Much has been written in the community about object-oriented JavaScript, and various patterns have emerged to accomplish with JavaScript's flexible nature the kinds of behaviors that are built directly into a language like C++. Having myself done plenty of C++ in the past, much more than true functional programming, I've found the object-oriented approach to be more familiar and comfortable.

One place OO programmers can hit a bump in JavaScript is the lack of a clear idea of a "class," that is, the definition of a type of object that can be instantiated. JavaScript, in short, has not **class** keyword. Instead it has functions, and only functions. Conveniently, however, if you create a function that populates members of the this variable, the name of that function works exactly like the name of a class.

To borrow a snippet of code from [*Eloquent JavaScript*](http://eloquentjavascript.net/) (a book I like for its depth and brevity, and which is also available for free in electronic form, thank you Marijn), take following function named Rabbit:

function Rabbit(adjective) {

this.adjective = adjective;

this.speak = function (line) {

print("The ", this.adjective, " rabbit says '", line, "'");

};

}

By itself, this function really doesn't do anything meaningful. You can call it from wherever and this will end up being your global context. Maybe handy if you like to pollute the global namespace!

When used with the new operator, on the other hand, this type of function becomes an object constructor, and effectively defines a class as we know in OO programming. We can use it as follows:

var fuzzyBunny = new Rabbit("cutie");  
fuzzyBunny.speak("nibble, nibble!");

As Marijn points out, using *new* with a constructor function like this has a nice side effect which is to assign the object’s *constructor* property with the constructor function itself. This is not true if you just create a function that returns a newly instantiated object.

To make our class even more object-oriented, the other thing that we typically do is assign default values to properties and assign methods within the class, rather than on individual instances. In the first example above, each instance gets a new copy of the anonymous function assigned to *speak*. It would be better to define that function such that a single copy is used by the different instances of the class. This is accomplished by assigning the function to the class’ prototype as so:

function Rabbit(adjective) {

this.adjective = adjective;

}

Rabbit.prototype.speak = function (line) {

print("The ", this.adjective, " rabbit says '", line, "'");

};

Of course, having to write this syntax out for each and every member of the class that’s shared between instances is both cumbersome and prone to errors. Personally, I also like to avoid messing with *prototype* as you can really hurt yourself if you’re not careful.

Thus WinJS provides a helper that provides a cleaner syntax as well as clear separation between the constructor function, instance members, and static members: [*WinJS.Class.define*](http://msdn.microsoft.com/en-us/library/windows/apps/br229813.aspx):

var ClassName = WinJS.Class.define(constructor, instanceMembers, staticMembers);

where *constructor* is a function and *instanceMembers* and *staticMembers* are both objects. The general structure you see in code looks like this (you can find many examples in the WinJS source code itself):

var ClassName = WinJS.Class.define(

function ClassName\_ctor () { //adding \_ctor is conventional

},

{

//Instance members

},

{

//Static members

}

);

As many classes don’t have static members, the third parameter is often omitted. Also, if you pass *null* as the constructor, WinJS will substitute an empty function in its place. You can see this in the WinJS source code for *define* (base.js, comments added):

function define(constructor, instanceMembers, staticMembers) {

constructor = constructor || function () { };

//Adds a supportedForProcessing property set to true. This is needed by

//WinJS.UI.process[All], WinJS.Binding.process, and WinJS.Resources.process.

WinJS.Utilities.markSupportedForProcessing(constructor);

if (instanceMembers) {

initializeProperties(constructor.prototype, instanceMembers);

}

if (staticMembers) {

initializeProperties(constructor, staticMembers);

}

return constructor;

}

You can also see how *define* treats static and instance members (*initializeProperties* is a helper that basically iterates the object in the second parameter and copies its members to the object in the first). Static members are specifically added as properties to the class function itself, *constructor*. This means they exist singularly on that object—if you change them anywhere, those changes apply to all instance. I consider that a rather dangerous practice, so I like to consider static members as read-only.

Instance members, on the other hand, are specifically added to *constructor.prototype*, so they are defined just once (especially in the case of methods) while still giving each individual instance a copy of the properties that can be changed without affecting other instances.

You can see, then, that *WinJS.Class.define* is really just a helper: you can accomplish everything it does with straight JavaScript, but you end up with code that’s generally harder to maintain. Indeed, the team that wrote WinJS really needed these structures for themselves to avoid making lots of small mistakes. But otherwise there is nothing magical about *WinJS.Class.define*, and you can use it in your own app code or not.

Along these lines, people have asked how *WinJS.Class.define* relate to the class structures of TypeScript. When all is said and done, they accomplish the same things. In fact, you can derive WinJS classes from TypeScript classes and vice-versa.

The one exception is that call to *WinJS.Utilities.markSupportedForProcessing* in WinJS. This is a requirement for functions that are used from other parts of WinJS (see Chapter 4 of [my book](http://blogs.msdn.com/b/microsoft_press/archive/2012/10/29/free-ebook-programming-windows-8-apps-with-html-css-and-javascript.aspx), pages 145-146) and is the only “hidden” benefit. If you use TypeScript or other libraries to create classes, you’ll need to call that function directly.

In Part 2 of this topic, we’ll look at *WinJS.Class.derive*. In Part 3 we’ll finish up with *WinJS.Class.mix* and the idea of mixins.