Object-oriented programming in JavaScript

What is an *object*?

An object combines **state**, **behavior**, and **identity**.

→ State is data, known as attributes or properties.

→ Behaviors are known as methods. Methods are just functions that are defined as part of an object.

→ Identity is the object's *class*.

Class

A *class* is a blueprint for an object. In JavaScript, classes normally have names starting with capital letters.

```
let students = new Set()
```

`Set` is a class; `students` is an object.

Composition

A person *has a* job, has family members, has communities.

```
Person
  Job
  Person[] (family)
  Community[]
```

Building objects out of other objects is called *composition*.

Is this better?

- → Claimed benefits
  - → Code reuse
  - → Improved software maintainability
  - → Better design: OOP forces programmers to spend more time in design
  - → Encapsulation: once an object is created, knowledge of its implementation is not necessary to use it

What does a JavaScript object look like?

```
let employee = {
  name: "Phoenix Carter",
  jobTitle: "Front-End Developer",
  startDate: new Date(2018, 7, 1),
  baseSalary: 68000
}
```

But that doesn't have a class!

True. Objects in JavaScript are unusual.

→ They can be created without a class.

→ They don't technically have a class.

→ They are used in the same way hashes/maps/dictionaries are used in other languages.

Can objects without a class have methods?

YES! In JavaScript, we can use functions as values in an object.

```
let phoenix = {
  name: "Phoenix Carter",
  startDate: new Date(2018, 8, 1),
  getFirstName: function () {
    return this.name.split(" ")[0]
  }
}

phoenix.getFirstName() // => Phoenix
```

What is `this`?

→ `this` is like a pronoun.

→ `this` refers to the object that invokes the function where `this` is used.

It changes depending on its context (scope).

```
let getFirstName = function () {
  let fullName = this.name.split(" ")
  return fullName[0]
}

let phoenix = {
  name: "Phoenix Carter",
  getFirstName: getFirstName
}

let morgan = {
  name: "Morgan Nevada",
  getFirstName: getFirstName
}

phoenix.getFirstName() // => what will this return?
morgan.getFirstName()  // => what will this return?
```

But what about `new Date()`?

If objects in JS don't have a class to act as a blueprint, what is happening with `new Date()`?

Prototypes

Objects in JavaScript have a *prototype*. This is another object that attributes and methods are delegated to.

```
// NOTE: You will never have to do this!
let Employee = {
  getFirstName: function () {
    return this.name.split(" ")[0]
  }
}

let phoenix = Object.create(Employee); // what do you think happens here?

Object.assign(phoenix, { name: "Phoenix Carter",
                         startDate: new Date(2017, 8, 1) })

// what does the variable `phoenix` return now?

phoenix.getFirstName() // how does this work?
```

Is there an easier way?

```
function Employee(name, startDate) {
  this.name = name;
  this.startDate = startDate;
}

Employee.prototype.getFirstName = function () {
  return this.name.split(" ")[0]
}

let phoenix = new Employee("Phoenix Carter",
                           new Date(2017, 8, 1))

phoenix.getFirstName() // => Phoenix
Object.getPrototypeOf(phoenix)
// => Employee { getFirstName: [Function] }
```

This seems kind of nuts

True. That's a lot of code to make an object that can respond to `getFirstName()`. It's not clear what's happening. *Prototypal inheritance* is pretty confusing.

But it's important to know about it and be able to recognize it when you see it.

We do have a better way!

```
class Employee {
  constructor(name, startDate) {
    this.name = name;
    this.startDate = startDate;
  }

  getFirstName() {
    return this.name.split(" ")[0]
  }
}

let phoenix = new Employee("Phoenix Carter",
                           new Date(2017, 8, 1))
let alex = new Employee("Alex Corey", new Date(2018, 3, 2))

phoenix.getFirstName()          // => Phoenix
alex.getFirstName()             // => Alex
Object.getPrototypeOf(phoenix) // => Employee {}
```

A couple of questions

→ I thought JS didn't have classes

→ What is `this` again?

`class` != class

ES2015 introduced new keywords to make it look like JavaScript has class-based inheritance. It doesn't actually have class-based inheritance, but it also doesn't really matter, given that the new keywords work just like class-based inheritance.

So why do we have to know this? Primarily for reading others' code.

Looking at `this` one more time

`this` is one of the more confusing things in JavaScript. We can think of it as "the object on which this method is defined," but as we will see later, that's not always the case. It is the object that invokes the function where `this` is used -- whatever is on the left-hand size of the period.

`instanceof`

`instanceof` lets us know if an object is of a particular class (that is, has a particular prototype).

```
let phoenix = new Student("Phoenix")
console.log(phoenix instanceof Student) // => true
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/instanceof

What didn't we cover?

→ Inheritance

→ `super`

→ class expressions

→ `static`

→ getters and setters

Three OO Design Tips

Do one thing well

You may hear this referred to as "the Single Responsibility Principle."

One way to think of this is that a class should only have one reason to change. Imagine you're making a payroll system. The class that calculates the amount of a paycheck shouldn't be the class that actually pays people. If they were, you would have to change it if you changed how pay was calculated and if you changed the method by which people were being paid.

Don't do too much in the constructor

```
// nope
class Game {
  constructor() {
    this.randNum = Math.random()
    console.log("Game starting...")
  }
}

// yep
class Game {
  constructor() {
    this.randNum = Math.random()
  }

  run() {
    console.log("Game starting...")
  }
}
```

Inject the data you need

```
// nope
class Game {
  constructor() {
    this.randNum = Math.random()
  }
}
let game = new Game();

// yep
class Game {
  constructor(randNum) {
    this.randNum = randNum
  }
}
let game = new Game(Math.random());
```