()
```

This logs `undefined` and `window` as the object

`Window` **(the global object) is the *caller***

Now with classes

```
class Example {
  theIdentifier = 'object number one'

  someMethod() {
    console.log('---- this ----')
    console.log(this)
    console.log('---- this.theIdentifier ----')
    console.log(this.theIdentifier)
  }
}

const objectOne = new Example()
objectOne.someMethod()
```

Detach the method

```
const detachedMethod = objectOne.someMethod
detachedMethod()
```

`this` becomes `undefined`

Binding `this`

We can use `bind` to tell the object what `this` is when called:

```
const detachedMethodBound = objectOne.someMethod.bind(objectOne)
detachedMethodBound()
```

Binding to whatever variable we like

```
const objectTwo = new Example()
objectTwo.theIdentifier = 'whatever'

const detachedMethodBound = objectOne.someMethod.bind(objectTwo)
detachedMethodBound()
```

Arrow functions!!

Arrow function definition will bind `this` to the object.

```
class Example {
  theIdentifier = 'object number one'

  someMethod = () => {
    console.log('---- this ----')
    console.log(this)
    console.log('---- this.theIdentifier ----')
    console.log(this.theIdentifier)
  }
}

const objectOne = new Example()

const detachedMethod = objectOne.someMethod
detachedMethod()
```

Where would this come up?!?

When using `class` style React components, or using `classes` with `addEventListener` style coding.

How to avoid the `this` confusion.

1. Prefer arrow functions when needed
2. Prefer React `function` based programming over `class` based programming

Is this something a new junior developer needs to worry about?

1. Not really
2. Only to be able to respond to a tricking question during a job interview.