Object Oriented Programming in Javascript

This chapter expands on the idea of Objects introduced in Chapter 2, including: Properties and Methods, Constructors, 'this' and object members, scope, 'Classes', inheritance and the prototype, Javascript file management and test methods, Javascript "oddities".

Objects and Object Creation

There are three main ways to create an object in Javascript code:

1. Object Literal Notation
2. Literal notation and augmentation
3. A constructor function

All of these are useful in some situation or other, but for people who have programmed in another programming language (typically Java, C#, Python), the third is probably the most recognisable. It also tends to be the most useful in situations where you want to create many objects that follow a specific template. After all, Object Literal Notation is a way to build objects on an object-by-object basis, and contains nothing to enforce the constraint that all objects of a specific type have the same members in them.

Object Literals

An object literal is a statement that assigns a group of object data to a single variable:

```
var person = {
  name: "Fred Bloggs",
  email: "fred@bloggo.com",
  dob: new Date("February 25, 1975"),
  age: function(){
       var msPerYear = 1000*60*60*24*365.25;  // millis*secs* mins*hrs*days
       return Math.floor((new Date()-this.dob)/msPerYear);
  }
};
```

Listing 3.1: An Object defined as an Object Literal

Note the structure of this – each individual assignment is of the form ***name: value;***. All assignments are separated by commas (i.e. the last assignment, which is the age() function, does not have a trailing comma). The main assignment is a complete declaration statement (var person =) which ends in a semicolon.

As a consequence of this, we now have an object that contains data (the person's name, email address and date of birth) and a method (i.e. a member function, which calculates the person's age). The method produces "live" results since part of the age calculation involves the current date (new Date() always produces the date and time ***now***). When we wish to use the object in a program, we can easily access the individual bits of data:

```
document.writeln("Email: " + person.email);
```

and can also access the function:

```
document.writeln(person.name + " is " + person.age() + " years old.");
```

An object literal gives us a quick and easy way to package up information that we want to use in a program, and as such represents the most straightforward way of building objects into an application. However, as convenient as this method is, it does not always fit in with the way we need to write a program.

Object Literals with Additional Statements

Consider, for example, a web-form, which lets an end-user enter various bits of personal data into text boxes, which can be interrogated by JS code. In this case, we don't have the option of putting literal values like "Fred Bloggs" or "February 25, 1975") directly into the JS code, and will need to come up with an alternative method. Instead we can create an empty object and assign the values to it.

```
var person = {};      // Here, we have an "empty" object.  It is not truly empty,
                          // since it will  already have the properties and methods
                          // that ALL objects have
person.name = "Fred Bloggs"; // We could have assigned from a variable or
                                        // another object, like a text box on a form
person.email = "fred@bloggo.com";
person.dob = new Date(1975, 2, 25);   // Note the alt way to create a date
person.age = function() {
    var msPerYear = 1000 * 60 * 60 * 24 * 365.25;
    return Math.floor( (dob – new Date())/msPerYear );
};
```

Listing 3.2: Filling an empty object literal

Here we have a more workable method of dealing with the values that might emerge as a user interacts with a JS application. This **person** object and the previous one constructed by object literal notation are exactly equivalent, can be used in the same way and will behave in the same way.

Constructor Functions

One fairly smart way of improving the consistency of how objects are built is to use a **constructor** function. This is just a function that contains all of the code that you would have used to create an object by adding to an empty object (i.e. the previous method described), although there is scope to improve this by passing values into the constructor function as parameters:

```
var Person = function(name, email, dob) {
  this.name = name;
  this.email = email;
  this.dob = dob;
};
Person.prototype.age = function() {
  var msPerYear = 1000 * 60 * 60 * 24 * 365.25;
  return Math.floor( (this.dob – new Date())/msPerYear );
};
```

Listing 3.3: using a Constructor function

There are a few things about this constructor function that need explaining:

- It is a convention among JS programmers to start a constructor's name with a capital letter. This is the only type of identifier that conventionally starts with a capital, so makes it easy to distinguish the definition and use of a constructor from other functions.

Given that the purpose of this function is to create objects, and that it has some special syntax features that other functions don't have, this is a good convention, and one you should stick with

- The keyword ***this*** is used to distinguish the members of the object being constructed. This serves two purposes – first it means you don't have nonsense statements like ***name = name;*** in the code, and secondly, it lets the JS interpreter distinguish between members of the object and other variables. Note that in the age() function, we need to use ***this.dob*** to make sure we access the dob value for a specific object. Since the code can only be called as part of an object, using dot notation (**person.age()**), this refers to the specific object that **person** references. Sometimes we call this the *context object*

- Although it would be possible to also pass the function definition into the constructor as a parameter, this would be considered to be bad practice, since the code would have to be put into a string to stop it being mixed up with the calling code. Generally, you should avoid putting function code into strings in Javascript, since these become areas of vulnerability that hackers might make use of (e.g. if a hacker can change the definition of a function in a string, they can change the way a website works – this is never done for anything other than nefarious purposes). Functions are therefore added to the object as separate assignments, to the object's ***prototype***. We'll see more on this later, but basically it is the way to indicate that *every* object created with this function should have access to the age() function. This use of ***this*** and the ***prototype*** definition are core features of using a constructor function.

Again, we can use the resulting object in exactly the same way as the ones created by object literal notation or by augmenting an empty object (the two previous methods described). There is one small change in how an object is constructed, however:

```
var p = new Person("Fred Bloggs", "fred@bloggo.com", "February 25, 1975");
```

Note that here we've used the keyword ***new***, which is a signal to JS that a ***new object*** is to be created. If we were to use the Person() function without the new keyword, an object would be created, but the internal members would not be properly wired up and so it would behave differently (specifically, the ***this*** keyword would not refer to an object, but instead to the current context, which would be the constructor function itself – weird!).

There are some significant advantages to using a constructor (and added prototype functions) for object creation:

- Using a function, it is much more likely that all the objects made by it will behave consistently. It is too easy to miss out an important assignment by just adding members to objects in separate statements. All objects created through the same constructor will have the same set of members

- In objects created by separate statements the ***prototype*** would not normally be used (it is only automatically associated with calls that use the ***new*** keyword) – a consequence of this is that each separately defined object would need to have member functions

assigned individually to it, so instead of one function definition serving any number of objects, any number of objects would have their own definition of functions assigned. Apart from the waste of memory this brings about, it could also lead to different objects behaving differently if not all of the function definitions are exactly equivalent

- Generally, it is now much quicker and easier to create a set of consistent objects, each in a single statement. For the lazy programmer (i.e. all of us), this is a massive reduction in work compared to the alternatives

Properties & Methods

In programming, we talk of variables and functions as being the units from which a program is assembled. In object oriented programming, the more common terminology is ***object properties*** and ***object methods***. Of course, an object property is just a variable that is bound to an object, and similarly a method is a function that is bound to an object. However, it is a good idea to keep these distinctions in mind (especially the "bound to an object" bit) since they make a significant difference in the way that variables and functions are used.

A ***Property*** is a setting that applies to an object – e.g. a Person's name is accessed through the ***.name*** property. The significance of this is that every Person object has a name, whether it has a value assigned to it or not.

A ***Method*** is a behaviour that an object can be expected to have – e.g. we can ask any Person object what the age() of the person is. If no ***dob*** value has been assigned, the method won't return a useful value, but it will always be valid to call the method, and we can build code into the method to make sure that even an unassigned birth date does not cause a problem.

In Javascript, properties are simply member variables (i.e. those accessed through the ***this*** keyword). Some other programming languages are more prescriptive – for example in Visual Basic and C#, a property is defined as a pair of functions – one for setting a value in an object, and one for retrieving the current value. You can still do this in Javascript, but you would need to use two separate method definitions to do it. For example:

```
var Person = function(name, email, dob) {
  this.name = name;
   this.email = email;
  this.dob = dob;
};

Person.prototype.age = function() {
      var msPerYear = 1000 * 60 * 60 * 24 * 365.25;
      return Math.floor( (this.dob – new Date())/msPerYear );
};
Person.prototype.setEmail(email) {
      this.email = email;
};
Person.prototype.getEmail(){
      return this.email;
};
```

Listing 3.4: An object with an *email* property

In the code above, functions setEmail() and getEmail() have been added to the Person object type so

that we can access and change the email member of the object (this is the one most likely to need changing – you rarely change your name and never change your date of birth). Of course, this is almost pointless, since we can always just access the ***.email*** member directly. However, there are a couple of good reasons for creating ***accessor*** functions (e.g. getXXX(), setXXX()) for an object:

- Rather than simply assigning or retrieving values from the object, we could also make these functions do some useful work. For example:

```
Person.prototype.setEmail(email) {
    if((email.indexOf('.') > 2) && (email.indexOf('@') > 0){
        this.email = email;
    } else {
        window.alert("Invalid email address");
    }
};
```

Listing 5: Using a setXX() property to validate a value

  This version of setEmail() includes a fairly clunky test for a valid email address (there are better ones but this serves the purpose). The end result is that it is now not possible to assign an invalid email address (i.e. one that follows the rules for an email address), although it is still possible that the address could be wrong

- We can if we wish make the ***email*** member *inaccessible* from outside the object. We could do this using a ***closure***, which is a bit like wrapping up object members in a separate scope. If we did this, getEmail() and setEmail() would be the *only* ways of accessing that member variable, and the two functions in combination would become the actual email property of the object. We'll look into closures later in the module

So, most of the time, the terms Property and Methods are simply new terminology for existing bits of code (variables and functions) that we use to distinguish that they are parts of an object. However, there is always the actual distinction that properties and methods are associated with the ***this*** of an object, and sometimes we will actually alter the way they work fundamentally.

Some Javascript programmers create *pseudo-private* member-variables by preceding a variable's name with one or two underscores (e.g. __email). This does nothing in the JS code to make the __email variable inaccessible, but it does at least indicate (to people who follow the convention) that it should not be accessed in code outside that which defines the object, and that in turn allows programmers to create true properties as getXX() and setXX() functions to be used to access private members. Javascript 2.0 (the proposed next version of the language) has reserved the keyword *private*, suggesting that it will be used to create truly private members of an object (similar to the way Java does). Javascript programmers are advised to avoid using this keyword in their code (see javascript.about.com/library/blreserved.htm for a full list of reserved words and words you are advised to avoid).

Constructors

We've already seen that a constructor is simply a function that you use to define the make-up of an object. There is a bit more to it than that because we need to use ***new*** to make a constructor work properly (otherwise, the object will not be connected to the functions defined for the prototype). The function that was used as an object constructor ought to have a name that begins with a capital,

but this is not enforced.

Under the hood, JS does a bit more when you use a constructor. As you can see in figure 3.1, one of the members of an object that JS recognises is called "constructor":

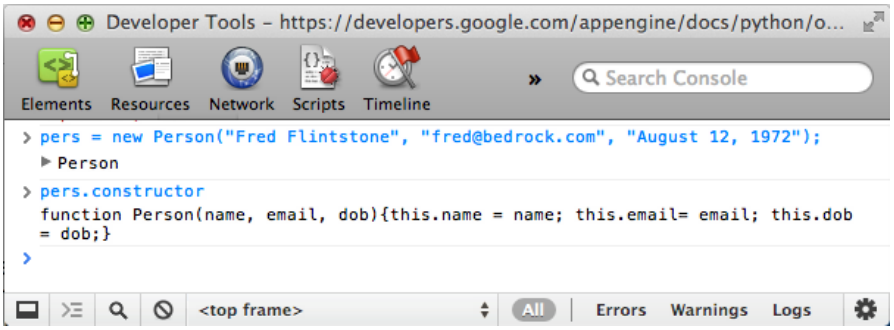

Figure 3.1: An object in the JS console, showing the "constructor" member

The value returned by the ***constructor*** property is the entire function definition of the constructor that was used to create the object. Mostly there is not any use for this in JS programming, although developers who go on to create programming tools may find a useful purpose for it. The fact that a constructor function is definitely physically bound to an object gives you ways to find out about how an object was built, and this would allow you to generate code that interacts with the objects at a fundamental level if that was necessary (this is more a system programmer's job than anything you might do while building a JS application).

'Classes'

Javascript is best described as a "prototypical object oriented programming language" (try saying that after a few rounds). This description distinguishes JS from most of the current popular programming languages, which are "class based". In languages like Java, C, C++, C#, Python, Ruby and Visual Basic (to name just a few), it is not possible to simply create an object. First a programmer needs to create a class (which is more or less a *template for an object*), and then create objects by getting the class to generate them. This follows from the "strong typing" employed by these languages.

Strong typing is a way of saying that once a variable has been created, its type cannot be changed. A number variable is stuck with storing a numeric value; a string variable can only ever hold a string. One consequence of this is that every variable must belong to a recognised (by the language) type, and it is a variable's type that determines what values it can hold and what functions it is bound to.

Javascript is the only popular object-oriented language that goes against this flow, and one consequence is that programmers who move to Javascript from other languages can get confused. Programmers wrongly talk of a Javascript class, when the only similar notion in JS is the use of a constructor function. The idea of a class is that there is a set of objects that all have the same properties and are bound to the same functions as each other – they share similar member definitions, and can therefore be used identically. In Javascript, you can always upset that by adding a new function or member variable to a Person object or changing an existing one, thereby destroying the compatibility between objects. For example:

```
var joePerson = new Person("Joe Bloggs", "joe@bloggo.com", "June 2, 1980"),
    fredPerson = new Person("Fred Smith", "fred@smiddy.co.uk", "March 18, 1976");
```

```
// So far, we have two person objects that are equivalent types – they will behave
// differently when manipulated by similar executable statements.

fredPerson.jobDescription = “Programmer”;
fredPerson.description = function(){
    return this.name + “ is a “ + this.jobDescription;
};
```

Listing 3.5: Amending an object created by a constructor

We can no longer say that ***joePerson*** and ***fredPerson*** are identical types – ***fredPerson*** has an extra property (***jobDescription***) and an additional method (***description()***). If a call was made to ***joePerson.description()***, it would return the value *undefined* (same with the .***jobDescription*** property).

Ok – so what is the significance of this? Well, first, it means that JS objects can be used more flexibly; if it would be useful for a specific object to have a generateWidget() method, we can add it. It will have no effect on other objects created with the same constructor (if constructors were used at all). That can, in some circumstances, be a good thing.

However, the downside of this is that we can no longer know what an object is capable of simply by finding out what its constructor function was (by accessing it using the object.constructor syntax). It also means that other programmers *who manage to access one of your objects in their code* can change how it works, and this must be considered a very dangerous thing indeed. Many web hackers find ways into a system by getting access to and manipulating JS code; this is one reason why web browsers must be constantly updated, as hackers find new and more ingenious ways to get access to objects within a browser.

For example, think of the potential if a hacker could access the code in your browser while you were doing some online banking; the text boxes for your account id and password are objects, and could easily have new methods added to them to send the information you type back to the hackers server. Fortunately, browser security makes this pretty much impossible (but only until the next exploit is found – update your browser frequently!).

In Javascript, we can use an object’s prototype to define behaviour that is common to a whole collection of objects. If they share the prototype, they also share the same capabilities. We can also use the prototype to ***augment*** all of the objects of a particular type – ***even ones that have already been created***. For example:

```
String.prototype.times = function(count) {
    if(count < 1) {
        return “”;
    } else {
        var s = “”;
        while(count) {
            s += this;
        }
        return s;
    }
}
```

Listing 3.6: Updating the String type

This fairly ordinary looking function augments the behaviour of the Javascript String type – once the JS interpreter has seen this code, *every string* within that program will have a .times() method, so:

```
"Fred..".times(4);        // evaluates as "Fred..Fred..Fred..Fred.."
```

By manipulating the prototype, we have just changed the way a normal String works. With great power must come great responsibility (or so Spiderman seems to suggest).

Inheritance

Object-oriented programming is based on three broad principles. The first, encapsulation, we have seen already. In Javascript, encapsulation is the ability to create objects – 'capsules' – made up of data and behaviour. Other programming languages (e.g. Java) are a bit more restrictive in their definition of encapsulation: not only is it about amalgamating lots of bits of data and functions into objects, but also defining access mechanisms so that only the properties and methods that a user (i.e. a programmer who uses the class) needs to access are accessible. We'll find how we can do this later (in Javascript it requires a trick).

The other two principles are **Inheritance** and **Polymorphism**. Inheritance in programming means that you can automatically assign methods or properties from one class on to members of another class – i.e. we can re-use the code in other classes. As we've seen, Javascript has no classes, but we can still inherit properties and methods from one object into other objects. As you might guess, this is done using the prototype of the first object. Polymorphism is about how inheritance can be used to create objects that are compatible with each other but specialised in some aspects of their behaviour.

We already have a Person constructor that allows us to create many instances of a person. What if, instead of simply adding a jobDescrption property and a description() method, we decided it would be better to have a Programmer constructor, that meant we could create programmers (specialised forms of Person) at will? Every programmer is also a person, so it would be easiest if we could re-use the Person constructor in some way – i.e. the Programmer type should inherit the traits of the Person type. Recall that Person has two bits of code defining it – the constructor function and the age() method. Here's how we'd inherit it:

```
// First define a constructor that applies the person constructor internally…
var Programmer = function(name, email, dob, language){
   Person.apply(this, arguments);  // arguments is the set of values passed – name etc.
   this.language = language;              // This is the one value that Person does not handle
};

// Now assign a new Person object as a prototype (it doesn't matter that there are no
// values passed to this – they'd be ignored anyway; it is only age() we want to inherit.
Programmer.prototype = new Person();

// Finally, add new methods to the Programmer class (purely as an example)…
Programmer.prototype.programsIn = function(lang){
   if(lang === this.language) {
      return true;
   }
   return false;
};
```

Listing 3.7: A pattern for inheriting properties *and* methods

The form of the inheritance used above is about as close as you will get to the type of inheritance used in "class-based" languages like Java. It ensures that all of the members sent as parameters to the parent's constructor (i.e. Person()) are passed on to the new constructor and that all of the methods available to the parent class are also available to the new class. The first two blocks of code do this. From here on, we can add members, including prototype members for the new class, as we wish. A Programmer object can be used as:

```
var prog = new Programmer("Joe Bloggs", "joe@bloggo.com", "August 26, 1978",
                          "Javascript");
document.writeln(prog.age());                          // prints 33
document.writeln(prog.programsIn("Javascript");        // prints true
```

Listing 3.8: Using the Programmer class (that inherits all of the Person class)

In the set text for this module (Object-Oriented Javascript, Stoyanov), you'll find a lot of other approaches to inheritance in JS. Some of these try to emulate classical object-oriented programming are quite complex and code-heavy. Others are approaches that could only be used in Javascript, and acknowledge that JS, as a prototypical object-oriented language, has more appropriate ways of doing things. I get by using the method outlined above, but that may be because I'm more comfortable with the Java/C++/Visual Basic ways of doing things.

Polymorphism

In object-oriented programming, one of the main principles is ***polymorphism***. This simply means "many-forms" (Greek: poly = many, morph = shape or form). For object-oriented programs, this principle is used to create groups of objects that are compatible with each other, and therefore can be manipulated by exactly the same code. We can see this in the simple inheritance example above. A Person object has properties name, email and dob, and an age() method. A Programmer object, by virtue of inheriting from Person, also has these properties and methods.

As a consequence of this inheritance, and statement that we can apply to a Person object is also valid to apply to a Programmer. For example, a programmer has name, email and dob properties, and also an age() method. Semantically, we can say that a Programmer IS-A Person. In OOP, IS-A is a test for inheritance – it says that an object is a specialized version of some other object, and so we are saying that it is type compatible.

Note that we can't say the converse – a Person is not a programmer, because it lacks the ***language*** property and does not have a ***programsIn()*** method. Inheritance is a one-directional relationship, from the parent object to the child object.

So what does a bit of polymorphic code look like? Generally, it is use to manipulate a group of objects arranged in a collection, so:

```
var people = [];  // An empty array, now we'll add objects.
people[0] = new Person("Joe", "joe@schmoe.com", "January 12, 1979");
people[1] = new Programmer("Pete", "pete@coders.com", "April 3, 1982");
people[2] = new Person("Bill", "bill@stickers.com", "June 11, 1975");
var totalAge = 0;
for(person in people){              // This is a for loop for iterating through all of the array
      totalAge += person.age();
}
document.write("Average age is: " + (totalAge/people.length).toFixed(2));
```

Listing 3.9: Using polymorphism (A Programmer is-a Person)

The polymorphism here is in the use of all of the members of the people[] array as if all of them were Person objects – i.e. what goes on in the for() loop. Person is the common type in this array (as described earlier, a Progammer IS-A Person), so it is safe to consider each as a person. The neat thing is, this would have worked even if we had re-defined how a Programmer's age was worked out.

The classic example of polymorphism is one that you'll get working in the lab this week. We build a number of different object types based on a Shape class (i.e. all will inherit from Shape), each implementing a different geometrical figure (rectangles, circles, triangles etc.). Each of these types will have a draw() method, but obviously, each of the draw() methods will have to be implemented differently so that the Rectangle type draws a rectangle, the Circle draws a circle etc. We'll then build an array of different object types, and implement a method that will draw the collection.

Questions

1. Each of the three ways to create an object in JS code has its own advantages: give one advantage for each creation style?
2. When using a constructor function, it is important to use the keyword ***new*** at the point of calling it. What would JS do if you were to omit the new keyword? Look this up in Google if you can't work out what would happen.
3. An object's prototype is the mechanism by which the object's traits can be inherited by other objects, so for an object X, we can add a function to all objects based on X by using *X.prototype.f = <some function definition>;*. What do you think would happen if the word *prototype* were omitted from this? If you're not sure, Google "Javascript prototype" for an explanation (http://www.javascriptkit.com/javatutors/proto.shtml provides an easy to follow one).
4. Since everything in Javascript is an object, the distinction between a function and a method is indistinct. Assuming we have created a function that we have not bound to any specific object or prototype, what object is this function a method of?
5. Javascript does not implement strong typing. What drawback does this present to a JS programmer – especially one who comes from a language that does implement strong typing? (You should Google an answer to see how accurate your own answer is).
6. Explain how the ability to augment an object in JS code can pose a security risk in some situations.
7. Javascript makes it easy to add functions and variables to an object. Why should a programmer bother with defining a constructor and manipulating the prototype – i.e. what is gained from this way of doing things?
8. In an online staff management application, we need to define a new type of programmer, a SeniorProgrammer, so that objects of this type can be assigned a single normal programmer to supervise. Write code (like that above for the Programmer type) to define this new type, which should have a ***subordinate*** member variable (e.g. boss.subordinate = "Fred Bloggs"), and a ***supervises()*** method (which returns true if a name passed is managed by that SeniorProgrammer, false otherwise – e.g. if(boss.supervises("Fred Bloggs")).
9. In a method definition, what would be the result of an error in which the **this** keyword was omitted when assigning a value?
10. Inside a method definition we can place a call to a global function. Does this have an effect on the object's scope?
11. What would happen if, in the previous example, we were to add a second loop, which iterated through all of the array members to find if any of them programmed in Java?
12. Polymorphism is usually used to simplify programming at the *application level*. i.e. creators of a new object type do not directly benefit from it, but developers of an application that uses the new type do get to benefit. What justifies the additional time a developer spends on an object definition to make it work polymorphically?