JavaScript Objects

In this session we will look at JavaScript ***objects***. We have briefly looked at objects when we looked at JSON parsing in JavaScript but today we will look at them in more detail.

Introduction to Objects

In programming we frequently come across the concept of an ***object***. An object is a representation of a real-world entity, such as a person, a cat, or a web page element, in programming code. Objects have:

- ***properties***, also known as ***attributes***, which ***describe*** an object. For example a cat object might have ***name***, ***age*** and ***weight*** properties.
- ***methods***, which are things we can ***do*** with an object. For example, a cat object might have methods to ***walk***, ***feed*** and ***meow***. Methods are very similar to ***functions***, which you've already seen, but they work with a specific object.

JavaScript Objects Built In To The Browser Environment

Remember that you can use the DOM from within JavaScript to manipulate page elements. This involves the use of JavaScript objects which are built into the web browser environment. They are available whenever you use JavaScript with a standards-compliant web browser. To go through an example:

```
var elem = document.getElementById("mydiv");
elem.style.color = 'red';
```

This code shows the use of some in-built JavaScript objects. ***document*** is an object representing the document, i.e. the web page. We call a ***method*** of the document object called ***getElementById()***, which takes an element ID as an argument and returns the corresponding element as another object. In this code, we assign the element object to the variable ***elem*** - from this point onwards, ***elem*** will represent our page element object and we will be able to manipulate its ***properties***. Here, we set the ***color*** property of the element's ***style*** (and the style itself is an object) to red. So it can be seen that:

- The ***document*** object has a ***getElementById()*** method which takes an element ID as an argument and returns a page element as an object;
- An element object has a ***style*** property which is itself an object, and that ***style*** object has a number of its own properties representing different aspects of its style, e.g. its text colour, background colour, position, font, and so on.

Defining our own objects

In many cases it is sufficient to merely work with inbuilt JavaScript objects. However, if you are building the more complex HTML5 applications increasingly commonly found today, you will probably find yourself needing to create your own objects. Here is an example of how to do this.

```
<html>
<head>
<script type='text/javascript'>
function example1()
{
    var cat = { name: "Tiddles",
                age: 10,
                weight: 10 };

    alert(cat.name + " " + cat.age + " " + cat.weight);
}
</script>
</head>
<body>
<h1>Objects Example</h1>
<p>
<input type="button" value="test" onclick="example1()"
</p>
</body>
</html>
```

Note how the properties are contained within curly brackets, and each property consists of its name (e.g. ***age***) and its value (e.g. ***10***), separated by a colon. If you have encountered associative arrays or Maps in other languages, you can probably appreciate that a JavaScript object is fairly similar in the sense that it has a series of ***keys***, i.e. properties, and values. These are known as ***key-value pairs***. We could then add this code to the ***example1()*** function to create a second cat:

```
var cat2 = { name: "Tom",
             age: 5};

alert(cat2.name + " " + cat2.age);
```

Note that the second cat does not have a weight. If you have had previous experience with object-orientated languages such as Java or C++, you will probably notice a difference here. In JavaScript, there are no true ***classes***

(i.e. blueprints from which we create objects of a given type, such as from a Cat class). Instead, we simply create a plain ***Object*** and give it arbitrary properties (and methods). We can dynamically add new properties (and methods) to an object at any time; this is another difference from Java or C++.

Methods

So far we have only dealt with object properties, which ***describe*** the object concerned. Remember, however, we can also ***do*** things to objects, such as change their state. We use ***methods*** to do this. A method represents an action on an object: it will typically change its state in some way (such as increase the weight for a cat).

Anonymous functions

Before showing how methods are done in JavaScript, it is helpful to revise the concept of the ***anonymous function***, first encountered when we looked at AJAX. An anonymous function is a function without a name. We used a form of these when doing AJAX callbacks (arrow functions) but we can also have regular functions as anonymous functions. In objects, the ***methods*** are ***anonymous functions which are properties of the object***.

Setting up methods

On a basic level, we set up methods in a very similar way to properties. As we saw above, each method can be considered a property, in which the value of the property is an anonymous function. For example:

```
<html>
<head>
<script type='text/javascript'>
function objectsexample()
{
    var cat = {
        name: "Tiddles",
        age: 10,
        weight: 10,

        makeNoise: function()
        {
            alert("Meow!");
        },

        walk: function()
        {
            // "--" reduces the weight by 1
```

```
            this.weight--;
        },

        eat: function()
        {
            // "++" increases the weight by 1
            this.weight++;
        }

    };
}

</script>
</head>
<body>
<h1>Objects Example</h1>
<p>
<input type="button" value="test" onclick="objectsexamp
</p>
</body>
</html>
```

Note the similar syntax in setting up the ***makeNoise()***, ***walk()*** and ***eat()*** methods compared to the properties. The difference is that rather than setting the properties to values (such as "Tiddles" or 10) we set them to ***functions*** which perform a particular task, such as decrease or increase the weight. This makes them ***methods***. So in JavaScript we can say that:

```
A method is a property which is a function.
```

Note the use of ***this*** in the method. ***this*** refers to the object which we are operating on. So, the walk() method is reducing the weight of the current cat object by 1, and the eat method is increasing it by 1.

Calling methods

The previous code merely defines how the cat object works. It specifies how cats make a noise, eat or walk ***but does not actually make the cat make a noise, eat or walk***. To do this we need to ***call*** the methods, as oppose to merely defining them as we did above. To call a method we use the object name, followed by a dot, followed by the method name. The code below creates a cat object, defines its methods ***and then*** calls the ***eat()*** and ***makeNoise()*** methods:

```
<html>
<head>
<script type='text/javascript'>
function objectsexample()
{
    var cat = {
        name: "Tiddles",
        age: 10,
        weight: 10,

        makeNoise: function()
        {
            alert("Meow!");
        },

        walk: function()
        {
            // "--" reduces the weight by 1
            this.weight--;
        },

        eat: function()
        {
            // "++" increases the weight by 1
            this.weight++;
        }

    };

    alert(cat.weight);
    cat.eat();
    alert(cat.weight);
    cat.eat();
    alert(cat.weight);
    cat.makeNoise();
}
</script>
</head>
<body>
<h1>Objects Example</h1>
<p>
<input type="button" value="test" onclick="objectsexamp
</p>
</body>
```

```
</html>
```

Note carefully the difference between:

```
eat : function() { this.weight++; }
```

and

```
cat.eat();
```

The former ***specifies how the eat() method works***. It does ***not*** actually call the method, in other words, it does not make the cat eat. It just stores the method in memory, ready for later use. The latter, by contrast, actually calls the method and makes the cat eat, so that its weight goes up by one. Note also how we can call the same method more than once.

The toString() method

JavaScript has a special ***toString()*** method which can be used to ***return*** a string representation of that object. For a cat, this could be its name, age and weight. Here is an example:

```
var cat3 =
    { name: "Tigger",
      age: 8,
      weight: 7,
      makeNoise: function() { alert("Meow!") },
      walk: function() { this.weight--; },
      eat: function() { this.weight++; },
      toString: function() { return this.name +" " + tl
    };
```

The advantage of ***toString()*** is that it allows you to easily display an object. If we supply an object as an argument to ***alert()***, for instance, it would use ***toString()*** to determine how to display the object. So if we did the following:

```
alert(cat3);
```

we would get:

```
Tigger 8 7
```

Exercise

1. Create a web page containing ***only*** a button which links to a JavaScript function called ***objectstest()***. Use this code:

```
<html>
<head>
</head>
<body>
<h1>Objects Exercise</h1>
<p>
<input type="button" value="test" onclick="objectst
</p>
</body>
</html>
```

2. Write the ***objectstest()*** function. In the function, create a ***car*** object. It should have properties for make, model, engine capacity (cc), top speed, and current speed (mph or km/h - doesn't really matter). Set them to appropriate values (current speed should initially be zero)
3. Add ***accelerate()*** and ***decelerate()*** methods. These should increase and decrease the speed by 5 units, respectively. ***decelerate()*** should ideally check to make sure that the speed doesn't go below 0 and ***accelerate()*** should ideally check to make sure that the speed doesn't go above the top speed.
4. Add a ***toString()*** method to return a string representation of the car.
5. Try displaying the car object, accelerating it, displaying it again, decelerating it, and displaying it once more.
6. Create a ***second*** car object which works in exactly the same way as the first, but with different properties. Can you see the problem? Next time we will see how to solve this! (***If you feel like it, read ahead on this subject and try to fix the problem!***)

If you finish

If you have started it already, continue to work on the DOM exercise from last time. If not, catch up on unfinished work.