JavaScript for App Development

By Mark Lassoff, Founder Framework Television

ES6 Classes

After completing this section you will:

- ☐ How to create an ES6 style class
- ☐ How to Create instances from a class
- ☐ Create a Class Constructor
- ☐ Design a Subclass (aka a child class)
- ☐ How to pass values to a superconstructor
- ☐ How to create instances of a subclass
- ☐ Override methods in a subclass
- ☐ Use setters and getters in your classes.

Introduction

With the ES6 standard, JavaScript classes have become more like classes in traditional object-oriented programming languages. A new method for implementing constructors, setters, and getters and formal subclassing will allow you to apply an object-oriented methodology to problems you're solving in Javascript.

Some new to programming feel that the full-blown object-oriented methodology is overkill for most small programming problems.

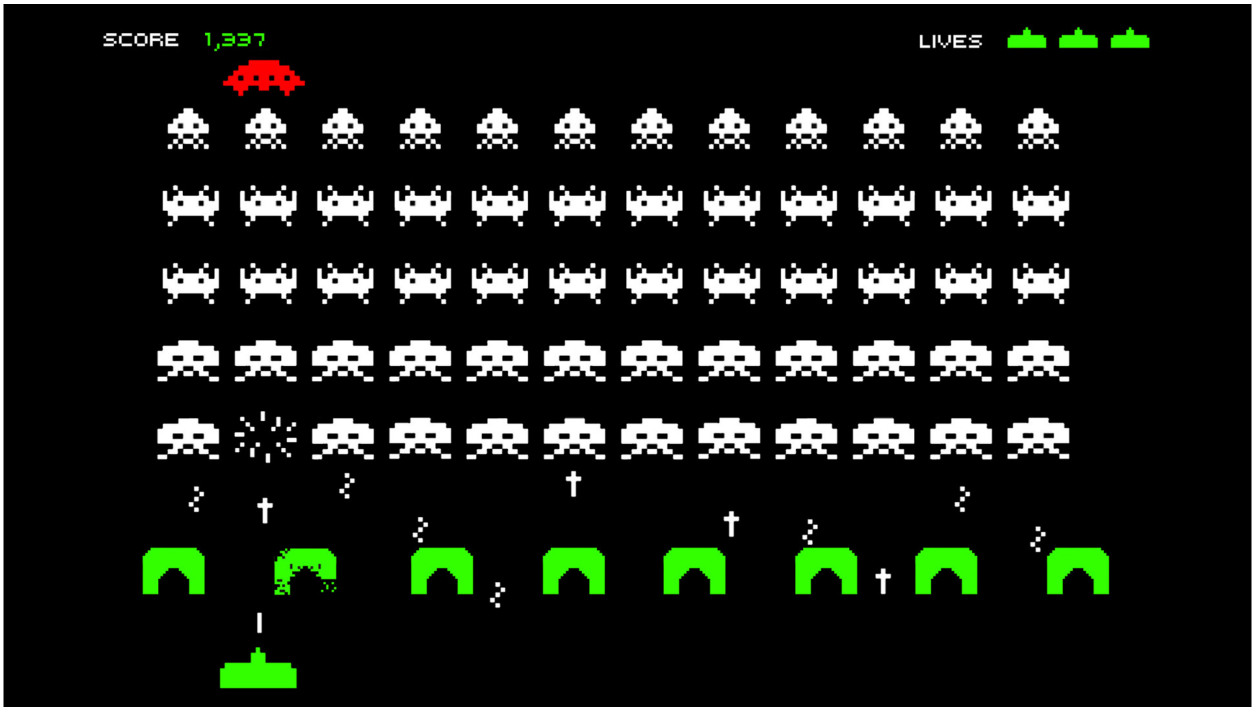
And, they're right.

Breaking your problem space into classes and objects is useful for solving complex computing problems with many entities interacting with each other in a system. However, for simpler problems JavasScript's more traditional event-driven model should be just fine.

Creating an ES6 Style JavaScript Class

The foundations of classes in ES6 are the same as always. A class should be designed to emulate real-world objects that are part of your problem space. Classes provide a blueprint for those elements.

You might find it helpful to imagine the early video game Space Invaders.



In the game space invaders, you have a player, enemies, barriers, , and bombs. Each of these can be represented by a class outlining properties and methods contained within.

It’s helpful to think of properties as adjectives. Properties describe an object. Methods can be thought of as verbs or what a class is capable of doing.

An enemy class in space invaders may be described by the following pseudocode.

```
enemy{
  Properties:     xPosition,
            yPosition,
            image,
            status
  Methods:        moveLeft(),
            moveRight(),
            spawn(),
            die()
}
```

The class outlined above simply provides a blueprint for each instance of an enemy that exists in the game in any one time. (It looks like you can have five rows of twelve enemies.) . Each one would contain a different value for its position along the x-axis and y-axis as well as values for image and status.

Each enemy is capable of moving left or right, spawning and, of course, dying when hit with a player’s missile.

Creating the Class Constructor

Let’s create our own class `Employee`. The class represents an employee in a simple company human resources system.

```
class Employee {

//constructor
  constructor(firstName, lastName, social, jobTitle, salary ){
    this._firstName = firstName;
    this._lastName = lastName;
    this._social = social;
    this._jobTitle = jobTitle;
    this._salary = salary;
    this._active = true;        //default value;
  }
}
```

You should save this file as 'Employee.js'.
The class constructor is a function that runs each time a new object of the `Employee` class is created. The job of the constructor is to provide an initial value for the object properties.

In our example the values for `firstName`, `lastName`, `social`, `jobTitle`, and `salary` are passed in from the code creating an instance of the object. The values are assigned to these variables scoped only for the constructor.

Inside the constructor function, you can see the assignment statements for each of these values. On the left side of the assignment operator are the instance variables. The keyword `this` is used to indicate that you are talking about *this* instance of the object: The instance currently being created.

You're probably wondering about the unusual use of the underscore character. `_`.

The reason it is used here is to disambiguate between the variables being passed into the constructor and the variables that will live in the instance. In this case, the instance variables which we'll access in the object will begin with the underscore character.

You've also noticed that the `_active` variable does not have a value passed in, but instead, is assigned a *default value* in the constructor. This is by design. Every Employee object should have an `_active` value of true, so, we make the assignment directly in the constructor.

Creating an Instance

The class code above provides us with enough to create an instance of `Employee`. An instance is created with the following code:

```
let scott = new Employee("Scott", "Wilson", "090-00-0000",
"accountant", 36.50);
```

Our instance called `scott` has initial values passed to the constructor as either strings or floating point numbers. We could create as many instances of the `Employee` class as desired at this point by using unique names to reference them.

Adding Class Methods

At this point, we've created our class, but it can't really do anything. We need to implement some methods to make our class functional. Let's add a method `info()` which provides information about the class instance and a method `fire()` which allows us to terminate the employee by setting the `active` variable to value to *false*.

```
//Methods
  fire(){
    this._active = false;
  }

  info(){
    if (this._active){
      const info = `${this._firstName} ${this._lastName}, ${this._
social}, ${this._jobTitle}`;
      return info;
    } else
    {
      const info = "Employee is not active";
      return info;
    }
  }
```

The `info()` method is using string templating (aka string interpolation covered earlier in the JavaScript for App Development Program.

Make sure you place these methods inside the class brackets.

Implementing class methods

We can implement the class methods with the instance we created. Create a second file and save it as an `.html` file.

```
<script src="Employee.js"></script>
<script>
  let scott = new Employee("Scott", "Wilson", "090-00-0000",
"accountant", 36.50);

  alert(scott.info());
  scott.fire();
  alert(scott.info());
</script>
```

The code above is straight forward. First we bring in the Employee.js script. Then we create our instance `scott` and pass the initial property values to the contructor. Now, with the instance created we fire the `info()` function.

The initial `if` statement in the function evaluates as `true` since the `_active` variable holds the `true` value. The rest of the `info()` function executes alerting out information about the instance.

Next we `fire()` poor `scott`.

the `fire()` method runs one line of code:

```
this._active = false;
```

With this change we call the `info()` method again which now evaluates the initial condition as `false` and runs the `else` bracket outputting the message `Employee is not active`.

Inheritance with Child Classes

Let's say that we need to also account for part-time employees in our application. Part-timers have all of the properties and methods of a full-time employee, but, we also need to track how many hours they are assigned to work per week.

We could create an entirely new class for part-time employees, however, that would require we duplicate a lot of code and doesn't allow us to establish the correct relationship between the two classes.

Since all part-time employees are employees, but not all employees are part-time employees a parent-child relationship exists. Therefore, it's appropriate, to create part-time employees as a child class of `Employee`.

We create the child class with the keyword `extends` as in the example below:

```
class PTEmployee extends Employee  {
   constructor(firstName, lastName, social, jobTitle, salary,
hoursPerWeek){
      super(firstName, lastName, social, jobTitle, salary,
hoursPerWeek);
      this._hoursPerWeek = hoursPerWeek;
   }
```

Our new class `PTEmployee` has a constructor, like it's parent, but, there's additional complexity. Inside the constructor, we use the keyword `super()` to send the values passed into the constructor to the parent.

> The parent class is also known as the superclass, hence, the keyword super().

After passing the values to the parent class, we deal with the properties unique to the child class. In this case, we establish our `_hoursPerWeek` property to set the number of hours worked per week worked by the part-time employee.

Note that both the properties and methods in the parent method are inherited by the child method. The `info()` and `fire()` methods now exist in the `PTEmployee` class as they've been inherited from the `Employee` class.

> The child class is often referred to as a subclass of the parent.

Instantiating the Child Class

Create an object from the child class is the same as creating the original employee object.

Let’s update our HTML file as follows:

```
<script src=”Person.js”></script>
<script>
   let scott = new Employee(“Scott”, “Wilson”, “090-00-0000”, 
“accountant”, 36.50);
   let bob = new PTEmployee(“Bob”, “Johnson”, “080-00-0000”, 
“driver”, 18, 16);

   alert(scott.info());
   alert(bob.info());
</script>
```

When you load this file into the browser, it seems to run just fine. The `info()` method runs in both the parent instance and the child instance. But there is a small issue. Did you catch it?

Overriding the Parent Implementation of Info()

When we initially created the info() method we didn’t account for the `_hoursPerWeek` property because it does not exist in the parent. We need the info() method to recognized and output a value for this property in the child.

We’ll accomplish this by overriding the `info()` method in the child. Essentially we’ll write the method again in the subclass, but, this time we’ll write it to accomplish changes in the subclass.

```
info() {
      if (this._active){
         const info = `${this._firstName} ${this._lastName}, ${this._
social}, ${this._jobTitle}, ${this._hoursPerWeek}`;
         return info;
     } else
     {
         const info = "Employee is not active";
         return info;
     }

   }
```

In this version, the `_hoursPerWeek` property is included.

If you again run your HTML file in the browser, you'll get the correct result for both the parent class and the child class.

Setters and Getters

Formal setters and getters are also new to ES6.

Formal setters and getters allow you to set the value of properties from outside of a class in a controlled way. For example, let's say you had a property representing the speed of a vehicle. The speed can't be a negative number so you could you a setter function to ensure that the value was always positive.

In our example, a salary would always be a positive number so we can create a setter:

```
set salary(newSalary){
    if (newSalary > 0){
        this._salary = newSalary;
    }
  }
```

As you can determine from the code, the salary will only be set if the value is greater than 0. Otherwise, nothing will happen to the salary property.

Getters in JavaScript allow access to retrieve property values in a class:

```
get salary(){
     return this._salary;
  }
```

It's good form to use setters and getters to prevent accidentally setting inappropriate values within a class.

Submit This:
The Vehicle Class

Using proper ES6 methods, create a class that represents a vehicle. Use the pseudo code below to create the class:

```
vehicle{
   Properties:       color (string),
                  direction (integer 0-359),
                  currentSpeed (integer),
                  topSpeed (integer),
                  engineStarted (boolean),
   Methods:          accelerate(),
                  brake(),
                  turnOn(),
                  turnOff(),
                  turnLeft(),
                  turnRight()
}
```

When coding your class ensure that the value of direction is only between 0-359 (representing a compass) in any instance.

Create two subclasses. Create a `bus` subclass that includes a property for `numberOfSeats` and creates an ambulance subclass that has methods to turn the siren on and off.

Use text output to the console to describe what is going on with the vehicle. For example, when the `accelerate()` method is used output something like: “Accelerating. Speed is now 20”.

Create a separate HTML file that instantiates each of your subclasses and allows you to drive them around by issuing methods.