**The Basics of Object-Oriented JavaScript**

Over recent years, JavaScript has increasingly gained popularity, partly due to libraries that are developed to make JavaScript apps/effects easier to create for those who may not have fully grasped the core language yet.

While in the past it was a common argument that JavaScript was a basic language and was very 'slap dash' with no real foundation; this is no longer the case, especially with the introduction of high scale web applications and 'adaptations' such as JSON (JavaScript Object Notation).

JavaScript can have all that an Object-Orientated language has to offer, albeit with some extra effort outside of the scope of this article.

**Let's Create an Object**

|  |  |
| --- | --- |
| 1  2  3 | function myObject(){    }; |

Congratulations, you just created an object. There are two ways to  
create a JavaScript object: they are 'Constructor functions' and  
'Literal notation'. The one above is a Constructor function,  
I'll explain what the difference is shortly, but before I do, here  
is what an Object definition looks like using literal notation.

|  |  |
| --- | --- |
| 1  2  3 | var myObject = {    }; |

Literal is a preferred option for name spacing so that your JavaScript  
code doesn't interfere (or vice versa) with other scripts running on the  
page and also if you are using this object as a single object and not requiring  
more than one instance of the object, whereas Constructor function type  
notation is preferred if you need to do some initial work before the object  
is created or require multiple instances of the object where each instance  
can be changed during the lifetime of the script. Let's continue to build  
on both our objects simultaneously so we can observe what the differences are.

**Defining Methods and Properties**

**Constructor version:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | function myObject(){      this.iAm = 'an object';      this.whatAmI = function(){          alert('I am ' + this.iAm);      };  }; |

**Literal version:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | var myObject = {      iAm : 'an object',      whatAmI : function(){          alert('I am ' + this.iAm);      }  } |

For each of the objects we have created a property 'iAm' which contains a  
string value that is used in our objects method 'whatAmI' which alerts a message.

*Properties are variables created inside an object and methods are functions created inside an object.*

Now is probably as good a time as any to explain how to use properties and  
methods (although you would already have done so if you are familiar with a library).

To use a property first you type what object it belongs to - so in this case it's myObject -  
and then to reference its internal properties, you put a full stop and then the name of the  
property so it will eventually look like myObject.iAm (this will return 'an object').

For methods, it is the same except to execute the method, as with any function, you must  
put parenthesis after it; otherwise you will just be returning a reference to the function  
and not what the function actually returns. So it will look like myObject.whatAmI()  
(this will alert 'I am an object').

**Now for the differences:**

* The constructor object has its properties and methods defined with the  
  keyword 'this' in front of it, whereas the literal version does not.
* In the constructor object the properties/methods have their 'values'  
  defined after an equal sign '=' whereas in the literal version, they are  
  defined after a colon ':'.
* The constructor function can have (optional) semi-colons ';' at the  
  end of each property/method declaration whereas in the literal version  
  if you have more than one property or method, they MUST be separated with  
  a comma ',', and they CANNOT have semi-colons after them, otherwise JavaScript will return an error.

There is also a difference between the way these two types of object declarations are used.

To use a literally notated object, you simply use it by referencing its variable name,  
so wherever it is required you call it by typing;

|  |  |
| --- | --- |
| 1 | myObject.whatAmI(); |

With constructor functions you need to instantiate (create a new instance of)  
the object first; you do this by typing;

|  |  |
| --- | --- |
| 1  2 | var myNewObject = new myObject();  myNewObject.whatAmI(); |

**Using a Constructor Function.**

Let's use our previous constructor function and build upon it so it performs some basic  
(but dynamic) operations when we instantiate it.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | function myObject(){      this.iAm = 'an object';      this.whatAmI = function(){          alert('I am ' + this.iAm);      };  }; |

Just like any JavaScript function, we can use arguments with our constructor function;

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | function myObject(what){      this.iAm = what;      this.whatAmI = function(language){          alert('I am ' + this.iAm + ' of the ' + language + ' language');      };  }; |

Now let's instantiate our object and call its whatAmI method, filling in the required  
fields as we do so.

|  |  |
| --- | --- |
| 1  2 | var myNewObject = new myObject('an object');  myNewObject.whatAmI('JavaScript'); |

This will alert 'I am an object of the JavaScript language.'

**To Instantiate or not to Instantiate**

I mentioned earlier about the differences between Object Constructors and Object Literals and that  
when a change is made to an Object Literal it affects that object across the entire script, whereas when  
a Constructor function is instantiated and then a change is made to that instance, it won't affect any  
other instances of that object. Let's try an example;

First we will create an Object literal;

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12 | var myObjectLiteral = {      myProperty : 'this is a property'  }    //alert current myProperty  alert(myObjectLiteral.myProperty); //this will alert 'this is a property'    //change myProperty  myObjectLiteral.myProperty = 'this is a new property';    //alert current myProperty  alert(myObjectLiteral.myProperty); //this will alert 'this is a new property', as expected |

Even if you create a new variable and point it towards the object, it will have the same effect.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15 | var myObjectLiteral = {      myProperty : 'this is a property'  }    //alert current myProperty  alert(myObjectLiteral.myProperty); //this will alert 'this is a property'    //define new variable with object as value  var sameObject = myObjectLiteral;    //change myProperty  myObjectLiteral.myProperty = 'this is a new property';    //alert current myProperty  alert(sameObject.myProperty); //this will still alert 'this is a new property' |

Now let's try a similar exercise with a Constructor function.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16 | //this is one other way of creating a Constructor function  var myObjectConstructor = function(){      this.myProperty = 'this is a property'  }    //instantiate our Constructor  var constructorOne = new myObjectConstructor();    //instantiate a second instance of our Constructor  var constructorTwo = new myObjectConstructor();    //alert current myProperty of constructorOne instance  alert(constructorOne.myProperty); //this will alert 'this is a property'     //alert current myProperty of constructorTwo instance  alert(constructorTwo.myProperty); //this will alert 'this is a property' |

So as expected, both return the correct value, but let's change the myProperty for one of the instances.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19 | //this is one other way of creating a Constructor function  var myObjectConstructor = function(){      this.myProperty = 'this is a property'  }    //instantiate our Constructor  var constructorOne = new myObjectConstructor();    //change myProperty of the first instance  constructorOne.myProperty = 'this is a new property';    //instantiate a second instance of our Constructor  var constructorTwo = new myObjectConstructor();    //alert current myProperty of constructorOne instance  alert(constructorOne.myProperty); //this will alert 'this is a new property'     //alert current myProperty of constructorTwo instance  alert(constructorTwo.myProperty); //this will still alert 'this is a property' |

As you can see from this example, even though we changed the property of constructorOne  
it didn't affect myObjectConstructor and therefore didn't affect constructorTwo. Even if  
constructorTwo was instantiated before we changed the myProperty property of constructorOne,  
it would still not affect the myProperty property of constructorTwo as it is a completely different  
instance of the object within JavaScript's memory.

So which one should you use? Well it depends on the situation, if you only need one object of its kind for  
your script (as you will see in our example at the end of this article), then use an object literal, but if you need several instances of an object, where each instance  
is independent of the other and can have different properties or methods depending on the way it's constructed, then use a constructor function.