JavaScript Prototypes

Today we will look at JavaScript ***prototypes***.

Constructors and Prototypes

Last time we looked at JavaScript objects. However, there was a problem in which we were unable to easily set up ***many versions of similar objects***, such as many cats which all have different names but all work in the same way, i.e. all have the same methods. From what we have seen already, the only way of doing this is to add the methods (such as ***eat()***, ***walk()*** and ***makeNoise()***) to ***every single object of a given type*** (e.g. every single cat). Clearly, this is inefficient and time-consuming, as the following code illustrates:

```
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++;
        }

    };

var cat2 = {
        name: "Tom",
        age: 5,
        weight: 5,

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

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

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

    };
```

To create two cats with different properties, we have to ***repeat the code for the methods*** which is inefficient as the methods work in the same way for each cat. This topic will examine how we can get round this problem.

The Constructor

If you have had experience of other object-orientated languages, you will have come across the ***constructor***. The constructor is a special method used to define ***how objects are created***. In JavaScript, the constructor works ***in a different way to other languages***. It is ***an ordinary function*** which is used to create objects when ***used together with the "new" keyword***. What do we mean by this? Consider this function which could be used as a possible constructor:

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

When ***used with new***, this constructor function can be used to create new Cats. It takes two parameters, n (the name) and a (the age), and sets the ***name*** and ***age*** properties of objects created using this constructor to those two parameters. (***"this"*** refers to the object that will be created with the constructor). So we could use the constructor function to create two new Cat objects:

```
var cat1 = new Cat("Tiddles", 10);
var cat2 = new Cat("Tigger", 7);
```

```
// Display the cats to show that it worked
alert(cat1.name + " " + cat1.age);
alert(cat2.name + " " + cat2.age);
```

When the ***new*** keyword is used in code together with a function name (e.g. ***Cat***) this tells JavaScript to ***create a new object*** using the specified function (Cat here). Note that unlike other languages, Cat is ***not*** a data type. It is simply a function name.

In fact, one could call Cat just like a regular function, but it wouldn't do a lot in that case. For instance the code

```
var cat1 = Cat("Tiddles",10); // note no "new"
```

would be perfectly legal JavaScript, but wouldn't do anything, unless the Cat function returned something, in which case the variable ***cat1*** would be assigned the return value of the Cat function.

Constructors and methods

We can also set up ***methods*** in the constructor. For instance, we could do the following:

```
function Cat(n,a,w)
{
    this.name=n;
    this.age=a;
    this.weight=w;
    this.species="Felis catus";
    this.makeNoise = function()
        {
            alert("Meow!");
        };
    this.eat = function()
        {
            this.weight++;
        };
    this.walk = function()
        {
            this.weight--;
        };
}

// Function to test creating objects - might be linked t
function testobjects()
{
    var cat1=new Cat("Tom",7,7);
    var cat2=new Cat("Tigger",8,8);
    cat1.makeNoise(); // "Meow!"
    cat2.makeNoise(); // "Meow!"
    cat2.walk();
    alert(cat2.weight); // 7
}
```

The code above will work. The Cat constructor creates objects which have ***name***, ***age*** and ***weight*** properties, a ***species*** property which is always "Felis catus" (the Latin species name for the domestic cat), and ***makeNoise()***, ***eat()*** and ***walk()*** methods. It attaches them to the current object being created with the constructor because we use ***this***. And it saves us from the laborious task of having to add the methods to ***every single cat object*** in our code.

There is one problem though. Remember that the constructor is a function used to ***create new objects***. The problem is that ***each object will have its own copy in memory of the properties and methods***. For those properties which change for every cat, such as name, age and weight, that is fine, but for the methods, which always work in the same way for every cat, and constant properties such as the species, this is inefficient - each cat object will hold its ***own copy*** of the method or property in memory. What we really want is ***one copy*** of each method, and constant property, to be shared amongst ***all*** cat objects. How do we do that? We use a ***prototype*** - a special "template" object which can be shared across a set of similar objects.

Prototypes

Constructor functions have a special property, ***prototype***, which defines a "blueprint" or "template" object to use when creating new objects with that constructor. (How can a function, such as the constructor, have properties? In JavaScript, functions are a form of object, so because a function is an object, it can have properties). So we can define a prototype for cats, by assigning properties and methods to the ***prototype*** property of the Cat constructor, and then create many Cats which use that prototype. The code below shows this:

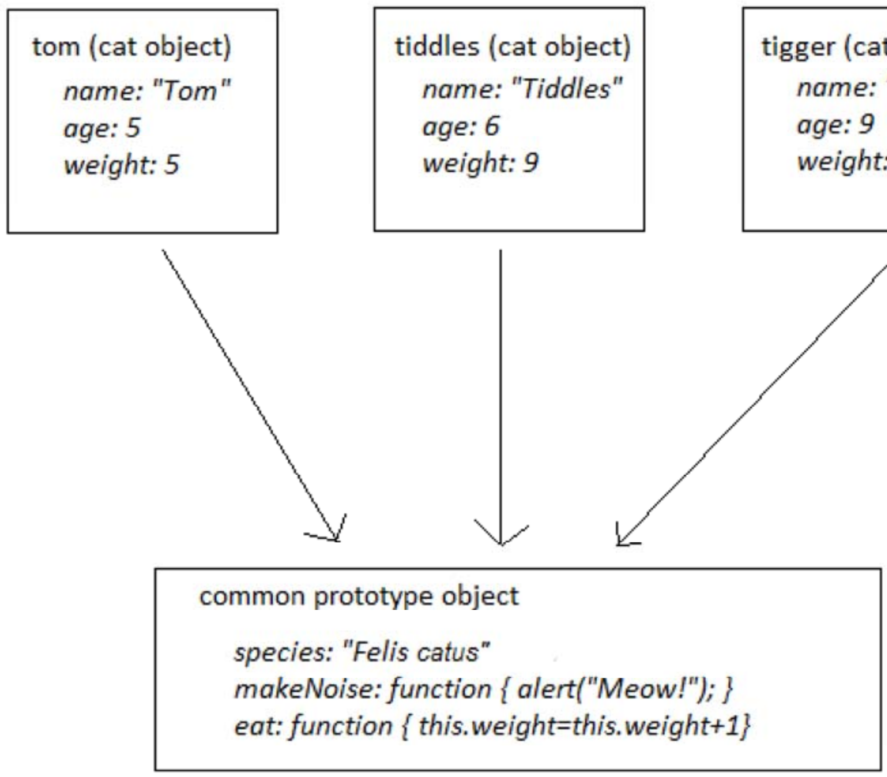
```
function Cat(n,a,w)
{
    this.name=n;
    this.age=a;
    this.weight=w;
}


Cat.prototype.species = "Felis catus";
Cat.prototype.nLegs = 4;
Cat.prototype.makeNoise = function() { alert("Meow!"); }
Cat.prototype.eat = function() { this.weight++; };
```

Note how we use the ***prototype*** property of the constructor function, Cat. The prototype property represents ***the prototype of all objects created with this constructor function***, in other words the prototype property of ***all cats***. So, all cats will have a prototype containing:

- A species property of "Felis catus";
- a nLegs property of 4;
- a ***makeNoise()*** method which displays "Meow!" in an alert box.
- an ***eat()*** method.

It should be noted that the prototype is ***not*** part of the individual cat objects themselves. It is a ***separate*** object which is shared between all objects created with the Cat constructor, i.e. all cat objects. The diagram below illustrates this.



We can prove that the prototype works with the following code:

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

// Setup the prototype in the "global" area, outside any
Cat.prototype.species = "Felis catus";
Cat.prototype.nLegs = 4;
Cat.prototype.makeNoise = function() { alert("Meow!"); }
Cat.prototype.eat = function() { this.weight++; };

// Function to test creating objects with prototypes - m
function testprototypes()
{
    var cat1 = new Cat("Tiddles", 10, 10);
    var cat2 = new Cat("Tigger", 7, 8);

    alert(cat1.species); // "Felis catus"
    alert(cat2.species); // "Felis catus"
    cat1.makeNoise(); // "Meow!"
    cat2.makeNoise(); // "Meow!"
}
```

Note how, even though ***makeNoise()*** and ***species*** are part of the ***prototype***, we can access them directly using simply the object name plus the method or property, for example ***cat1.makeNoise()*** (we don't have to use something like ***cat1.getPrototype().makeNoise()***). The reason for this is given under the discussion on "prototype chaining", in the further notes on prototypes.

Exercise

Return to your car exercise from the objects topic.

1. Create a ***Car*** constructor which takes parameters for the make, model, engine capacity, and top speed, and sets up corresponding properties. It should also set a ***currentSpeed*** property to 0.
2. Set up a ***prototype*** object belonging to the ***Car*** constructor and containing all the car methods you wrote last time, in other words, ***toString()***, ***accelerate()***, ***decelerate()***, and so on.
3. Test it out by creating two car objects which use this prototype, and doing things with them, such as accelerating, decelerating and displaying 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

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

If you finish

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