JavaScript Further Prototypes

The following topics should be considered optional extras for those who wish to understand JavaScript objects more deeply.

Prototype chaining

Consider this code which creates a cat object:

```
var tom = new Cat("Tom", 4);
tom.makeNoise();
tom.favouriteFood = "fish";
```

As we have seen already, ***makeNoise()*** is ***not*** actually a method of our cat object, ***tom***, but of its ***prototype*** - yet we can access it simply with ***tom.makeNoise()***. (Compare that to the ***favouriteFood*** property which belongs to the ***tom*** object specifically). This code illustrates the concept of ***following the prototype chain***. If we try to reference a method or property of an object, this happens:

- Does this specific object have this method or property? If so, use that (e.g. ***favouriteFood*** in the example above)
- If not, examine the object's ***prototype*** for this method or property. If found, use that (e.g. ***makeNoise()*** in the example above)
- If the method or property is still not found, go back one step to the prototype of ***that*** object, and so on until there are no more objects in the chain, at which time, if not found, stop with an error.

Changing the prototype of an object

It is possible to dynamically change properties and methods of the prototype ***after*** objects have already been created using the constructor. If this is done, ***any new objects created with that constructor will have the new properties and methods***. Furthermore, because ***there is only one copy of the prototype object which is shared between all created objects***, any ***existing*** objects will have the new properties and methods too! For example:

```
function Cat(n,a)
{
    this.name=n;
    this.age=a;
}

Cat.prototype.species = "Felis catus";
Cat.prototype.nLegs = 4;
```

```
Cat.prototype.makeNoise = function() { alert("Meow!");

var cat1 = new Cat("Tiddles", 10);

alert(cat1.species); // "Felis catus"
cat1.makeNoise(); // "Meow!"

Cat.prototype.species = "Canis familiaris"; // Species
Cat.prototype.makeNoise = function() { alert("Woof!");

var confusedCat = new Cat("Tigger", 8);

confusedCat.makeNoise(); // "Woof!"
cat1.makeNoise(); // "Woof!"
```

As well as changing individual properties of the prototype, we can ***completely replace the prototype*** with a new object. The effect of this will be that ***any new objects will use the new prototype*** while ***objects already created will continue to use the old prototype***. This is because the new prototype occupies a separate area of memory to the old, so the old prototype used by previous objects is untouched. In the previous example, if we change individual properties of a prototype, we are changing the contents of memory shared between all objects, so the properties of all objects using that prototype will change. The code example below illustrates this:

```
function Cat(n,a)
{
    this.name=n;
    this.age=a;
}

Cat.prototype.species = "Felis catus";
Cat.prototype.nLegs = 4;
Cat.prototype.makeNoise = function() { alert("Meow!");

var cat1 = new Cat("Tiddles", 10);

alert(cat1.species); // "Felis catus"
cat1.makeNoise(); // "Meow!"

// Reset the prototype to an *entirely new object*
Cat.prototype = new Object();
Cat.prototype.species = "Canis familiaris"; // Species
Cat.prototype.makeNoise = function() { alert("Woof!");
```

```
var confusedCat = new Cat("Tigger", 8);

confusedCat.makeNoise(); // "Woof"
cat1.makeNoise(); // "Meow", still
```

Use of Object.create()

Recent versions of JavaScript, supported on current browsers, can make it easy to create objects which use a given prototype. ***Object.create*** takes in a prototype and gives us an object which uses that prototype. A good way of visualising it is a machine, in which you insert a prototype and you get an object out which contains that prototype. It takes an object prototype as an argument (and optionally, a list of additional attributes to add to the created object as a second argument).

Example of using Object.create()

To create a single object which uses a given prototype:

```
// Setup a Cat prototype
// Blank constructor, we no longer specify name and age
// as we create objects with Object.create(), not with
// The constructor is just there to "attach the prototy
// the prototype is a property of the constructor
function Cat()
{

}

Cat.prototype.species = "Felis catus";
Cat.prototype.makeNoise = function() { alert("Meow!");

// Create an individual cat object, i.e. an object usir
var tiddles = Object.create(Cat.prototype);
tiddles.name="Tiddles";

// and another
var tom = Object.create(Cat.prototype);
tom.name="Tom";

// call makeNoise on both
tiddles.makeNoise(); // "Meow!"
tom.makeNoise(); // "Meow!"
```

Inheritance in JavaScript

What is inheritance?

It is possible to simulate ***inheritance*** in JavaScript using prototypes. The object-oriented concept of ***inheritance*** allows us to use an existing type of object as a basis for a new, related, more specific object. Imagine we wanted to write objects representing different types of employee, e.g. Programmer and Manager. All employees would have common properties, e.g. a name, job title and salary, and common methods, e.g. code to display the name, job title, and salary, and a method to update the salary (this could perform validation, such as checking it's not a negative number, and, hopefully, checking that the new value is an increase from the old which matches the rate of inflation). Programmers could then have their own properties, e.g. favourite programming language(s), and managers could have properties such as number of shares in the company.

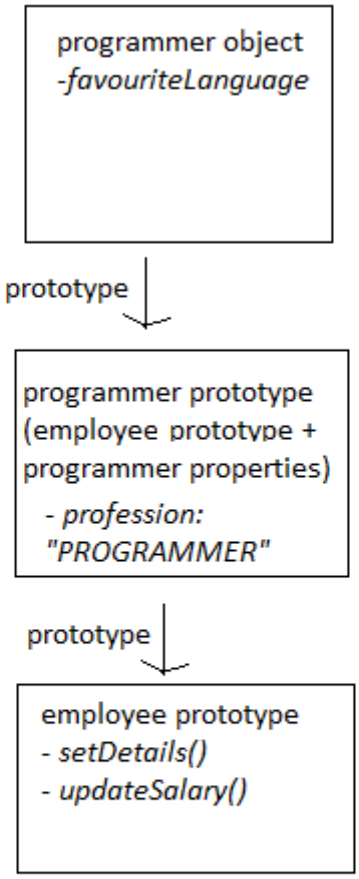
If we were to write the code (in JavaScript, prototype objects) for programmers and managers separately, we'd be repeating the code common to employees in Programmers and Managers. So what can we do instead? We can use ***inheritance***. This involves defining methods and properties common to all employees ***once***, and then ***inheriting*** these methods and properties in subtypes of employee.

Simulating inheritance in JavaScript

JavaScript doesn't provide true inheritance (because of this, I will frequently use the term "inheritance" in quotes when discussing it in the context of JavaScript) but we can simulate it with prototypes.

To simulate inheritance in JavaScript, we do the following (using programmers inheriting from employees as an example):

- First define a constructor for creating employee objects, together with an associated prototype. This prototype will contain methods and properties common to all employee objects.
- Next, define a constructor for the ***subtype*** of employee, such as Programmer. Set the ***prototype*** associated with this constructor (i.e. the prototype for programmers) to be an object which ***uses the employee prototype as*** **its** ***prototype***. This can be done using ***Object.create()***.
- Add subtype-specific properties and methods to the subtype prototype, e.g. programmer-specific properties and methods to the programmer prototype.
- Due to prototype chaining, Programmer objects will be able to use both properties and methods defined in the subtype (programmer) prototype ***and*** properties and methods defined in the parent type (employee) prototype. This is shown on the diagram below.



This is probably a bit confusing without some code, so here it is:

```
// Constructor to create employee objects
function Employee(n,t,s)
{
    this.name = n;
    this.jobTitle = t;
    this.salary = s;
}

// Method to set the details, intended to be used immed
// creating the object
Employee.prototype.setDetails = function(n, jt, s)
{
    this.name=n;
    this.jobTitle=jt;
    this.salary=s;
}
```

```
// Method to update the salary
Employee.prototype.updateSalary = function(newSalary)
{
    if(newSalary < this.salary)
    {
        alert("ERROR!!! Salaries cannot go down!!!");
    }
    else
    {
        this.salary = newSalary;
    }
}

// Constructor for Programmer objects
function Programmer(favLang)
{
    this.favouriteLanguage = favLang;
}

// The Programmer prototype will have, as its prototype
// This means that programmer objects will be able to u
// and properties as well as programmer methods and pro
// prototype chaining.

Programmer.prototype = Object.create (Employee.prototyp

// Add a method to the Programmer prototype. All Progra
// able to use this method.

// (In case you are not aware, the proper definition of
// code for the love of it. It does not mean to illegal
// steal data. That is "cracking")

Programmer.prototype.hack = function () { alert("Hackin


// We can also add additional properties to the Program
// So the prototype for programmers will include a prof
//"PROGRAMMER"

Programmer.prototype.profession = "PROGRAMMER";

// Do the same with managers
function Manager(shares)
```

```
{
    this.nShares = shares;
}

Manager.prototype = Object.create (Employee.prototype);
Manager.prototype.profession = "MANAGER";
```

Prototype chaining and "inheritance"

To clarify how prototype chaining works with "inheritance" (which was introduced above), consider this code which creates an employee object and sets its details:

```
var john = new Programmer("JavaScript");
john.setDetails("John Scott-Jones","Web Developer",3500
john.hack();
```

What's happening here?

- ***setDetails()*** is ***not*** actually a method of our specific programmer object, ***john***, so we go up one level to its prototype.
- The programmer's prototype ***still*** does not contain a ***setDetails()*** method. Recall that ***setDetails()*** belongs to ***the Employee prototype***, ***not*** the Programmer prototype.
- So we go up one step further in the chain and find the ***setDetails()*** method in the Employee prototype, which is the prototype of the Programmer prototype which is in turn the prototype of our specific programmer, ***john***.
- On the other hand, ***hack()*** belongs to the prototype of Programmer.

Exercise

1. Create a Vehicle constructor and prototype to represent ***any*** type of vehicle, such as cars, bikes, buses, and so on. The constructor should take no parameters.
2. Add to the Vehicle prototype a ***setDetails()*** method with parameters for ***make***, ***topSpeed*** and ***nWheels***. This should set up the corresponding properties.
3. Add a ***move()*** method to the Vehicle prototype. This should simply bring up an alert box displaying "Moving along..."
4. Create a ***Car*** constructor. This should have one parameter, an ***engineCapacity***, and set up an ***engineCapacity*** property to the value of this parameter. It should also set up an ***engineRunning*** property to be ***false***.
5. Set the ***Car*** prototype to use the ***Vehicle*** prototype.

6. Add a ***move()*** method to the Car prototype. This should test whether the engine is running (use the ***engineRunning*** property) and display either "Moving along!" or an error message accordingly. Also add ***startEngine()*** and ***stopEngine()*** methods to set engineRunning to true or false, respectively.
7. Add appropriate ***toString()*** methods to the Vehicle and Car prototypes. For Vehicle, toString() should return a string containing the make, top speed and number of wheels, while for Car, it should contain the engine capacity and engine running properties too.
8. Test it out by creating a couple of car objects and a plain vehicle (could be a bike, for instance), setting their details, and trying to move them.

Online References

The presentation of this material was inspired by the following online references which you might find useful:

- Dmitry Soshnikov: ECMA-262-3 in detail. Chapter 7.2. OOP: ECMAScript implementation
- Angus Croll: Understanding JavaScript Prototypes
- Ian Elliot: Easy JavaScript Prototype Inheritance

As always I would recommend David Flanagan's "JavaScript - The Definitive Guide" for further reading.