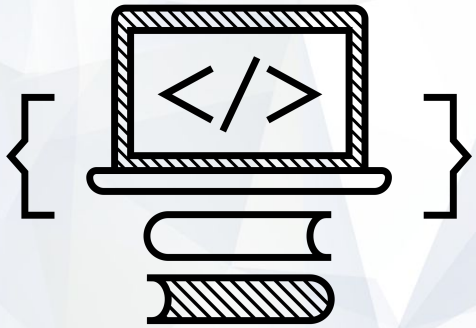
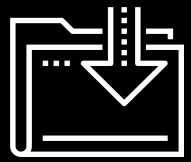
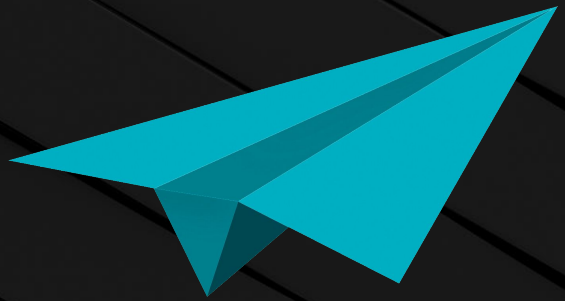


Intro to Object-Oriented Programming

Skills Bootcamp in Front-End Web Development

Lesson 12.1





Office Hours

30 Minutes


WELCOME

Learning Objectives

By the end of class, you will be able to:

Use OOP to create a banking application.

Use constructor functions to create new objects.

Use object prototypes to add methods to objects.

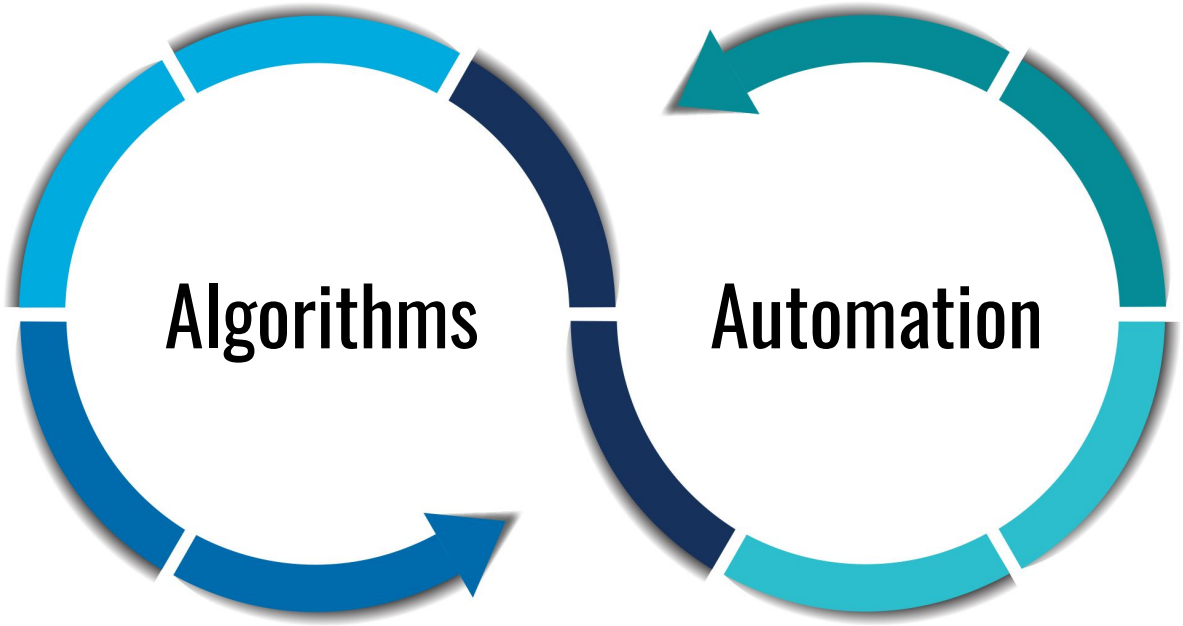
What is programming?

Designing and building an executable program to accomplish a specific computing task. Essentially, programming is problem solving.

What problems do we solve?

Algorithms and Automation

Programming allows us to solve almost any task or problem on a computer.
There are two primary categories:



What is DRY?

Rewriting code wastes time, memory, and can confuse readers and contributors to your code.

What is an object?

Objects

Objects in JavaScript are unordered collections of related data built on a `key:value` structure, where values can be any data type, including functions.

```
const person = {
 name: ['Bob', 'Smith'],
 age: 32,
 gender: 'male',
 interests: ['music', 'skiing'],
 bio: function() {
   alert(this.name[0] + ' ' + this.name[1] + ' is ' + this.age + ' years old. He
likes ' + this.interests[0] + ' and ' + this.interests[1] + '.');
 },
 greeting: function() {
   alert('Hi! I\'m ' + this.name[0] + '.');
 }
};c
```

Why are objects important in JavaScript?

Everything Is an Object!

Well, almost everything.

Data types are objects:

- Array
- Date
- Math
- ...and more!

Even **functions** are objects!

Primitive types are *not* objects:

- Boolean
- Null
- Undefined
- Number
- String
- Symbol

What is object-oriented programming?

Object-Oriented Programming (OOP)

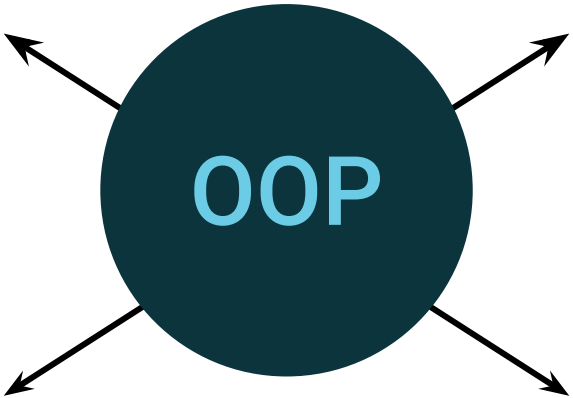
OOP is a programming paradigm, or pattern of programming, centered around objects. Problems are approached as a collection of objects working together to solve a problem.

Encapsulation

Object data (and often functions) can be neatly stored (i.e., encapsulated)

Inheritance

New classes can be created based on other classes (i.e., the Person class is parent to the Student and Teacher classes)



Abstraction

Creating a simple model of something complex

Polymorphism

Multiple object types can implement the same functionality

So, what do you think we are going to do today?

Program some objects!

Activity: Raining Cats and Dogs

In this activity, you will make a cat object and a dog object, each with three keys.

Continue to use ES6 syntax whenever possible!

Suggested Time:

15 minutes

Time's Up!
Let's Review.

Review: Raining Cats and Dogs

We create a `makeNoise` key and give it the value of a function.

```
makeNoise: function() {
  if (this.raining === true) {
    console.log(this.noise);
    }
  }
};
```

We use dot notation to call methods contained in our object.

```
dogs.makeNoise()
```

We can change the value of a key using dot notation as well.

```
cats.raining = true;
```

Review: Raining Cats and Dogs

We create a function `massHysteria` which will take in a dogs object and cats object and check that BOTH have a key:value of `raining: true`.

```
const massHysteria = function(dogs, cats) {
if (dogs.raining === true && cats.raining === true) {
  console.log("DOGS AND CATS LIVING TOGETHER! MASS HYSTERIA!");
  }
};
```

Finally we invoke our function passing in our two objects.

```
massHysteria(dogs, cats);
```

Q What if we wanted to create multiple different animal objects from a blueprint?

A We can use a constructor function to create objects based on a structure we specify.

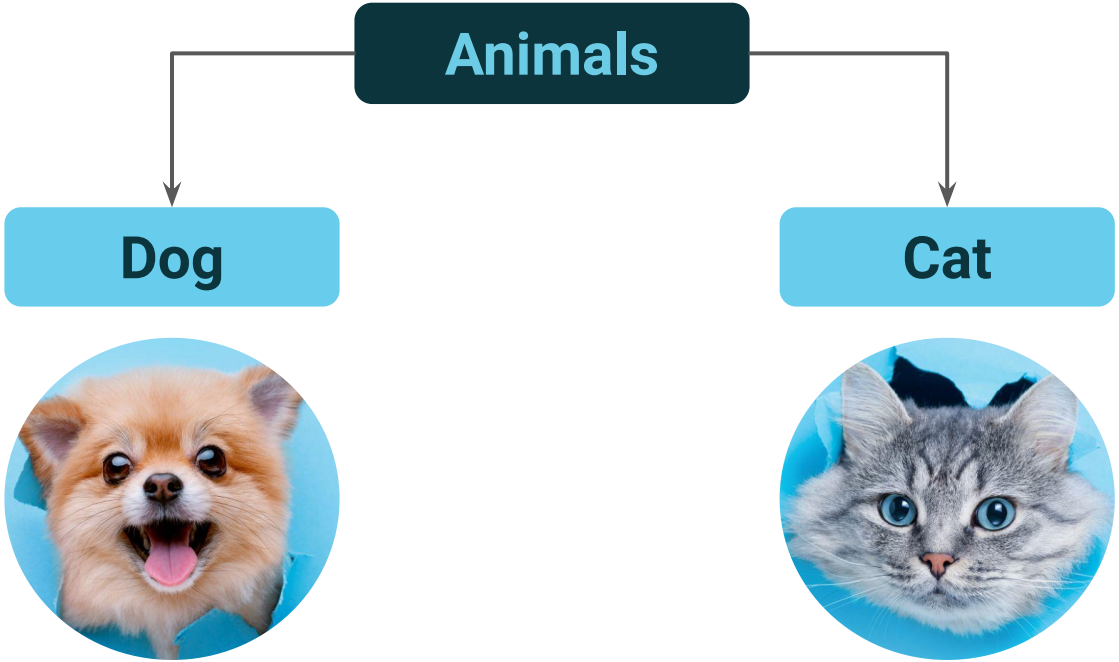
Questions?

Cats and Dogs with Constructors!

What difference do you see from the previous activity we worked on?

Constructor Function

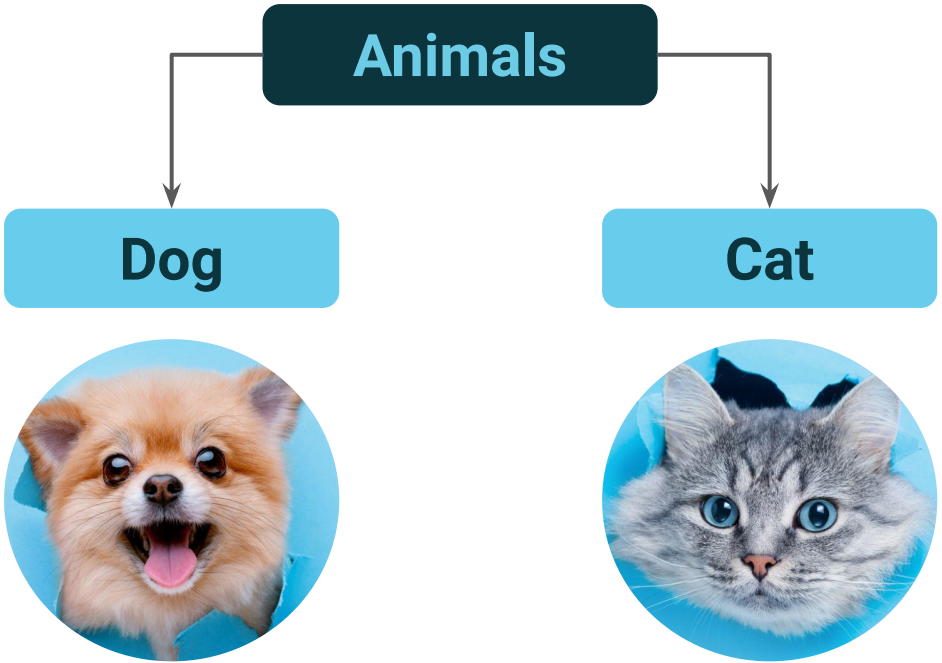
We create a constructor function called `Animals`, instead of individual cat and dog objects.



Constructor Function

Q Why is Animal upper-cased?

A It is a common naming convention to upper-case the names of constructor functions, as well as classes.



Review: Cats and Dogs with Constructors!

We first declare a constructor function named `Animal`. It will take 2 parameters which will be passed into our keys as their value.

```
function Animal(raining, noise) {
  this.raining = raining;
  this.noise = noise;
```

We give our object a key of `makeNoise` whose value is a function. The function checks if the `raining` key's value is `true`. If it is, `console.log` the value of the key `noise`.

```
this.makeNoise = function() {
  if (this.raining === true) {
    console.log(this.noise);
  }
```

Review: Cats and Dogs with Constructors!

We create a new object via our constructor function using the `new` keyword.
We pass in the values we want our keys to have as arguments to the constructor.

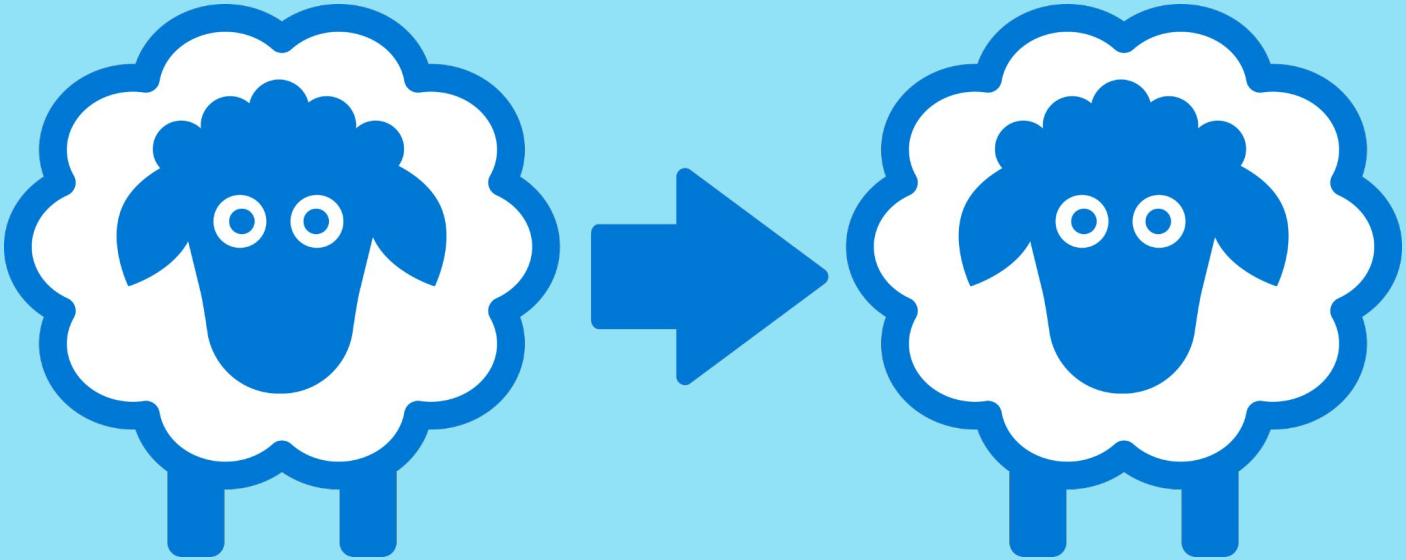
```
var dogs = new Animal(true, "Woof!");
var cats = new Animal(false, "Meow!");
```

We can now invoke the `makeNoise` method on our created objects.

```
dogs.makeNoise();
cats.makeNoise();
```

How are constructors useful?

They allow us to create as many objects as we want, all from a single blueprint. This lessens redundant code.



Activity: MiniBank

In this activity, you will use objects to create a mini banking application.

Suggested Time:

20 minutes

Time's Up! Let's Review.

Review: Mini-Bank

We create a constructor function called `MiniBank` that will take in one argument, the starting balance.

```
function MiniBank(balance) {
```

We use `this.` notation to declare functions on our constructor so all objects created from this constructor have access to those methods.

```
this.getBalance = () => {
  return this.balance;
};
```

Review: Mini-Bank

In our `deposit` function, we first check to make sure the provided argument is a number and the number is greater than 0. We then set our `newBalance` to be equal to our current balance plus the current given value of the deposit.

```
this.deposit = value => {
  if (typeof value !== "number" || value <= 0) {
    throw new Error("'value' must be a positive number!");
  }
  const newBalance = this.getBalance() + value;
```

After we have gotten our `newBalance`, we invoke the setBalance function and `updateStatement()` function. Finally, we console log the deposited value.

```
  this.setBalance(newBalance);
  this.updateStatement(newBalance);
  console.log(`Deposited ${value}!`);
};
```

Review: Mini-Bank

To create our new mini bank via our constructor, we invoke the constructor using the `new` keyword.

```
const bank = new MiniBank(0);
```

Now we can call any of the functions we coded into the constructor earlier.

```
bank.printBalance();
bank.deposit(85);
bank.printBalance();
bank.withdraw(20);
bank.printBalance();
bank.printStatement();
```

How does OOP make solving this activity easier?

The use of objects and constructors allows us to create a single blueprint that we can then use to create as many instances of our MiniBank as we like.

Questions?

Activity: Weather Admin

In this activity, you will will create a CLI-based weather application that will give updates about the weather at the searched location.

Suggested Time:

15 minutes

Time's Up! Let's Review.

Review: Weather Admin

First, we require the npm package `weather-js`.

```
const weather = require("weather-js");
```

We create a constructor function called UserSearch that will take a `name` and `location` as arguments. It will also use `Date.now();` to get the current date.

```
const UserSearch = function(name, location) {
this.name = name;
this.location = location;
this.date = Date.now();
```

Review: Weather Admin

Our constructor also has a method of `getWeather`. It will make use of the `weather-js` search function to search for weather of a given location.

Lastly, we export our `UserSearch` constructor.

```
this.getWeather = function() {
  weather.find({ search: this.location, degreeType: "F" }, function(err, result) {
    if (err) {
      console.log(err);
    }
    console.log(JSON.stringify(result, null, 2));
    });
  };
};

module.exports = UserSearch;
```

Review: Weather Admin

First, we require all the pieces necessary. `fs` is the File System, allowing us to create, delete, or update files on a user's local machine. `UserSearch` is our constructor function we exported from `UserSearch.js`. Finally, we import `moment`, an NPM package for dates and times.

```
var fs = require("fs");
var UserSearch = require("./UserSearch");
var moment = require("moment");
```

We create a constructor function called `WeatherAdmin`. It is given a method of `getData` which will use the file system to read a `log.txt` file if it exists and log that data to the console.

```
var WeatherAdmin = function() {
this.getData = function() {
  fs.readFile("log.txt", "utf8", function(error, data) {
    console.log(data);
  });
};
```

Review: Weather Admin

`WeatherAdmin` also gets a method of `newUserSearch`. This method takes in two arguments, name and location, much like our `UserSearch` constructor. This is so we can pass those two arguments along into the `UserSearch` constructor, as this method will instantiate a new `UserSearch` object and save it to a variable of `newUserSearch`.

```
this.newUserSearch = function(name, location) {
  var newUserSearch = new UserSearch(name, location);
```

We set our `logTxt` variable to equal a string we build that will display the name, location, and date of the search. We then call `moment` to get the date, and format it to `MM-DD-YYYY`.

```
var logTxt =
"\nName: " +
newUserSearch.name +
" Location: " +
newUserSearch.location +
" Date: " +
moment(newUserSearch.date).format("MM-DD-YYYY");
```

Review: Weather Admin

We use the `fs.appendFile` method to append the current value of `logTxt` to our `log.txt` file.

```
fs.appendFile("log.txt", logTxt, function(err) {
  if (err) throw err;
});
```

Next, we call the `getWeather` method on our `newUserSearch` object.

```
newUserSearch.getWeather();
```

Finally, we export our `WeatherAdmin` constructor.

```
module.exports = WeatherAdmin;
```

Review: Weather Admin

First, we require our WeatherAdmin export from `WeatherAdmin.js`

```
const WeatherAdmin = require("./WeatherAdmin");
```

We use `process.argv`, taking the 3rd argument to find out if the value is `admin` or `user`.

```
const loginType = process.argv[2];
```

We also need `Users` to provide a name and location.

```
const userName = process.argv[3];
const userLocation = process.argv[4];
```

Review: Weather Admin

We create an instance of the `WeatherAdmin`. If they are running it as an `admin`, run the `getData` method. If they are not an `admin`, we will run `newUserSearch`, passing the arguments from the command line.

```
const myAdmin = new WeatherAdmin();

if (loginType === "admin") {
  myAdmin.getData();
}
else {
  myAdmin.newUserSearch(userName, userLocation);
}
```


Break

Introduce Prototypes

Introduce Prototypes

Key points:

Objects, arrays, and primitives all have a `.prototype.`

The `.prototype.` has methods and properties attached to it.

Methods declared on the prototype are declared once and memory is allocated for them once, but all objects made from it have access.

Instance methods only exist on a particular instance of an object; prototype methods are on all instances.

Introduce Prototypes

We create an array and console log it. Next, we call the `.forEach` and `.map` methods on it.

```
myArray = [2, 4, 6, 8];
console.log(myArray);

myArray.forEach((num) => console.log(num));

myArray.map((x) => console.log(x * 2));
```

Next, we console log the string `Hello`. We then call `"Hello”.toLowerCase.`

```
console.log("Hello");
console.log("Hello".toLowerCase());

console.log(1337);
console.log((1337).toString());
```

Where did the `.toLowerCase` come from?

Introduce Prototypes

While those two methods did not show up when we console logged our string, the prototype has these methods built in. Arrays, Objects, and even primitives all have a prototype from which they take their structure and methods. Any of of these that you create will have the prototype methods available to it via the `.prototype.` (i.e., `Array.prototype.forEach()`).

We created a constructor function named `Movie`, which will take in two arguments, `title` and `releaseYear`.

```
function Movie(title, releaseYear) {
  this.title = title;
  this.releaseYear = releaseYear;
}
```

What if we wanted to add a method to our constructor later on in our code?

Introduce Prototypes

We would add that method to the `Movie.prototype`.

We declare the title of our method, which will be `logInfo`. We do so by typing `Movie.prototype.logInfo = function(){}`.

- We can only add to our constructor via the object prototype.
- When we add a method to an object's prototype, all the objects made from it will get it.
- If there is something that's going to be the same between objects and isn't going to change, it should be on the prototype.
- If it is defined on the prototype, it is only defined once, and memory for it is only allocated once.

```
Movie.prototype.logInfo = function() {
  console.log(`${this.title} was released in ${this.releaseYear}`);
};
```

Introduce Prototypes

When we create a new object via our `Movie` constructor, it will have access to all the methods defined in the constructor and those that have been added to its prototype.

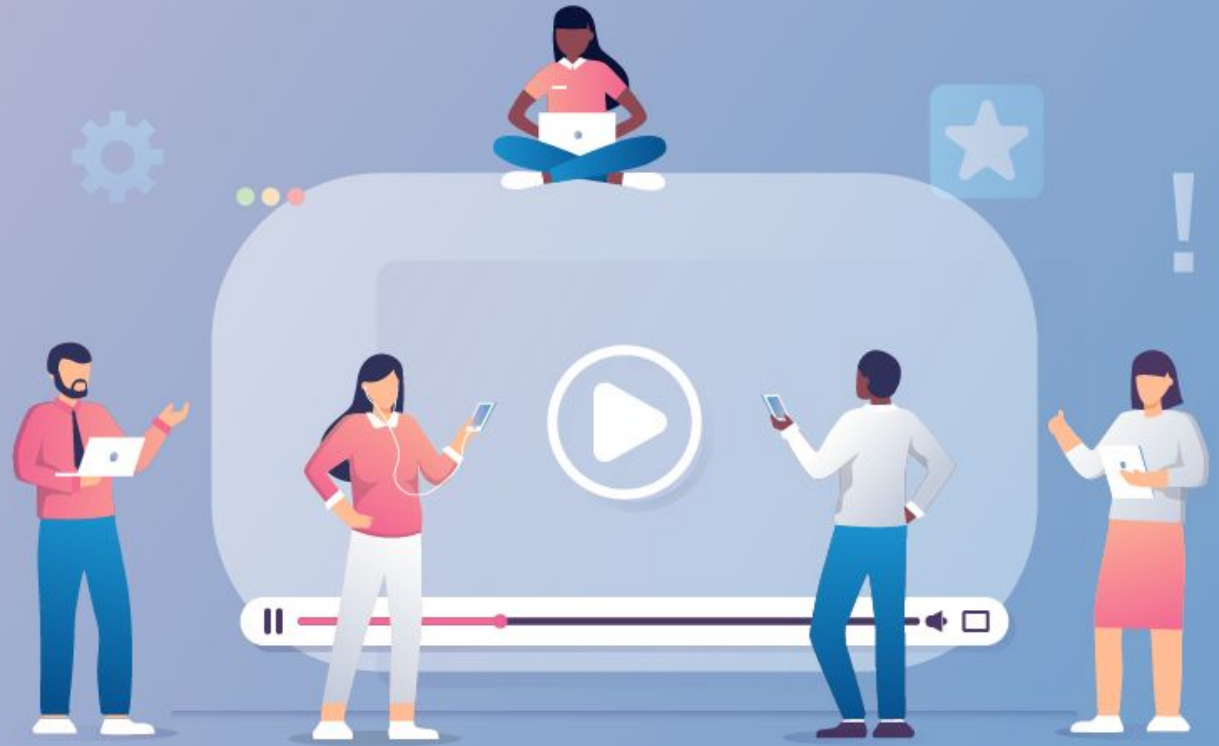
```
const theShining = new Movie("The Shining", 1980)
theShining.logInfo();
```

Objects also have their own prototype methods built in. Even though our object was created via a constructor function, it still has access to all the built-in object prototype methods.

```
console.log(theShining.hasOwnProperty('title'));
console.log(theShining.hasOwnProperty('logInfo'));
console.log(Movie.prototype.hasOwnProperty('logInfo'));
```

What does the Object Prototype allow us to do?

It allows us to reuse properties and methods between objects that need to share them (i.e., all movies can share the same logInfo methods, but get their own unique name and release year).



Activity: RPG Prototype

In this activity, you will generate RPG characters using Objects and prototypes.

Suggested Time:

20 minutes

Time's Up! Let's Review.

Review: RPG Prototype

We create a `Character` constructor that will take in 5 arguments. We assign those arguments to keys in our constructor.

```
function Character(name, profession, age, strength, hitpoints) {
  this.name = name;
  this.profession = profession;
  this.age = age;
  this.strength = strength;
  this.hitpoints = hitpoints;
}
```

Review: RPG Prototype

We add an `isAlive` function to our object prototype.

```
Character.prototype.isAlive = function() {
  if (this.hitpoints > 0) {
    console.log(this.name + " is still alive!");
    console.log("\n-------------\n");
    return true;
  }
  console.log(this.name + " has died!");
  return false;
};
```

Review: RPG Prototype

We also add two other functions to our `prototype`. The `attack` method takes in a second object and decreases their “hitpoints” by this character's strength. The `levelUp` method increases `this` character’s stats when called.

```
Character.prototype.attack = function(character2) {
  character2.hitpoints -= this.strength;
};

Character.prototype.levelUp = function() {
  this.age += 1;
  this.strength += 5;
  this.hitpoints += 25;
};
```

Review: RPG Prototype

Finally we can use our constructor to create two characters, calling their methods from the prototype that we added.

```
var warrior = new Character("Crusher", "Warrior", 25, 10, 75);
var rogue = new Character("Dodger", "Rogue", 23, 20, 50);

warrior.printStats();
rogue.printStats();

rogue.attack(warrior);
warrior.printStats();
warrior.isAlive();

rogue.levelUp();
rogue.printStats();
```

Why don’t we just declare the methods in the constructor?

When we bind a function using the `this` keyword, the method only exists on that instance of the object.

For any method bound to `this`, it will be re-declared with each new instance of an object.

How does the prototype help us solve this problem?

The prototype allows us to declare methods that will be attached to all instances of an object of that prototype.

Because the method is applied to the prototype,
it is only stored in memory once for all instances.

Activity: Tamagotchi App

In this activity, you will create your own basic Tamagotchi clone using constructors.

Suggested Time:

25 minutes

Time's Up! Let's Review.

Review: Tamagotchi

We first create a constructor function named `DigitalPal`.

It takes no arguments, as the value of the keys are predefined.

```
var DigitalPal = function() {
  this.hungry = false;
  this.sleepy = false;
  this.bored = true;
  this.age = 0;
};
```

Review: Tamagotchi

We create a function feed and attach it to the `.prototype.`. The method `feeds` the DigitalPal when they are hungry and sets them to sleepy.

```
DigitalPal.prototype.feed = function() {
  if (this.hungry) {
    console.log("That was yummy!");
    this.hungry = false;
    this.sleepy = true;
  } else {
    console.log("No thanks, I'm full.");
  }
};
```

Review: Tamagotchi

We create a method called `sleep`, which puts the DigitalPal to sleep when they are sleepy. It also invokes the `increaseAge` function.

```
DigitalPal.prototype.sleep = function() {
  if (this.sleepy) {
    console.log("ZZzzZZZzzZZz~~");
    this.sleepy = false;
    this.bored = true;
    this.increaseAge();
  }
  else {
    console.log("No way! I'm not tired!");
  }
};
```

Review: Tamagotchi

We create a method called `play` which allows the user to play with their DigitalPal when they are bored and sets hungry to `true` and bored to `false`.

```
DigitalPal.prototype.play = function() {
  if (this.bored) {
    console.log("Yay! Let's play!");
    this.bored = false;
    this.hungry = true;
  }
  else {
    console.log("Not right now. Maybe later?");
  }
};
```

Review: Tamagotchi

This is the `increaseAge` method which is called within our `sleep` method.
It will increase the age of our DigitalPal by 1.

```
DigitalPal.prototype.increaseAge = function() {
  this.age++;
  console.log("Happy Birthday to me! I am " + this.age + " old!");
};
```

Review: Tamagotchi

The `destroyFurniture` will allow us to decrease our `houseQuality`.

```
DigitalPal.prototype.destroyFurniture = function() {
  if (this.houseQuality - 10 > 0) {
    this.houseQuality -= 10;
    this.bored = false;
    this.sleepy = true;
    console.log("MUAHAHAHAHA! TAKE THAT FURNITURE!");
  }
  else {
    console.log("I've already destroyed it all!");
  }
};
```

Review: Tamagotchi

We create and attach a `letOutside` function to our prototype that will let our pet outside and make them bark!

```
DigitalPal.prototype.letOutside = function() {
  if (!this.outside) {
    console.log("Yay! I love the outdoors!");
    this.outside = true;
    this.bark();
  }
  else {
    console.log("We're already outside though...");
  }
};a
```

Review: Tamagotchi

We create and attach a `letInside` function to our prototype that will let our pet back inside.

```
DigitalPal.prototype.letInside = function() {
  if (this.outside) {
    console.log("Aww... Do I have to?");
    this.outside = false;
  }
  else {"We're already inside though...""We're already outside
though...");
  }
};
```

Review: Tamagotchi

Finally, we can grab the command-line arguments provided by the user and use store them in variables called `animal` and `method`.

```
var animal = process.argv[2];
var method = process.argv[3];
```

Questions?


The End