CS 22A JavaScript for Programmers

Today (Day 11)

- Announcements
- Objects: Methods

Announcements

- Make sure you are **always** testing in **FirefoxDeveloperEdition** while so you can see error messages on the **console**
  - Don't use other browsers unless you are directed to do so because your work will be graded using Firefox**DeveloperEdition**
  - Can't figure out what's going on with your code? Use console.log().
  - Still can't figure out what's going on? Use the FirefoxDeveloperEdition **debugger**. There is a video on how to use it in the Day06 module.
  - Also the console is a great place to check on what objects exist and what functions work!

Objects: Adding Methods

- In JavaScript, a **method** is a *function* that is part of an **object**.
- When a method is called it can perform various tasks that you might want to execute *with* the properties of the *object*.

Objects: Adding Methods

Consider the **Car** object that we created with a constructor function earlier in the lecture. We might want to have a method to make a calculation of some sort, like a payment.

```
var getPayment = function () {
     var thePayment = 250;
     if(this.seats == "leather") {
          thePayment += 100;
     }
     else {
          thePayment += 50;
     }
     if(this.engine == "V-6") {
          thePayment += 150;
     }
     else {
          thePayment += 75;
     }
     if(this.soundSystem == "MP3 Player")
     {
          thePayment += 10;
     }
     else {
          thePayment += 35;
     }
};
```

Objects: Adding Methods

Well, the previous function is really long. It can be shortened by using the *conditional (ternary) operator* for each **if/else** statement here:

```
var getPayment = function() {
  var thePayment = 250;
  thePayment += (this.seats == "leather") ? 100 : 50;
  thePayment += (this.engine == "V-6") ? 150 : 75;
  thePayment += (this.soundSystem == "MP3 Player") ? 10 : 35;
  return thePayment;
};
```

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

Objects: Adding Methods

- After you have defined the function, you need to assign it to your Car objects within your object constructor function.
- Using your trusty Car object constructor function, you would assign it as shown in the following code:

```
var Car = function (seats, engine, soundSystem) {
  this.seats = seats;
  this.engine = engine;
  this.soundSystem = soundSystem;
  this.payment = getPayment; // <-- Notice no parentheses used!
};
```

- Notice that this example defines a method named **payment** that calls the **getPayment()** function from outside the constructor function.
- Also notice that when the function is called here, the *parentheses* are not used on the end of the function call.
- This is how your outside function becomes a method of the object (by assigning it the function rather than the result of the function).

Methods CONTINUED

In order to call the **payment()** method of the object, you need an instance of the Car object. If you add the three instances you made earlier, you will be able to do some things with your new method. So, add those instances to the code you already have:

```
var getPayment = function () {
     var thePayment = 250;
     thePayment += (this.seats == "leather") ? 100 : 50;
     thePayment += (this.engine == "V-6") ? 150 : 75;
     thePayment += (this.soundSystem == "MP3 Player") ? 10 : 35;
     return thePayment;
};
var Car = function (seats, engine, soundSystem) {
     this.seats = seats;
     this.engine = engine;
     this.soundSystem = soundSystem;
     this.payment = getPayment;
};
var myCar = new Car("leather","V-6","Radio with 6 CD Player");
var wifesCar = new Car("cloth","V-4","MP3 Player");
var customCar = new Car(myCar.seats, myCar.engine, wifesCar.SoundSystem);
```

Methods CONTINUED

Even better you can add the method to the constructor's prototype:

```
var getPayment = function () {
     var thePayment = 250;
     thePayment += (this.seats == "leather") ? 100 : 50;
     thePayment += (this.engine == "V-6") ? 150 : 75;
     thePayment += (this.soundSystem == "MP3 Player") ? 10 : 35;
     return thePayment;
};
var Car = function (seats, engine, soundSystem) {
     this.seats = seats;
     this.engine = engine;
     this.soundSystem = soundSystem;
};
Car.prototype.payment = getPayment;
var myCar = new Car("leather","V-6","Radio with 6 CD Player");
var wifesCar = new Car("cloth","V-4","MP3 Player");
var customCar = new Car(myCar.seats, myCar.engine, wifesCar.SoundSystem);
```

Methods

- Now you have the function that is used to create the **payment()** method of the Car object, the creation of the **payment()** method within the Car object constructor function, and three instances of the Car object.
- To find the monthly payments for **myCar**, you would call the **payment()** method using the following syntax:

```
var myCarPayments = myCar.payment();
```

- The value of the **myCarPayments** *variable* will be the value returned from the **payment()** method, which is what is returned from the **getPayment()** *function* when run with the values used for the **myCar** *instance* of the **Car** object.
- Since the *seats* are "*leather*", **thePayment** is increased by **100** (if they were *cloth* it would have been **50**).
- Since the *engine* is a "*V-6*", **thePayment** is increased by **150**.
- Finally, since the *soundSystem* property has a value of "*Radio with 6 CD Player*", the **thePayment** variable is increased by **35**.
- This gives you a payment of **250** (initial value) + **100** (*leather* seats) + **150** (*V-6* engine) +**35** (*non*-MP3 Player), which turns out to be **535**.

Methods CONTINUED

- Using the **payment()** method, you could now write a script to display the payment amount for each type of Car in the browser so the viewer can decide what type of Car to buy.
- You would just expand on your previous code to include in the body of the page some **$("#content").append()** commands that use the values returned from the **payment()** method.
- See the example spread across the next three slides to see the whole thing put together:

Example Part 1

```
var getPayment = function () {
     var thePayment = 250;
     thePayment += (this.seats == "leather") ? 100 : 50;
     thePayment += (this.engine == "V-6") ? 150 : 75;
     thePayment += (this.soundSystem == "MP3 Player") ? 10 : 35;
     return thePayment;
};
var Car = function (seats, engine, soundSystem) {
     this.seats = seats;
     this.engine = engine;
     this.soundSystem = soundSystem;
};
Car.prototype.payment = getPayment;
```

Example Part 2

```
var myCar = new Car("leather","V-6","Radio with 6 CD Player");
var wifesCar = new Car("cloth","V-4","MP3 Player");
var customCar = new Car(myCar.seats, myCar.engine, wifesCar.soundSystem);

var myCarPayment = myCar.payment();
var wifesCarPayment = wifesCar.payment();
var customCarPayment = customCar.payment();
```

Example Part 3

```
$("#content").append("<h2>The information on the cars you requested:</h2>");
$("#content").append("<strong>My Car: </strong>");
$("#content").append(myCar.seats + ", " + myCar.engine + ", " + myCar.soundSystem);
$("#content").append("<br>");
$("#content").append("<strong>Payments:</strong> $" + myCarPayment);
$("#content").append("<p>");
$("#content").append("<strong >My Wife's Car: </strong>");
$("#content").append(wifesCar.seats + ", " + wifesCar.engine + ", " + wifesCar.soundSystem);
$("#content").append("<br>");
$("#content").append("<strong>Payments:</strong> $" + wifesCarPayment);
$("#content").append("</p>");
$("#content").append("<p>");
$("#content").append("<strong>My Dream Car: </strong>");
$("#content").append(customCar.seats + ", " + customCar.engine + ", " );
$("#content").append(customCar.soundSystem);
$("#content").append("<br>");
$("#content").append("<strong>Payments:</strong> $" + customCarPayment);
$("#content").append("</p>");
```

Object Manipulation Statements

JavaScript allows you to use the **for-in** loop to help you *manipulate* objects and the **with** statement to access particular objects more easily.

The for-in Loop

The for-in loop allows you to cycle through the *properties* of an object to display them or to manipulate their values.

The following code shows the structure of a for-in loop:

```
for (var variableName in objectName) {
     // JavaScript statements
}
```

Object Manipulation Statements CONTINUED

Suppose you wanted to cycle through the properties of a **myCar** *instance* of a **Car** *object* in order to display the values of each property on the page. The **for-in** loop allows you to do this without the need to type each property name, as in this code:

```
var Car = function (seats, engine, soundSystem) {
     this.seats = seats;
     this.engine = engine;
     this.soundSystem = soundSystem;
}

var myCar = new Car("leather","V-6","Radio with 6 CD Player");
for (var propName in myCar) {
               $("#content").append(myCar[propName] + "<br />");
}
```

You will notice that the **myCar[propName]** part of the script is unfamiliar. The **for-in** loop uses an array to store the *property* values, which calls for this syntax.

Summary: What Does an Object Look Like?

In JavaScript, an object can be created in a couple of ways.

A primary way to create an object is by the **new** keyword, like so:

```
var myObject = new ObjectConstructor();
```

Objects can also be created by using two **curly braces**, like so:

```
var myObject = {};
```

This is called an object literal.

Object Properties CONTINUED

You can also nest objects, as shown in this example:

```
var guitar = {
     ”Front Color": "red",
     "strings": {
          "number": 6,
          "smallGauge": 9,
          "largeGauge": 42
          },
     "type": "electric"
};
```

In this example, a nested object is created and contains properties about the strings of the guitar, including the number of strings and their gauge.

Object Properties CONTINUED

Properties are accessed by using **dot notation** or by including the **property name in brackets**.

Here's an example of *dot notation*:

```
guitar.strings.smallGauge; // 9
```

Here's an example of *property name in brackets*:

```
guitar[”Front Color"]; //red
guitar["strings"]["smallGauge"]; //9
```

It's usually preferable from a readability standpoint to use dot notation when possible, but that's really a matter of developer preference and his-or-her coding standard.

Object Methods

In addition to having properties, many objects *do* things. With my assistance, the guitar plays individual notes on a given string as well as chords with multiple strings.

In JavaScript, you can create a *function*, store it as a property, and then call or invoke that function.

These functions are essentially just like the functions you learned about earlier, except they're called methods when used with objects.

Methods are declared, they can accept arguments just like functions, and they can return values.

For the purposes of this class, methods are just like functions that are *added or attached to an object*.

Object Methods CONTINUED

Building on the guitar object, here's a method to play a string:

```
var guitar = {
     "color": "red",
     "strings": 6,
     "type": "electric",
     "playString": function (stringName) {
          var playIt = "I played the " + stringName + " string";
          return playIt;
     } //end function playString
};
```

In this example a method called **playString** is declared and accepts *one* argument.

That variable, called **stringName**, is used to indicate the string to play on the guitar and is then used to create a message indicating that the guitar string was played.

Object Methods CONTINUED

If you preferred, you could have created this method later then add it to the guitar object by using dot notation.

Assuming that the guitar object has been created already, adding the method looks like this:

```
guitar.playString = function(stringName) {
          var playIt = "I played the " + stringName + " string";
          return playIt;
};
```

the this Keyword

One of the more powerful aspects of object-oriented programming is the use of the **this** keyword.

The **this** keyword provides *self-referential* access to an object's *properties* and *methods*.

You can use this to perform advanced operations within an object's methods.

For example, using this, you can get and set an object's properties in a consistent manner (sometimes known as *getters* and *setters*).

the this Keyword CONTINUED

Here's an example with the guitar object again:

```
var guitar = {
     "color": "red",
     "strings": 6,
     "type": "electric",
     "playString": function (stringName) {
          var playIt = "I played the " + stringName + " string";
          return playIt;
     },
     "getColor": function() {
          return this.color;
     },
     "setColor": function(colorName) {
          this.color = colorName;
     }
};

alert(guitar.getColor()); // This alert will show 'red'
guitar.setColor("blue"); // This method changes the color to "blue"
alert(guitar.getColor()); // This alert will show 'blue'
```

the this Keyword CONTINUED

The two additions to the guitar object are the getter and setter methods, **getColor()** and **setColor(),** respectively.

The **getColor()** method looks like this:

```
"getColor": function() {
     return this.color;
}
```

This method merely returns the color property as it is currently set.

The **setColor()** method looks like this:

```
"setColor": function(colorName) {
     this.color = colorName;
}
```

This method sets the color as it is passed into the method call.