Object-Oriented Javascript

Objective:

know what an object is,

understand the many advantages of object oriented programming.

OOP development in JavaScript.

Prerequisite:

Object Literals, loops, arrays and other basic programming syntax.

In this course you'll learn new terms such as class, property, and method, as well as ES 2015 syntax for working with objects.

This is a basic OOP course, and we Inheritance and Prototypes won't be covered in this course.

What Is an Object

An object is a package of information about something that contains a group of properties and functions that work together to represent something in your program.

An object's properties are a series of key value pairs that hold information about the object where its functions are called methods.

Methods let your object do something or let something be done to it. The term object oriented programming is a way of thinking about in designing a computer program that uses objects.

A lot of developers find it helpful to think of object's in a code as a way to model real life objects. Real life objects have states and behaviors and so the JavaScript objects.

In JavaScript states are represented by the object's properties and behaviors are represented by the object's methods. So if we were helping to represent a car in our code. We can make a car object with properties like type, make, mode, colour and year built. Or if we want to model a radio. We could make a radio object with properties like station and volume and methods like turn off or change station.

Just like these object's states can be changed in real life, we can paint a car or build an addition. Tune a radio or increase the volume. We can change these states in our code by updating an object's properties.

The idea of mimicking or modelling real life objects is a pretty good way to try and think about objects, properties and methods when we're first learning about them. But it's not always super realistic. Sometimes we are trying to model real life objects like cars or radios in our code. But more often, our objects are a little more abstract or less analogous to things we might use in everyday life.

Most of the time when we design objects we're creating an easy way to store information about something via properties, and access, manipulate or utilize that information via methods.

These objects are more like data containers than abstractions of rea life objects. For example, you can create objects to help manage things like users. Some of the properties were username, birthday, status and number of friends. The methods can be addFriend() and updateStatus(). Or, you also can use objects when building API wrappers to share on GitHub with fellow programmers.

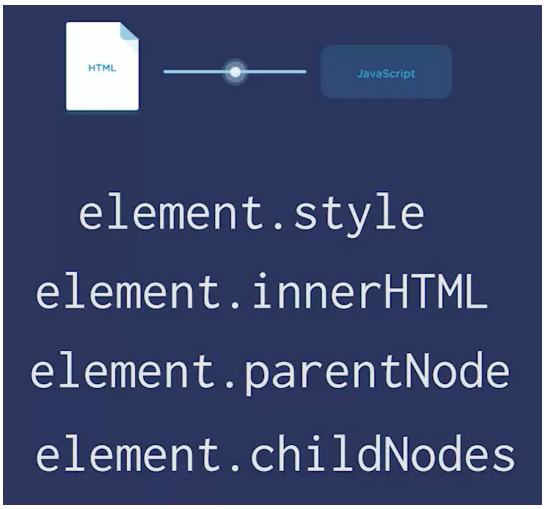
```
class User {
  constructor(username, birthday, status, numFriends) {
    this.username = username;
    this.birthday = birthday;
    this.status = status;
    this.numFriends = numFriends;
  }

  addFriend(){
   //add friends
  }

  updateStatus(){
   // update status
  }
}
```

JavaScript Objects

in JavaScript that you're already familiar with. Most everything you encounter when programming in JavaScript is an object or can be treated like an object. For example, you have with the DOM, or the Document Object Model. That's just the object that represents the HTML document your JavaScript interacts with.

DOM elements are objects and have properties and methods. For example, I can get an element styles with the **style** property. I can get the HTML that an element is made of with the **innerHTML** property.

I can even get an element's parent with the **parentNode** property, or its children with the **childNodes** property.



DOM objects have methods as well, like **getElementById** or **appendChild**.

Arrays are another example of a JavaScript object. Arrays have properties like length and methods like **push**, **pop**, and **concat**. What's great about these objects is that they've abstracted away a lot of the details for you.

In other words, they created an interface that lets you use the object without having to know what's going on under the hood.

You can use the method, getElementById, over and over without having to know how that works or rewrite all the code that's underneath it. As you can see, objects can make it really easy to get up to speed with coding.

You can design your own to model real life objects that you want to use in your code, or use them as a container to store and access data.

Object Literals and Components of Objects

Object literals are one way to create an object and they're really great when you're modeling only one single specific thing. Let's look at an example. Let’s say we have a dog named “ernie”

```
const ernie = {
    animal: 'dog',
    age: 1,
    breed: 'pug',
    bark: function(){
       console.log('Woof!');
    }
}
```

Let's practice them a bit in a code challenge after this video.2:42

After that, come back and join me to learn all about how we access and2:45

use properties and methods.

Exercise:

Fill in the missing code for the object literal.

To do this, inside the curly braces, create three key value pairs for properties name, color, and isTurn.

The values for `name` and `color` should be any string value, and the value for `isTurn` should be true.

```
const player1 = {



}
```

Now, add an empty method to the object literal called `play()`.

```

```

Dot Notation & Bracket Notation

You can access object properties in two ways:

```
objectName.propertyName
```

Or

```
objectName["propertyName"]
```

Examples:

```
person.lastName;

person["lastName"];
```

Read (source: https://www.w3schools.com/js/js_objects.asp) for more information.

The following code provides some examples for using object literals

```
// 1-
// an object literal example
student = {
    name: "john",
    age: 22,
    grades: [99, 77, 88],

    regiter: function() {
        console.log('student registeraion complete')
    }
}

// 2-
// accessing object properties
console.log(student.name);
console.log(student.age);
console.log(student.grades);
console.log(student.grades[0]);
console.log(student["age"]);

// 3-
// using objects in context
var greetings = `hi, my name is ${student.name}`;
console.log(greetings);

// 4-x
//
for (let propname in student) {
```

```
    // console.log (student.propname)
    console.log (student[propname]);
}

// 5 - methods
student.regiter();


// 6- using variables for bracket notation
bracket = 'age'
console.log(student[bracket]);
```

Exercise:

Inside the play method, write an empty if statement that checks if it's the players turn. Use dot notation.

```
const player1 = {
    name: 'Ashley',
    color: 'purple',
    isTurn: true,
    play: function(){
        // write code here.
    }
}
```

Inside the if statement, return a string equal to the value of the name property followed by the string " is now playing!". Use bracket notation.

```
const player1 = {
   name: 'Ashley',
   color: 'purple',
   isTurn: true,
   play: function(){
    if (this.isTurn == true) {
     return `${this.name} is now playing!`
    }
      // write code here.
   }
}
```

Answer to the exercise (pay attention to **this** keyword):

```
const player1 = {
    name: 'Ashley',
    color: 'purple',
    isTurn: true,
    play: function(){
      if (this.isTurn == true) {
        return `${this.name} is now playing!`
      }
    }
}

console.log(player1.play())
```

Changing and Adding Properties

You can change or add properties to an object by just using dot or bracket notation. It was briefly mentioned in the above. For example, you can implement the following changes:

```
player1.color = 'red'
player1.age = 22 (this will add a new property)
```

Using Class

When you have similar objects such as different players with similar property attributes, we can use Class to create a model and create each object.

For example see the following code for some pet objects :

```
const ernie = {
    animal: 'dog',
    age: '1',
    breed: 'pug',
    bark: function(){
        console.log('Woof!');
    }
}

const vera = {
    animal: 'dog',
    age: 8,
```

```
    breed: 'Border Collie',
    bark: function(){
        console.log('Woof!');
    }
}

const scofield = {
    animal: 'dog',
    age: 6,
    breed: 'Doberman',
    bark: function(){
        console.log('Woof!');
    }
}

const edel = {
    animal: 'dog',
    age: 7,
    breed: 'German Shorthaired Pointer',
    bark: function(){
        console.log('Woof!');
    }
}
```

We can use a Class to represent each object as the following code:

```
class Pet {
  constructor(animal, age, breed) {
    this.animal = animal;
    this.age = age;
    this.breed = breed;
  }
```

We use this same pattern in the constructor method, except instead of the object name, we are using keyword this, because the **this** keyword inside a constructor method is referring to the object that is being created. The keyword this has different meanings depending on where it's used, which is often confusing, even to experienced JavaScript developers.

Now, we can create each object with **new** keyword.

```
class Pet {
  constructor(animal, age, breed) {
    this.animal = animal;
    this.age = age;
    this.breed = breed;
```

```
  }
}

const ernie = new Pet('dog', 1, 'pug');
const vera = new Pet('dog', 8, 'border collie');
```

Adding Methods To a Class

Let's say if we want to add a method **speak()** to each object. We can add the method to the class as follows:

```
class Pet {
  constructor(animal, age, breed, sound) {
    this.animal = animal;
    this.age = age;
    this.breed = breed;
    this.sound = sound;
  }

  speak () {
    console.log(this.sound);
  }
}

const ernie = new Pet('dog', 1, 'pug', 'woof');
const vera = new Pet('dog', 8, 'border collie', 'woof woof');

console.log(ernie.breed);
ernie.speak();
vera.speak();
```

Exercise:

Create an empty method called stringGPA() and add it to the Student class, and Inside the stringGPA() method, convert the value of the gpa property to a string and return it.

```
class Student {
  constructor(gpa) {
    this.gpa = gpa;
  }

  }
}
```

Answer

```
class Student {
  constructor(gpa) {
    this.gpa = gpa;
  }

  stringGPA() {
    return this.gpa.toString();
  }
}
```

Getters and Setters

With getter and setter methods, you have the option to include logic when retrieving or setting the value of a property while still enjoying the simple syntax of accessing and setting these properties directly.

In JavaScript Getter is a special method used when you want to have a property that has a dynamic or computed value. The value of the property is computed in the getter method. And even though it's not stored or attached to the object it can be accessed just like a regular property.

For this example we're going to add a property called activity to the pet class. This property should tell us what our pet is up to based on the time of day. Unfortunately, we can't set that property in our constructor method, because its value is dynamic. It changes depending on what time it is. This is a perfect opportunity to use the getter method.

See the following code:

```
class Pet {
  constructor(animal, age, breed, sound) {
    this.animal = animal;
    this.age = age;
    this.breed = breed;
    this.sound = sound;
  }

  get activity() {
    const today = new Date();
    const hour = today.getHours();

    if (hour > 8 && hour <= 20) {
      return 'playing';
    } else {
```

```
      return 'sleeping';
    }

  }

  speak() {
    console.log(this.sound);
  }

}

const ernie = new Pet('dog', 1, 'pug', 'yip yip');
const vera = new Pet('dog', 8, 'border collie', 'woof woof');

console.log(ernie);
```

This special method allowed you to create and dynamically retrieve the value of a property called activity. Even though this property wasn't declared and set in a constructor method like the other properties were, you can access its value the same way, using dot or bracket notation.

But remember if you were to output the ernie object to the console, you would not see the new activity property. A value for the activity property is computed and returned from the getter method when we access it, but never actually attached to the object.

Accessing **ernie.activity** returns correspondin value, but if you **console.log(ernie)** you won't see **activity** as one of the attributes.

Exercise:

Inside the Student class, create an empty getter method called level(). And then the getter method should return the level of a student, based on how many credits (this.credits) they have.

If student has more than 90 credits, they are a 'Senior'.
If the student has 90 or fewer credits, but more than 60 (>= 61), they are a 'Junior'.
If the student has 60 or fewer credits, but more than 30 (>= 31), they are a 'Sophomore'.
If the student has 30 or fewer credits, they are a 'Freshman'.

Think about how you craft your conditional statement, keeping in mind that simplicity is a good objective.

```
class Student {
    constructor(gpa, credits){
        this.gpa = gpa;
        this.credits = credits;
    }


    stringGPA() {
        return this.gpa.toString();
    }
}
```

Answer:

```
class Student {
    constructor(gpa, credits){
        this.gpa = gpa;
        this.credits = credits;
    }

  get level() {
    let score = this.credits;
     if (score >90 ) { return 'Senior'} else if (score >=61 && score<=90)
{return 'Junior'} else if (score >=31 && score<=60) {return 'Sophomore'} else
{return 'Freshman'}
  }

    stringGPA() {
```

```
        return this.gpa.toString();
    }
}

const student = new Student(3.9, 91);
console.log(student.credits)
console.log(student.level)
```

Setters

A getter method is called, a property value is computed and returned, but this value is not ever updated or stored anywhere. A setter method, on the other hand, receives a value and can perform logic on that value if need be. Then it either updates an existing property with that value or stores the value to a new property.

For this example, we're going to add a setter method that sets an owner property for the pet class.

See the following code:

```
class Pet {
  constructor(animal, age, breed, sound) {
    this.animal = animal;
    this.age = age;
    this.breed = breed;
    this.sound = sound;
  }

  get activity() {
    const today = new Date();
    const hour = today.getHours();

    if (hour > 8 && hour <= 20) {
      return 'playing';
    } else {
      return 'sleeping';
    }
  }

  get owner() {
    return this._owner;
  }

  set owner(owner) {
    this._owner = owner;
    console.log(`setter called: ${owner}`);
```

```
  }


  speak() {
    console.log(this.sound);
  }
}

const ernie = new Pet('dog', 1, 'pug', 'yip yip');
const vera = new Pet('dog', 8, 'border collie', 'woof woof');

ernie.owner = 'Ashley';
```

A setter method is created by typing the key word set followed by the name of the property being set, which in our case is owner. And then our usual parentheses and curly braces.

Setters always receive exactly one parameter. The parameter is the value of the property that we'd like to set In this example, it will be the owner's name. Inside our method, we have to set an owner property equal to the parameter we passed in. But the name of a property can never be the same as the name of a getter or setter method. This means we can't do this. We have to create a property name that's different from owner to hold the value of owner.

This is called a backing property. And while we can name the backing property however we would like, convention dictates to use the name of the setter function but with an underscore before it. While we're in our setter method, let's also add a console log.

When we set the owner property, we've got to create a getter method called owner that returns the value of the backing property.

Exercise:

Inside the Student class, create an empty setter method called "major()". Then Inside the major() setter method, set the student's major to a backing property "_major". If the student's level is Junior or Senior, the value of the backing property should be equal to the parameter passed to the setter method. If the student is only a Freshman or Sophomore, set the "_major" backing property equal to 'None'.

```
class Student {
  constructor(gpa, credits) {
    this.gpa = gpa;
    this.credits = credits;
  }

  stringGPA() {
```

```
    return this.gpa.toString();
  }

  get level() {
    if (this.credits > 90) {
      return 'Senior';
    } else if (this.credits > 60) {
      return 'Junior';
    } else if (this.credits > 30) {
      return 'Sophomore';
    } else {
      return 'Freshman';
    }
  }
}

var student = new Student(3.9, 60);
```

Object Interaction

The value of an object's property can be another object. OOP is a programming paradigm that uses objects. These objects are designed to interact with one another.

See the following code:

```
class Pet {
  constructor(animal, age, breed, sound) {
    this.animal = animal;
    this.age = age;
    this.breed = breed;
    this.sound = sound;
  }

  get activity() {
    const today = new Date();
    const hour = today.getHours();

    if (hour > 8 && hour <= 20) {
      return 'playing';
    } else {
      return 'sleeping';
    }
  }
```

```
  get owner() {
    return this._owner;
  }

  set owner(owner) {
    this._owner = owner;
    console.log(`setter calld: ${owner}`);
  }

  speak() {
    console.log(this.sound);
  }
}

class Owner {
  constructor(name, address) {
    this.name = name;
    this.address = address;
  }

  set phone(phone) {
    // replace all nun numeric characters with '' using Regex
    const phoneNormalized = phone.replace(/[^0-9]/g, '');
    this._phone = phoneNormalized;
  }

  get phone() {
    return this._phone;
  }
}

const ernie = new Pet('dog', 1, 'pug', 'yip yip');
const vera = new Pet('dog', 8, 'border collie', 'woof woof');

ernie.owner = new Owner('Ashley', '123 Main Street');
ernie.owner.phone = '(123) 456-7890';

console.log(ernie.owner);
console.log(ernie.owner.name);
console.log(ernie.owner._phone);
```

In this example, we treat owner as an object that has three properties, **name, address, and phone**. As you can see we add **phone** property using getter and setter as we want to modify the input by deleting all non-numeric characters from the phone number.