Web Front-End Development
Week 5: JavaScript OOP and ES6+

In Object-Oriented Programming we use classes and objects to model real world things that we want to represent inside our programs, and/or provide a simple way to access functionality that would otherwise be hard or impossible to make use of.
We've talked about how an object is any entity, or thing, that you can describe.
In object-oriented programming objects can contain related data and behavior that represent information about the thing you're trying to model.

- Properties – attributes or characteristics of the object
    - Data values
- Methods – behavior or tasks an object can perform
    - Operations like functions
- Events – situations that happen
    - situations the object can send notifications about

We looked at the Document Object Model which represents a web page and now we're going to go deeper into objects in JavaScript.

JavaScript Objects
In JavaScript an object is a collection of properties. The properties of an object define the characteristics of the object.
A property is an association between a key and a value, similar to associative arrays. You can think of a property as a variable with a value. But properties are associated with an object.
A property name can be any valid JavaScript string, just like a variable.
A value can be any valid value including an object itself. A property's value can be a function, in which case the property is known as a method.

Methods are tasks associated with an object. They are the same as functions but associated with an object.

Putting data properties and methods together in an object is called encapsulation.

<u>Creating Objects</u>
In addition to objects that are predefined in the browser, you can define your own objects. There are three common ways to create objects.

1) Literal Notation (aka object initializers):

- define an object in { }
- The object is a series of key value pairs separated by commas
    - property : value
- Assign it to a variable if you need to refer to it in your code

https://repl.it/@aileenjp/ECMAScript5-example

Example:

```
var dog = {};
console.log(dog);
```

Add some properties:

```
var dog = {
name: "Hazel",
age: 1
};
```

Accessing Objects

You access the properties of an object with dot-notation, same as with the DOM, or bracket notation like associative arrays.
You can add properties to objects as well.
You can remove properties using 'delete'.

Example:

```
console.log(dog.name);
console.log(dog.age);
console.log(dog['name']);

dog.gender = "female";
console.log(dog.gender);
console.log(dog);
dog.age=2;
console.log(dog.age);
```

Methods

A method is a function associated with an object.
A method is a property of an object that is a function.
Methods are defined the way normal functions are defined, except that they have to be assigned as the property of an object.
The keyword 'this' refers to the current calling object in a method.

Example:

Update dog to have some methods.

```
  bark: function(){
    console.log ("WOOF");
  },
  speak: function(){
    console.log ("Hi, my name is " + this.name);
  }

dog.bark();
dog.speak();
```

Create a second object.

```
var dog2 = {
  name: 'Buddy',
  age: 1,
  bark: function(){
    console.log ("WOOF");
  },
  speak: function(){
    console.log ("Hi, my name is " + this.name);
  }
};

dog2.speak();
```

2) Constructor function:

- Define the object type by writing a constructor function
  - Create a function that specifies its name, properties, and methods.
  - The convention is to use a capital initial letter
  - Keyword 'this' is used to access the object's properties (*this* object)
- The constructor function is JavaScript's version of a class.
- Create an instance of the object with the keyword 'new' which tells the browser to create a new object.
- You can create as many instances as you want, each with their own values

Example:

```
function Dog(name, age){
  this.name=name;
  this.age=age;
  this.greeting=function(){
    console.log("Hi, my name is " + this.name + " and I'm " +
this.age + " years old");
  };
}

var dog3 = new Dog("Nikki", 8);
var dog4 = new Dog("Cole", 12);
dog3.greeting();
dog4.greeting();
```

3) Object() constructor

- JavaScript's Object class has a constructor Object() that creates a new object
  - You can then assign properties and methods
  - Or you can pass in the properties and methods when you call new Object()
- Use Object.create() to create a new object based on an existing object which is useful for inheritance

Example:

```
var dog5 = new Object();
dog5.name="Casey";
dog5.age=4;
dog5.greeting=function(){
  console.log("Hi, my name is " + this.name + " and I'm " +
this.age + " years old");
  };
dog5.greeting();
console.log(dog5);

var dog6=Object.create(dog5);
console.log(dog6.name);
dog6.name="Gizmo";
console.log(dog6.name);
```

In JavaScript objects are a reference type. Two distinct objects are never equal, even if they have the same properties. Only comparing the same object reference with itself yields true.

Inheritance

In JavaScript each object has a prototype object, which acts as a template object that it inherits methods and properties from. And that object's prototype object may also have a prototype object which it inherits from, and so on. This is often referred to as a prototype chain.
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object
All objects in JavaScript are descended from Object. All objects inherit methods and properties from Object.prototype, although they may be overridden.
The prototype property's value is an object, which is basically a bucket for storing properties and methods that we want to be inherited by objects further down the prototype chain.

So if a method is called on an object and it's not found on that object, it goes up the prototype chain and checks the object's prototype. It will keep looking up the chain until it's found.

You can add properties and functions to a previously defined object type by using the prototype property. This defines a property that is shared by all objects of the specified type, rather than by just one instance of the object because it's added to the prototype chain.

Example:
Objects dog3 and dog4 don't have a speak() method so this throws an error:

```
dog3.speak();
dog4.speak();
```

We can add a speak() method to their prototype chain so when it doesn't find that method in their object it goes up the prototype chain and finds it.

```
Dog.prototype.speak=function(){
  console.log(this.name + " says WOOF");
};
```

```
dog3.speak();
dog4.speak();
```

If you're familiar with other OOP languages this might be sounding a lot like the functionality that classes usually provide. Well, traditional JavaScript doesn't have classes, not really. Functions, objects, and prototypes have basically filled the role of classes.
What do I mean by "traditional JavaScript"? Well for that we need to take a little detour …

Detour into JavaScript History
JavaScript was created in 1995 by Brendan Ike at Netscape. It was originally going to be called LiveScript, but as an attempt to capitalize on the popularity of Sun Microsystem's Java language, it was renamed to JavaScript. Other than those 4 letters, they are not related at all. It was also adopted by Microsoft as JScript. With different versions of the language there was need to standardize. ECMA (European Computer Manufacturer Association) is the governing body that produces the ECMAScript specifications. JavaScript implements ECMAScript in browsers. ECMAScript is the official name for JavaScript. A new name became necessary because there is a trademark on JavaScript (held originally by Sun, now by Oracle). For common usage, these rules apply:

- JavaScript means the programming language.
- ECMAScript is the name used for the language specification.
- ES is short for ECMAScript

1995: JavaScript created
1997: ECMAScript 1 - first version of the JavaScript language standard
1998: ECMAScript 2 – minor changes
1999: ECMAScript 3 - came out very quickly introducing many features. Used 2000-2010
2008: ECMAScript 4 – very ambitious and was eventually abandoned. A compromise that was less ambitious led to ECMAScript 5
2009: ECMAScript 5 (ES 5)- This is the version of ECMAScript that has been most widely used in the past
New versions of ECMAScript are now released yearly.
2015: ECMAScript 6 (ECMAScript 2015 = ES6) – supported by major desktop browsers, using in frameworks like React. Since it had a ton of new features this is the version now most widely used in new development.
Later versions have good support in major desktop browsers
2016: ECMAScript 2016 (ECMAScript 7/ES 7)
2017: ECMAScript 2017 (ECMAScript 8/ES 8)
2018: ECMAScript 2018 (ECMAScript 9/ES 9)
2019: ECMAScript 2018 (ECMAScript 10/ES 10)
2020: ECMAScript 2018 (ECMAScript 11/ES 11)

Takes a few years for browsers to support each new version fully.
https://kangax.github.io/compat-table/es6/
http://kangax.github.io/compat-table/es2016plus/
There are also transpilers like Babel that can convert ES6 code back to ES5 so developers can use the latest JavaScript features and not worry about backwards compatibility.

- Notice that repl.it is using Babel

Classes
ES6 introduced the Class keyword, but underneath the hood it's really still functions and prototypes. JavaScript classes provide a much simpler and clearer syntax to create objects and deal with inheritance.

- Use the keyword 'Class' to declare a class with the body in {}
- Must be declared before it's referenced, or it will throw a reference error
- Methods in classes don't need the keyword 'function'
- Classes can also be assigned as expressions, named or unnamed

Create a constructor method to initialize objects of the class type.

- Constructor methods use the keyword constructor
- There can only be one constructor method per class.

https://repl.it/@aileenjp/ES6

Example:

```
class Animal{
  constructor(name, age){
    this.name=name;
    this.age=age;
  }
  greeting(){
    console.log("Hi, my name is " + this.name + " and I'm " +
this.age + " years old");
  }
}

var pet=new Animal("Jake", 5);
var pet2=new Animal("Samantha", 9);

pet.greeting();
pet2.greeting();
```

Inheritance
You can extend a class to implement inheritance

- A class can only inherit from one class (multiple inheritance is not supported)
- To create a child class use the keyword 'extends' when creating it
- To call functions in an object's parent class use the keyword 'super'
- Call the parent's constructor with super()
- If there is a constructor present in the sub-class, it needs to first call super() before using "this".

Example:

```
class Dog extends Animal{
  speak(){
    console.log(this.name + " barks");
```

```
  }
}
var dog=new Dog();
console.log(dog.name);
dog.name="Cindy";
dog.age=15;
dog.speak();
```

Since pet is of type Animal it does not recognize the speak method.

```
//pet.speak();

class Cat extends Animal{
  constructor(name, age, breed){
    super(name, age);
    this.breed=breed;
  }
  speak(){
    console.log(this.name + " meows");
  }
    greeting(){
    console.log("Hi, my name is " + this.name + " and I'm an " +
this.age + " year old " + this.breed);
  }
}

var cat=new Cat("Pandora", 12, "Tabby");
console.log(cat.name);
cat.speak();
cat.greeting();
dog.greeting();
```

Variable Scope

In ES5, you declare variables via `var`. These variables are function-scoped, their scopes are the innermost enclosing function.

ES6 introduced block-scoped variables.

const – constant, can't be changed once it's defined. Your script will crash if you try to change a constant

let – block scope variable (loop, if, closures, nested functions).

var has a scope of an entire function

Example:

Change pet and pet2 to be `const` since they don't change.

Update class Cat

```
  old(){
    var oldAge=10;
    if (this.age>oldAge){
      var oldAge=15;
      console.log(oldAge);
```

```
    }
    console.log(oldAge);
  }
cat.old();
```

Both console.log statements print 15 because oldAge is the same variable since variables have function scope.
Change `oldAge` in the if block to `let`.(you can change both too)
Now the first prints 15, the second 10 because let has block scope, and the if statement is a block, so they are treated as two different variables.

Template strings and literals
Template strings make it easier to create more complex strings with dynamic content.
String "Hello " + name
Template string `Hello ${name} `
(name is a variable)
Template literals can span multiple lines

- Template string go in back ticks `
- No need for the + sign
- Preserves spaces, white space, indentation, etc
- Wrap variables in ${ }

Example:
Change greeting()

```
console.log(`Hi, my name is ${this.name} and I'm an ${this.age}
year old ${this.breed}`);
```

Arrow Functions
ES6 introduces arrow functions

- more concise
- allow for implicit returns (no return keyword needed)
- keeps the keyword "this" in scope

Example:
So back in our very first JavaScript object we had a variable my dog with two functions. Let's add a third called growUp.

```
var mydog = {
  name: 'Hazel',
  age: 1,
  bark: function(){
    console.log("WOOF");
  },
  speak: function(){
    console.log("Hi, my name is " + this.name);
  },
  growUp: function (){
    setInterval(function(){
```

```
    this.age++;
    console.log(this.age);
    }, 1000);
  }
};
mydog.age;
mydog.growUp();
```

You'll notice you're getting NaN inside the setInterval function. That's because 'this' is out of scope.

- Each new function defines its own this, such as in nested functions, and often you wanted to keep the scope of this.
    - To keep the scope of 'this', you need to bind it manually using .bind(this) so it doesn't fall out of scope. You must do this for each method.
    - A *binding* is an entry in an environment, storage space for a variable.

In ECMAScript5 you would need to bind this.

```
  growUp: function (){
    setInterval(function(){
    this.age++;
    console.log(this.age);
    }.bind(this), 1000);
  }
```

Arrow functions (also called fat arrow functions) => shorten the syntax for functions and let you create one line mini-functions

- remove the 'function' keyword and the function name
- parentheses are optional if there's one parameter, required if no parameters or multiple parameters
- use the arrow for the body of the function, curly brackets are not required if there's only one instruction
- no return keyword is needed

Arrow functions don't bound the 'this' keyword so it keeps 'this' in scope

- An arrow function does not create its own `this`, the `this` value of the enclosing execution context is used.

In ES6 we can simplify the function syntax and use an arrow function to handle the scope of 'this'.

```
var mydog = {
  name: 'Hazel',
  age: 1,
  bark (){
    console.log ("WOOF");
  },
  speak (){
```

```
    console.log ("Hi, my name is " + this.name);
  },
  growUp (){
    setInterval(()=>{
    this.age++;
    console.log(this.age);
    }, 1000);
  }
}

mydog.age;
mydog.speak();
mydog.growUp();
```

And now you can see age incrementing.

Arrays
Prior to ES5 to iterate over data structures such as arrays, strings, Maps, Sets and others you used the traditional for loop.
ES5 introduced forEach() with more concise syntax but you couldn't break from it.
ES6 introduced the for-of loop which is both concise and you can break from it.

Example:
ES5:

```
var scores = []; //array of grades
var num; //number of grades
num = prompt("How many grades do you have?");
for (i=0; i<num; i++){
  scores[i]=prompt("Enter grade #" + parseInt(i+1));
}
console.log("Your grades are:");
var total=0;
scores.forEach(function(score, index){
  console.log("#" + parseInt(index+1) + " " + score);
  total=total+parseInt(score);
});
console.log("Your average is " + total/num);
```

ES6 for-of (note the switched order of parameters):

```
for(const score of scores){
  console.log(score);
  total=total+parseInt(score);
}
console.log("Your average is " + total/num);
```

If you want both the index and the value:

```
for(const[index, score] of scores.entries()){
  console.log("#" + parseInt(index+1) + " " + score);
  total=total+parseInt(score);
}
console.log("Your average is " + total/num);
```

There are many useful methods available on arrays, we'll just look at two.
Let's say you have an array of objects and you want access to just parts of the objects in the array.

Example:

```
var pets = [
  {name: "Hazel", age: 2 },
  {name: "Casey", age: 4 },
  {name: "Nikki", age: 8 },
  {name: "Cole", age: 12 },
  {name: "Gizmo", age: 1 }
];
console.log(pets);
```

Point out the syntax:

- [ ] indicates array
- { } is a Javascript object

How would I just access the data of the third pet?

```
console.log(pets[2]);
```

And how would I access the third pet's name?

```
console.log(pets[2].age);
```

Now let's create a new array with only the pet names.
Using forEach()

```
var petNames = [];
pets.forEach(function (pet) {
  petNames.push(pet.name);
});
console.log(petNames);
```

Using for-of

```
for(const pet of pets){
  petNames.push(pet.name);
```

```
}
console.log(petNames);
```

The map() function maps an array into a new array. It takes two parameters, a callback and an optional context. In this example we just have the callback. The callback function defines the mapping from the current array to the new one. The callback runs for each value in the array and returns each new value in the resulting array. The resulting array will always be the same length as the original array.

```
var petNames = pets.map(function (pet) {
  return pet.name
});
console.log(petNames);
```

Simplifying map() with an arrow function:

```
var petNames = pets.map(pet => pet.name);
console.log(petNames);
```

You can tell the result is an array because of the [ ]

Now let's say you just want part of the array.

The slice() method returns a copy of a portion of an array into a new array object selected from begin to end (end is not included). The original array will not be modified.

```
var currentPet = pets.slice(2,3);
console.log(currentPet);
```

currentPet is an array with one object in it: structure [{}]

```
console.log(currentPet[0]);
```

use [0] to get the first item in the array which is an object: structure {}

```
console.log(currentPet[0].name);
```

use [0].name to get the name property from the first item in the array

There are many more methods available for collections as well.