Creating and using JavaScript objects

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. About this tutorial

Should I take this tutorial?

This tutorial is for programmers who wish to take advantage of object oriented programming (OOP) using JavaScript -- either within the browser or on the server side -- by using custom-built JavaScript objects and their properties and methods.

This tutorial assumes that you already understand JavaScript in general, and that you have at least a familiarity with built-in objects such as `document`, though the basics are reviewed in the tutorial. An understanding of OOP is helpful, but not required, as the basic required concepts are also covered in this tutorial. (References to further information on these subjects are also included in Resources on page 34.)

What is this tutorial about?

Object oriented programming (OOP) is a means for dividing a program into objects with predefined properties and behaviors, known as *methods*. JavaScript is frequently used more as a procedural language, where a script proceeds through a series of steps. However, it is at heart an object oriented language (similar to other object oriented languages, such as Java or C++) which can be used to create objects.

This tutorial explains the very basics of OOP and how to use it within JavaScript. Concepts are covered by using the built-in JavaScript objects many programmers already use. These concepts are then extended to cover custom objects you can create yourself.

This tutorial covers the creation of objects, the nesting of objects within one another as one object becomes the property of another, and the creation of properties and methods (including dynamically created methods). It also explains how one JavaScript object can inherit the properties and methods of another, and how to alter the structure of an object after it has been created.

Tools

This tutorial helps you understand the topic even if you only read through the examples without trying them out. If you do want to try the examples as you go through the tutorial, make sure you have the following tools installed and working correctly:

* A text editor: HTML pages and the JavaScript sections within them are simply text. To create and read them, a text editor is all you need.
* Any browser capable of running JavaScript version 1.2 or above: This includes Netscape Navigator 4.7x and 6.2 (available at *http://browsers.netscape.com/browsers/main.tmpl*) and Microsoft Internet Explorer 5.5 (available from *http://www.microsoft.com/windows/ie/downloads/archive/default.asp*).

About the author

Nicholas Chase has been involved in Web site development for companies including Lucent Technologies, Sun Microsystems, Oracle Corporation, and the Tampa Bay Buccaneers. Nick has been a high school physics teacher, a low-level radioactive waste facility manager, an online science fiction magazine editor, a multimedia engineer, and an Oracle instructor. More recently, he was the Chief Technology Officer of Site Dynamics Interactive Communications in Clearwater, Fla. He is the author of three books on Web development, including *Java and XML From Scratch* (Que). He loves to hear from readers and can be reached at *nicholas@nicholaschase.com* .

Section 2. What is object oriented programming?

Procedural programming

Most programmers learn their craft by creating programs that proceed more or less sequentially: *Do this, do this, do this, then if this is true, do that.* Sometimes these programs branch off into modularized sections such as subroutines and procedures. For the most part, however, data is global in nature, meaning that any section of the program can modify it, and a small change in one section of code can have a profound effect in other parts.

This style of programming, known as *procedural programming*, has been around since the beginning. While there is nothing inherently wrong with it, in many cases there are better ways to get things done.

Object oriented programming

Consider a trip to the grocery store with a very specific list of items to pick up. A procedural program for this trip must attempt to find each item, determine the correct brand and size if it's located, and determine an alternative if it's not. Each item must then be put into the cart, and when the list has been processed, checked out. None of this is particularly daunting, of course, but what about taking into account distractions, such as knocking over a jar of pickles, or running into your first grade teacher? Because of the variety of distractions and environments in which they can occur, adding the capability to handle the distraction and go back to shopping in a procedural language can have a major impact on the application as a whole.

Object oriented programming provides a different way of thinking about this problem. Instead of constructing a series of steps, the programmer creates a series of objects that all know how to behave when various things happen. The `Shopper` object knows how to search for a particular `Item` object, and if that's located, the `Shopper` can initiate the process of putting it into the cart. (It might also know that if the `Item` cannot be located, it should use the `Phone` object to query the `Wife` object for a replacement `Item`.) When the time comes, the `Shopper` object also knows how to work with the `Cashier` object to check out the groceries.

The advantage here is that all the `Shopper` knows about, say, the `Cashier`, is how to interact with it, such as to present it with a series of `Item` objects to check out. If the process for checking out an `Item` changes -- with a new scanning system, for example -- the `Shopper` object isn't affected. All of those changes are taken care of within the `Cashier` object. Similarly, adding the capability for dealing with `Distraction` objects is taken care of within the `Shopper` object, without major impact to the rest of the application.

In its simplest form, OOP is a way of incorporating information and behaviors into objects so that they can interact with each other to get a particular job done.

What is an object?

An object is a collection of data and behaviors, or information about what it is and what it does. Slight differences between languages exist, but in general these are known, respectively, as *properties* and *methods*.

A `Shopper` object might have properties that provide information such as name, amount of available cash, debit card PIN number, and where the car is parked. These properties are typically designated as *private*, which means that they are kept within the object and accessed only through methods.

For example, the `Shopper`'s name may be accessible through the `getName()` method. In traditional OOP, properties generally have `get` and `set` methods to allow for read and write access, as appropriate. (JavaScript allows for the creation of these methods, but it does not allow for the designation of private data, so it is much more common to access properties directly.)

The `Shopper` object may also have methods that provide for specific functionality, such as `searchForItem()` and `useCoupon()`, or methods that provide a way for other objects to interact with it, such as `payBill()`. The methods that are made available for other objects to call make up an object's interface.

Interfaces

An object's *interface* is the collection of methods through which other objects can interact with it. For example, the `Item` object may have `getPrice()`, `getUnitPrice()`, `showCoupons()`, and `addToCart()` methods that the `Shopper` object can call.

Notice, however, that there is no `setPrice()` method available to the `Shopper` object. That makes sense, because the `Shopper` shouldn't be able to set the price. Naturally, this functionality must exist somewhere, perhaps in a method that is only available to the `Manager` object. But in a traditional OOP application, this isolation provides a way to control access to data, limiting the amount of damage that can be done by any one section of the application.

Again, JavaScript doesn't actually provide a way to isolate that data, but you can achieve this effect if you make a habit of only referring to properties through these methods.

Inheritance

One major aspect of OOP is *inheritance*. Inheritance is the ability to base one type of object on another type of object. For instance, an `Item` might be the basis for `SaleItem` and `HeavyItem` objects. Both are `Items`, but each has its own idiosyncrasies.

The advantage of using inheritance is that the base properties and methods of the original object can be retained while any extras or differences can be added. For example, all `Items` may have a price property (or a `getPrice()` method), but a `SaleItem` also needs an `originalPrice` property (or a `getOriginalPrice()` method).

`HeavyItem`, on the other hand, may not need any extra properties or methods, but it requires changes to the `addToCart()` method to accommodate the fact that it needs to be placed at the bottom of the cart, and not to the actual basket.

In both of these cases, the original `Item` methods and properties take precedence unless they are superseded by new properties and methods. For example, when the application calls `addToCart()`, what happens next depends on the object involved. If the `Item` involved is a `HeavyItem`, the application uses the `HeavyItem` version of `addToCart()`, and adds it to the bottom. On the other hand, if it is a `SaleItem`, the application doesn't find a local version of `addToCart()`, so it uses the `Item` version.

Constructors

A *constructor* is a routine that is executed when an object is first created in memory for use by the application. For example, when a new `Cashier` is created (i.e., when another register opens) certain steps need to be taken. The register is activated, the light is turned on, and the `Cashier` announces, "I can take the next person in line ..."

Constructors can be general, as in the previous example, or they can be more specific, as in an `Item` constructor that takes in an identifier such as a Product Look-Up (PLU) and uses it to set the values for properties such as price and unit price.

In some languages, an object may have more than one constructor. For example, the `Cashier` may have been sent to open the first available register, or the `Cashier` may have been sent to open a specific register, in which case the constructor would take the register number as an argument, *overloading* the constructor.

Unfortunately, JavaScript doesn't support overloading. However, it does allow for the building of constructors with some of the same logic to take into account optional arguments.

Classes vs. prototypes vs. objects

Those who have worked with OOP languages -- particularly Java -- in the past may be wondering why the discussion so far has specifically avoided using the word *class* when describing objects.

In traditional OOP, a class is a template for an object. It lists the properties and methods for the object and provides implementations for those methods. The object itself is an *instance* of that class. For example, the objects `joe`, `mary`, and `frank` may be instances of the `Shopper` class. The application deals with these instances, which are built with guidance from the class. Java is one example of a *class-based* language.

In a class-based language, the classes are typically defined when the class is compiled. Once an instance is created within an application, adding or removing properties or methods is impossible.

JavaScript, on the other hand, is a *prototype-based* language. In a prototype-based language, the objects that are based on a prototype remain connected to it. New properties and methods can be added both to an individual instance and to the prototype on which it is based. If the definition of the prototype changes, all objects based on that prototype change as well.

This change in ideology provides a great deal of flexibility. For example, a programmer can add elements to an HTML page programmatically using the built-in JavaScript objects.

Section 3. Using built-in JavaScript objects

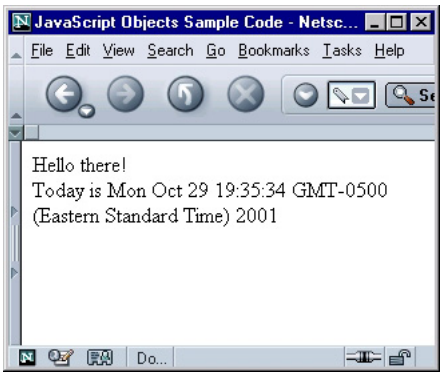
The document object: using methods

If you've worked with JavaScript, you have undoubtedly used objects already, even if they are just the objects that are already built into the language (and the browser). Use of custom objects is identical to use of these built-in objects in many ways, so it helps to look critically at how they are used and how they are structured.

The most common and most basic object used in JavaScript is the `document` itself. In addition to providing a reference point for many other built-in objects (as discussed in the next panel) the `document` object has several useful methods.

The most useful of these is the `write()` method, which enables the output of information to the browser page. This information may be static text, or it may be variables or other object properties or methods. For example:

```
<script type="text/javascript">

document.write('Hello there!')
document.write('<br />')
document.write('Today is
)

</script>
```

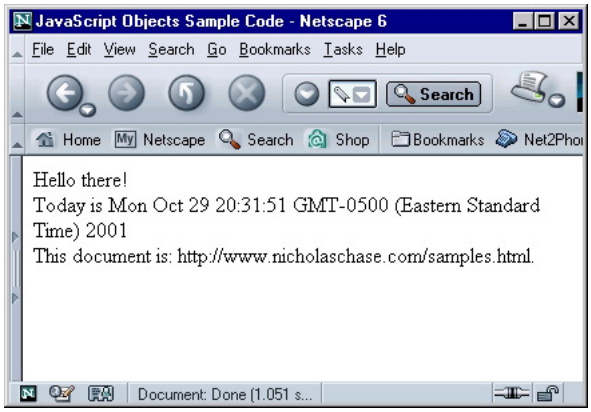


In this example, the `write()` method of the `document` object outputs information to the page. Note the format: object name, dot (.), method name, and arguments in parentheses.

The document object: properties

A script also accesses an object's properties using dot notation. For example:

```
<script type="text/javascript">

...

document.write('This document is: ')
document.write(document.location)
document.write('. <br />')

</script>
```



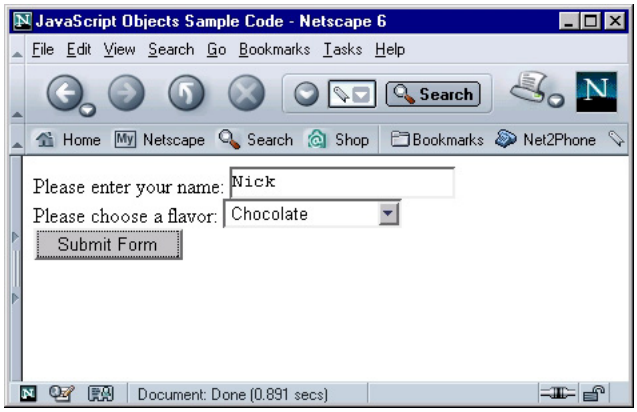
In JavaScript, any information about an object is considered a property, whether it is information about the location of the `document` object (as shown in the above example), or information on how to do something. In other words, methods are also considered properties of an object, because they provide information (in this case, about what the object should do).

An object property can also contain another object. For example, the `document` object is actually a property of the `window` object, and in turn has many properties that are objects. In fact, in a well-formed document, the entire content of the page can be referenced from within the `document` object. The structure of the information depends on the HTML elements involved.

Objects as properties: form elements

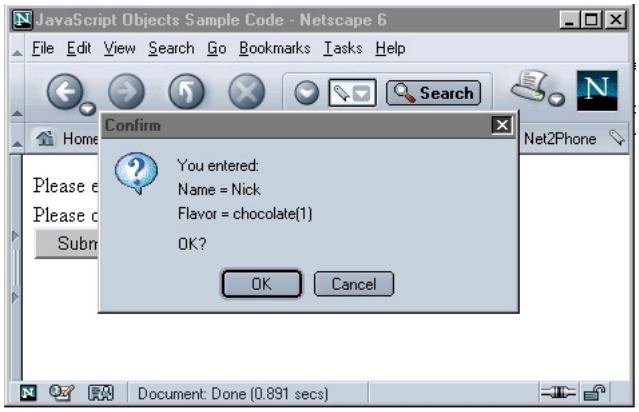
One common use for the chain of objects and properties involves form elements. JavaScript is often used to validate a form before it is submitted. In order to do that, the script must be able to access the various elements that are part of the form. Consider the following form:

```
<form id="sampleForm" name="sampleForm" action="bogus.asp" method="post">

    Please enter your name: <input type="text" name="yourName" /><br />
    Please choose a flavor: <select name="flavor">
                            <option value="no choice">Please Choose ...</option>
                            <option value="chocolate">Chocolate</option>
                            <option value="vanilla">Vanilla</option>
                            </select>
    <br />
    <input type="button" onclick="checkForm()" value="Submit Form" />
 </form>
```



This form, `sampleForm`, is a property of the `document` object, and itself has properties, as seen below:

```
<script type="text/javascript">
    function checkForm(){
       var confirmString = 'You entered: \n Name = '+document.sampleForm.yourName.value + "\n"+
                           'Flavor = '+document.sampleForm.flavor.value +
                           '('+document.sampleForm.flavor.selectedIndex+")" +
                           '\n\n OK? '

       if (confirm(confirmString)) {
           document.sampleForm.submit()
       } else {
           alert('Please adjust your answers.')
       }
    }
 </script>
```



Here the `sampleForm` form object is referenced as a property of the `document` object. The `sampleForm` object itself has properties that are objects with their own properties, such as the `yourName` and `flavor` form element objects.

The `sampleForm` object also has methods, such as the `submit()` method called within the `if-then` statement. Like the other properties, this method can be accessed from the `document` object by walking down the chain of objects.

Arrays of properties: forms

Sometimes the object being referenced is not a simple value or object, but an array of values or objects. For example, a page might have more than one form on it. In that case, the script can access it using the zero-based array of forms, as in:

```
document.forms[0].flavor.value
```

The script can also refer to the form as part of an *associative array*, which uses the name of the object instead of the index:

```
document.forms['sampleForm'].yourName.value
```

Properties can also be accessed as part of an associative array, where the name of the object acts as the index. For example:

```
document.sampleForm['flavor'].value
```

and

```
document.sampleForm.flavor['selectedIndex']
```

This flexibility allows you to decide programmatically what properties to retrieve, then use variables to determine the associative array index.

Creating objects: the Image prototype

All of the objects seen so far have been automatically created by the browser, but objects can also be created explicitly. Not all of these objects have to be custom objects, however. Many objects are already defined, such as the `Date`, `History`, and various form-related objects, such as `Text` and `Button`.

One object that gets a lot of use is the `Image` object. The browser knows that an `Image` object is normally displayed on the page, and it knows how to do that by referencing the `src` property to find out what image to display. In many cases, such as image rollover animations, the browser references an `Image` object that was created through the HTML code on the page. By changing the value of the `src` property, the script changes what appears on the page.

An `Image` object can also be created independent of the HTML. Unless the script explicitly adds it to the page, it won't be displayed, but the browser still tries to load the image referenced by the `src` property. Web authors often use this to preload images into the browser's cache, so they are available instantly when needed, such as for a rollover animation.

To do this, a new object must be created using the `Image` object as its prototype. The `src` property for that object can then be accessed:

```
var preLoader = new Image()
 preLoader.src = 'images/bluto.gif'
 preLoader.src = 'images/pyramid.jpg'
```

```
preLoader.src = 'http://www.example.com/images/plants.jpg'
```

The `preLoader` object is created just like any other JavaScript variable, but its value is set as the returned value from the `Image()` constructor. The `preLoader` then exists as an object with all of the properties and methods of the `Image` prototype, so the script can set the `src` property.

Custom objects are created in much the same way.

Section 4. Creating custom objects

A simple object and constructor

The foundation of any object is the creation of a constructor. A constructor is the code that actually creates a new instance of an object. The constructor can be simple, setting one or more property values. Consider the example of a project tracking application. The constructor for a `Project` object needs to set certain information:

```
function Project() {

    this.name = "Miracle Preso"
    this.manager = "Alex Levine"
    this.status = 0

 }
```

The `this` keyword refers to whichever object is the current object when the function is called. When the script calls the function as a constructor, the `this` keyword refers to the new object being created.

Actually creating the object is just like creating an `Image` object, as in the previous panel:

```
var miracle = new Project()
```

The variable (in this case `miracle`) is now a new `Project` object, just as `preLoader` was a new `Image` object in the previous panel.

Accessing object properties

Once the object has been created, the script can access its properties just as it accessed the properties of built-in objects:

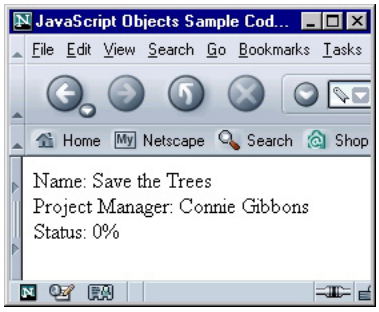
```
 function Project() {

    this.name = "Miracle Preso"
    this.manager = "Alex Levine"
    this.status = 0

 }

 var miracle = new Project()

 document.write('Name: ' +
miracle.name)
 document.write('<br />')
 document.write('Project Manager: ' +
miracle.manager)
 document.write('<br />')
 document.write('Status: ' +
miracle.status)
```

Each of these values has been set within the constructor, so it is accessible via its property name.

Changing object properties

Just as a script accesses object properties through dot notation, it can make modifications to those properties:

```
var miracle = new Project()

miracle.name = "Save the Trees"
miracle.manager = "Connie Gibbons"

document.write('Name: ' + miracle.name)
document.write('<br />')
document.write('Project Manager: ' + miracle.manager)
document.write('<br />')
document.write('Status: ' + miracle.status)
```



Because only `Name` and `Manager` were altered, `Status` retains its original value.

In traditional OOP, it's customary to use methods to get and set property values, but direct access is common in JavaScript.

Optional constructor arguments

JavaScript does not, unfortunately (or fortunately, depending upon whom you ask), support overloading of functions, so constructors must be built with all possible combinations in mind.

The most common permutations involve using arguments as values if they exist, and using default values if they don't. A constructor can accomplish this using `if-then` statements, as in:

```
function Project(projName, projMgr, projStatus) {

    if (projName == null) {
       this.name = "Miracle Preso"
    } else {
       this.name = projName
    }
    if (projMgr == null) {
       this.manager = "Alex Levine"
    } else {
       this.manager = projMgr
    }
    if (projStatus == null) {
       this.status = "0%"
    } else {
       this.status = projStatus
    }

 }
```

Optional constructor arguments (continued)

It can be much more convenient, however, to use "or" notation. Consider the following expression:

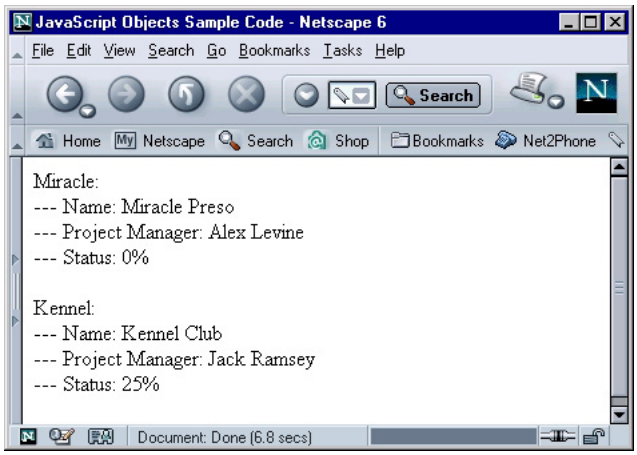
```
expr1 || expr2
```

If `expr1` is non-null, the "or" condition is satisfied, and that is the returned value. On the other hand, if `expr1` is null, the script goes on to evaluate `expr2`. If it is not null, then that is the value returned.

Translating this concept to the `Project()` constructor:

```
function Project(projName, projMgr, projStatus) {
    this.name = projName || "Miracle Preso"
    this.manager = projMgr || "Alex Levine"
    this.status = projStatus || "0%"
 }
```

```
var miracle = new Project()
var kennel = new Project('Kennel Club', 'Jack Ramsey', '25%')

document.write('Miracle: <br />')
document.write('--- Name: ' + miracle.name + '<br />')
document.write('--- Project Manager: ' + miracle.manager + '<br />')
document.write('--- Status: ' + miracle.status + '<br /><br />')

document.write('Kennel: <br />')
document.write('--- Name: ' + kennel.name + '<br />')
document.write('--- Project Manager: ' + kennel.manager + '<br />')
document.write('--- Status: ' + kennel.status + '<br /><br />')
```



In this example, two objects are created from the same constructor. The `miracle` object didn't provide arguments, and gets the default values. The `kennel` object, on the other hand, gets the arguments as property values.

Adding methods

Strictly speaking, a JavaScript object method is simply a property that contains a function. When the property is accessed, the function executes.

The simplest methods are used to change the properties of an object. In JavaScript, it is common to change these properties directly, but they can also be changed through methods. Consider a method that's used to set the status of the `Project`:

```
                    function setStatus(newStatus) {
    this.status = newStatus
 }
```

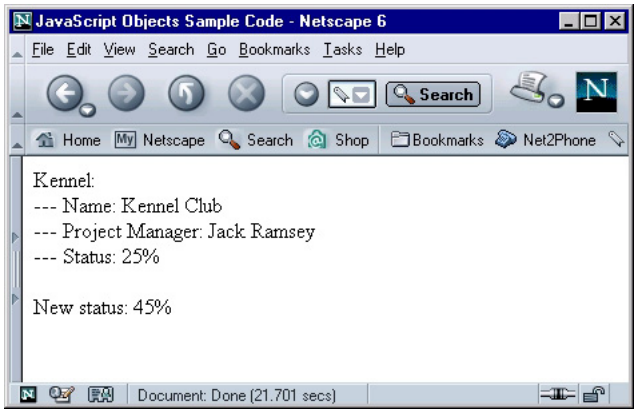
To create the method, assign the function to a property:

```
function Project(projName, projMgr, projStatus) {
    this.name = projName || "Miracle Preso"
    this.manager = projMgr || "Alex Levine"
    this.status = projStatus || "0%"

    this.setStatus = setStatus
 }
```

Note that the names don't have to be the same, though it can make the code easier to understand. Notice also that although the code refers to a function, it doesn't have parentheses after it (as in `setStatus()`).

To call the method, simply reference the property, along with any parameters:

```
var kennel = new Project('Kennel Club', 'Jack Ramsey', '25%')

 document.write('Kennel: <br />')
 document.write('--- Name: ' + kennel.name + '<br />')
 document.write('--- Project Manager: ' + kennel.manager + '<br />')
 document.write('--- Status: ' + kennel.status + '<br /><br />')

 kennel.setStatus('45%')

 document.write('New status: '+kennel.status)
```



Methods can also serve much more complex purposes, but their construction and access are the same in any case.

Array properties

In some situations, one property holds multiple pieces of data. For example, a project may have multiple team members. To add them as individual items in a single property, use an array:

```
function Project(projName, projMgr, projStatus) {
    this.name = projName || "Miracle Preso"
    this.manager = projMgr || "Alex Levine"
    this.status = projStatus || "0%"
    this.team = ["John Smith", "Mary Jones", "August McKendrick"]

    this.setStatus = setStatus
 }
```

To access the individual values, add an index value to the property:

```
var kennel = new Project('Kennel Club', 'Ja                    25%')

 document.write(kennel.team[0])
 document.write('<br />')
 document.write(kennel.team[1])
 document.write('<br />')
 document.write(kennel.team[2])
```

Modifying an object

One of the advantages of a prototype-based language over a class-based language is the ability to change not only an object, but an entire type of object, after it has been created. For example, suppose an auditor needs to be added to projects because they have been stalled for a certain amount of time. Adding a property to a single object is easy. Simply reference it and assign it a value:

```
kennel.auditor = "Janine Gottfried"

document.write('Miracle Auditor: ' + miracle.auditor)
document.write('<br />')
document.write('Kennel Auditor: ' + kennel.auditor)
```

Because the value has been specifically assigned to the `kennel` object, the `miracle` object is not affected.

But what if an overall lack of progress means that auditors should be assigned to all projects? If the change is applied to the prototype of the `kennel` object, all objects based on that prototype will be affected:

```
Project.prototype.auditor = "Janine Gottfried"

document.write('Miracle Auditor: ' + miracle.auditor)
document.write('<br />')
document.write('Kennel Auditor: ' + kennel.auditor)
```

Because the change is applied to the `Project` prototype, it affects all `Project` objects.

JavaScript also allows for the deletion of properties. For example:

```
delete kennel.auditor
 delete Project.prototype.auditor
```

The rules regarding propagation of deleting a property are the same as those for adding a property.

Section 5. Using inheritance

Adding inherited objects

The running project tracking application example has established a basic type of object, the `Project`. This is, of course, an extremely general object. Projects usually have requirements specific to themselves, or at least specific to the type of project at hand.

Take, for example, three types of interactive media projects: a Web site, a CD-ROM, and a kiosk. All three have the same requirements as a general project: a name, a project manager, a status, and a team of employees working on it. Each also has specific requirements. For example, a Web site project also has a base URL, a CD-ROM has a target platform, and a kiosk has a target input device, such as a keyboard or a touch screen.

All of these objects are, however, `Project` objects, so it makes sense to extend the `Project` object when creating them.

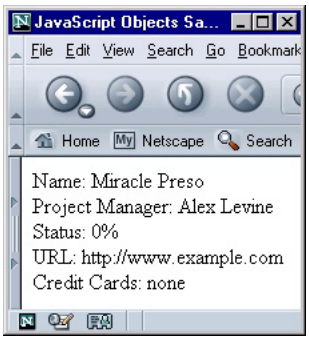
Using a prototype

The first step in creating new objects is to determine their prototype. In absence of other declarations, JavaScript uses the generic `Object` object, but the `Project` object can be explicitly set as the prototype for the new objects:

```
WebSite.prototype = new Project

function WebSite(webSiteURL) {
    this.URL = webSiteURL || "http://www.example.com"
}
```

In this way, when the script creates a new `WebSite` object, it has not only the original properties of the `Project` object, but also the additional `URL` property:

```
var kennel = new WebSite()

document.write('Name: ' +
kennel.name)
document.write('<br />')
document.write('Project Manager: ' +
kennel.manager)
document.write('<br />')
document.write('Status: ' +
kennel.status)
document.write('<br />')
document.write('URL: ' + kennel.URL)
```



Even though no `name`, `manager`, or `status` properties are defined within the `WebSite()`

constructor, they exist because they are inherited from `Product`.

The "is-a" relationship

In OOP, it is often convenient to know whether one object is descended from another. This is known as the "is-a" relationship, as in "`kennel` is a `WebSite`" so "`kennel` is a `Project`."

Suppose the project required the further breakdown of `WebSite` objects into type, such as `CommerceSites`:

```
function CommerceSite(creditCards){
   this.credit = creditCards || "none"
}

CommerceSite.prototype = new WebSite

var shawlsAreUs = new CommerceSite()

document.write('Name: ' + shawlsAreUs.name)
document.write('<br />')
document.write('Project Manager: ' + shawlsAreUs.manager)
document.write('<br />')
document.write('Status: ' + shawlsAreUs.status)
document.write('<br />')
document.write('URL: ' + shawlsAreUs.URL)
document.write('<br />')
document.write('Credit Cards: ' + shawlsAreUs.credit)
```

(This object clearly needs a way to access the parent constructors; this is discussed in the next panel.)

The hierarchy of these objects would be `Object` -- > `Project` -- > `Website` -- > `CommerceSite` -- > `shawlsareus`. To see this programmatically, access the `__proto__` property. (That's with two "_" characters at both the start and end.)

```
if (shawlsAreUs.__proto__ == CommerceSite.prototype) {
   document.write('shawlsAreUs is a CommerceSite <br />')
} else {
   document.write('shawlsAreUs is not a CommerceSite <br />')
}
if (shawlsAreUs.__proto__ == WebSite.prototype) {
   document.write('shawlsAreUs is a WebSite <br />')
```

```
} else {
   document.write('shawlsAreUs is not a WebSite <br />')
}
```

Notice that `shawlsAreUs` does not appear to be a `WebSite` object, even though `CommerceSite` is descended from `WebSite`. This is because the `__proto__` property contains a reference to the actual object. To move up the chain:

```
if (shawlsAreUs.__proto__.__proto__ == WebSite.prototype) {
   document.write('shawlsAreUs is a WebSite <br />')
} else {
   document.write('shawlsAreUs is not a WebSite <br />')
}
```

Note that the `__proto__` property is *not* supported by Internet Explorer 5.x, which limits its usefulness at this time.

Accessing the "parent" constructor

In the creation of a `CommerceSite` object, the object does inherit all of the properties of `WebSite` and `Project` objects, but there appears to be no way to set those values. One solution is to set them within the `commerceSite` constructor:

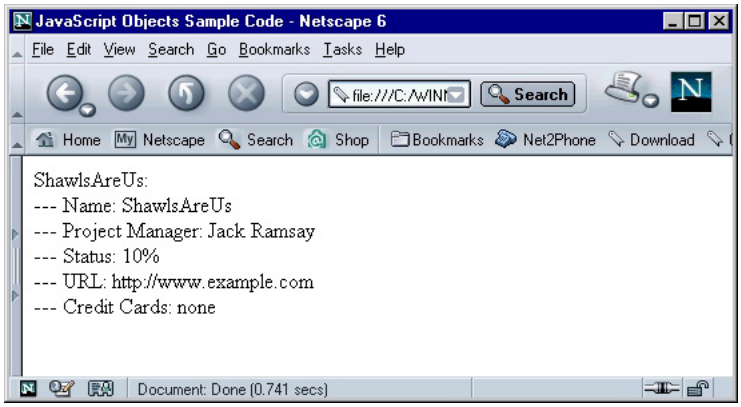
```
function CommerceSite(projName, projMgr, projStatus, projURL, projCreditCards){
    this.name = projName || "Commerce Site"
    this.manager = projMgr || "Alex Levine"
```

```
    this.status = projStatus || "0%"
    this.URL = projURL || "http://www.soap-to-shawls.com"
    this.credit = projCreditCards || "none"
 }
```

Unfortunately, this defeats the whole purpose of inheritance. That's not to say it's not useful, of course. There may be times when program requirements demand that the inherited property be overridden by a local version. The goal here, however, is to use the original constructor to set the values for this object.

The idea is to execute the constructor in such a way that it is tied to this particular object instance. The solution is to execute the constructor explicitly as a method:

```
function CommerceSite(projName, projMgr, projStatus, projURL, projCreditCards){
    this.projectBase = Project
    this.projectBase(projName, projMgr, projStatus)
    this.webSiteBase = WebSite
    this.webSiteBase(projURL)
    this.credit = projCreditCards || "none"
 }

 CommerceSite.prototype = new WebSite;

 var shawlsAreUs = new CommerceSite('ShawlsAreUs', 'Jack Ramsay', '10%')

 document.write('ShawlsAreUs: <br />')
 document.write('--- Name: ' + shawlsAreUs.name + '<br />')
 document.write('--- Project Manager: ' + shawlsAreUs.manager + '<br />')
 document.write('--- Status: ' + shawlsAreUs.status + '<br />')
 document.write('--- URL: ' + shawlsAreUs.URL + '<br />')
 document.write('--- Credit Cards: ' + shawlsAreUs.credit + '<br />')
```



In this way, the constructor functions are explicitly executed in relation to this object, with the appropriate arguments passed. Note that this is *not* a substitute for creating inheritance relationships using the `prototype` attribute. While it may provide the appropriate properties, simply calling the constructor does not create inheritance.

Modifying inherited objects

As seen in Modifying an object on page 18 , one of the advantages of a prototype-based language is the ability to modify an object after it has been created. Also as discussed,

modifying an object's prototype also modifies the object. This comes into play with inheritance, as well. For example, a change to the `Project` prototype propagates down through `WebSite` and `CommerceSite`:

```
                    var miracle = new Project()
var kennel = new WebSite()
var shawlsAreUs = new CommerceSite()

document.write('Miracle host: ' + miracle.host)
document.write('<br />')
document.write('Kennel host: ' + kennel.host)
document.write('<br />')
document.write('Shawls host: ' + shawlsAreUs.host)
document.write('<br />')

WebSite.prototype.host = "MyHostingCompany"

document.write('<br />')
document.write('Miracle host: ' + miracle.host)
document.write('<br />')
document.write('Kennel host: ' + kennel.host)
document.write('<br />')
document.write('Shawls host: ' + shawlsAreUs.host)
```

Because the property is added to the `WebSite` object prototype, it doesn't propagate back to the `Project` objects, but it does propagate forward to the `WebSite` and `CommerceSite` objects.

Inherited values and scope

It should be noted that there is a difference between properties that are local to an object -- either because they were defined for an instance or within the object's constructor -- and those that are basic to the object prototype.

When an application requests an object's property, the return value is going to depend heavily on how the object was built. If there is a value specific to that object or class, that is returned first. If not, JavaScript travels up the inheritance chain until it finds a value (or runs out of objects to check). For example:

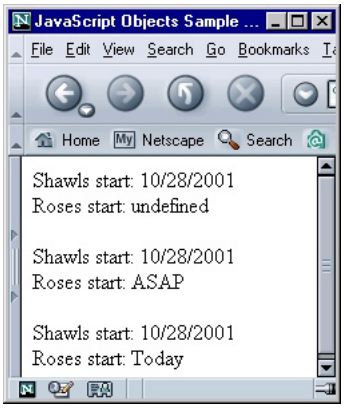
```
function CommerceSite(projName,
                      projMgr,
                      projStatus,
                      projURL,
                      projCreditCards){

   this.projectBase = Project
   this.projectBase(projName, projMgr, projStatus)
   this.webSiteBase = WebSite
   this.webSiteBase(projURL)
   this.credit = projCreditCards || "none"
   this.start = "10/28/2001"
}

var shawlsAreUs = new CommerceSite()
var bedOfRoses = new Project()

document.write('Shawls start: ' + shawlsAreUs.start)
document.write('<br />')
document.write('Roses start: ' + bedOfRoses.start)
document.write('<br />')

Project.prototype.start = 'ASAP'

document.write('<br />')
document.write('Shawls start: ' + shawlsAreUs.start)
document.write('<br />')
document.write('Roses start: ' + bedOfRoses.start)
```

Because `CommerceSite` defines the `start` property within the constructor, it becomes local to any `CommerceSite` objects. As a result, even though a `start` property was added through the `Project` prototype, the local value took precedence. Because `bedOfRoses` had no such "local" declaration, the change took effect.

It should be noted that even if the property is created globally, as it is here, locally changing the value creates a local value that overrides the prototype value:

```
...
document.write('Roses start: ' + bedOfRoses.start)
document.write('<br />')

bedOfRoses.start = 'Today'
Project.prototype.start = 'Tomorrow'

document.write('<br />')
```

```
document.write('Shawls start: ' + shawlsAreUs.start)
document.write('<br />')
document.write('Roses start: ' + bedOfRoses.start)
```



To ensure the ability to globally alter properties, be certain to declare them within the prototype and to always use the prototype to change them.

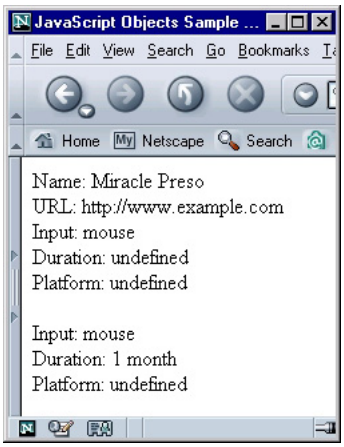
Simulating multiple inheritance

Some languages allow an object to inherit from multiple ancestors, drawing properties from all of them. JavaScript doesn't actually allow this -- the `prototype` property can hold only one object -- but it is possible to simulate some of the effect.

Because JavaScript objects are created by executing constructors, and because those ancestor constructors can be referenced directly within an object constructor, executing multiple constructors can *simulate* multiple inheritance even if they don't actually create it. Consider this example:

```
function CommerceSite(projName,
                      projMgr,
                      projStatus,
                      projURL,
                      projCreditCards){

   this.projectBase = Project
   this.projectBase(projName, projMgr, projStatus)
   this.webSiteBase = WebSite
   this.webSiteBase(projURL)
   this.kioskBase = Kiosk
   this.kioskBase('mouse')
   this.credit = projCreditCards || "none"
}

function Kiosk(projInput) {
   this.inputDevice = projInput || "touchscreen"
}

CommerceSite.prototype = new WebSite

var houseOfFish = new CommerceSite()
```

```
document.write('Name: ' + houseOfFish.name)
document.write('<br />')
document.write('URL: ' + houseOfFish.URL)
document.write('<br />')
document.write('Input: ' + houseOfFish.inputDevice)
document.write('<br />')
document.write('Duration: ' + houseOfFish.duration)
document.write('<br />')
document.write('Platform: ' + houseOfFish.platform)
document.write('<br />')

WebSite.prototype.duration = '1 month'
Kiosk.prototype.platform = 'Linux'

document.write('<br />')
document.write('Input: ' + houseOfFish.inputDevice)
document.write('<br />')
document.write('Duration: ' + houseOfFish.duration)
document.write('<br />')
document.write('Platform: ' + houseOfFish.platform)
```



Because the `Project`, `WebSite`, and `Kiosk` constructors are all executed as part of the `CommerceSite` constructor, the `houseOfFish` object gets all of their properties. But because the `Kiosk` prototype isn't in the inheritance chain, adding a property to it doesn't affect the `houseOfFish` object, even though adding one to the `WebSite` prototype does (because `CommerceSite` inherits from `WebSite`).

Section 6. Objects as properties

Adding other objects

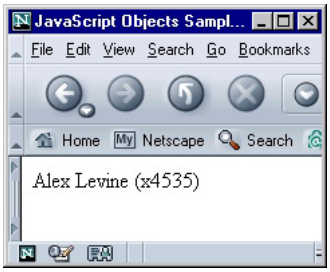
Just as the `document` object is a property of the `window` object, an application can use objects as the values of properties for objects that it creates.

For example, consider the definition of a `Project` object:

```
function Project(projName, projMgr, projStatus) {
    this.name = projName || "Miracle Preso"
    this.manager = projMgr || "Alex Levine"
    this.status = projStatus || "0%"
    this.team = ["John Smith", "Mary Jones", "August McKendrick"]

    this.setStatus = setStatus
 }
```

All of the people listed could also be defined as objects, with properties and methods of their own. For example, the application might define a `Worker` object, which is inherited by `Employee`, `Manager`, and `Contractor` objects. For simplicity's sake, this tutorial simply uses `Employees`:

```
function Employee(empFirstName, empLastName, empPhone) {
   this.firstName = empFirstName || "John"
   this.lastName = empLastName || "Doe"
   this.phone = empPhone || "none"
}

var alex = new Employee('Alex', 'Levine', 'x4535')

document.write(alex.firstName + ' '+alex.lastName)
document.write(' (' + alex.phone + ')')
```



Adding an object as a property

To add an object as a property for another object, simply reference it like any other value. For example, the `Project` object can be modified so that instead of simply taking a name for the project manager, it takes an object:

```
function Project(projName, projMgr, projStatus) {
    this.name = projName || "Miracle Preso"
    this.manager = projMgr || alex
```

```
    this.status = projStatus || "0%"
    this.team = ["John Smith", "Mary Jones", "August McKendrick"]

    this.setStatus = setStatus
}

var geena = new Employee('Geena', 'Tungsten', 'x2322')

var houseOfFish = new Project('House Of Fish', geena, '0%')
```

The actual definition itself hasn't changed much. The constructor still looks for the `projMgr` argument to populate the `manager` property, but now that value is an object instead of a string.

Accessing an object as a property

Accessing an object that has been assigned as a property of another object involves an understanding of the structure of the objects. For example, in the previous example, an employee object, namely `geena`, was set as the `manager` property of the `houseOfFish` object. This means that `houseOfFish.manager` and `geena` are equivalent. So the `manager` property of the `houseOfFish` object is `geena`, which also has `firstName`, `lastName`, and `phone` properties. In order to access those properties, you need to create a chain of objects:

```
document.write('Project Manager: <br />')
document.write(houseOfFish.manager.firstName + ' ')
document.write(houseOfFish.manager.lastName)
document.write('  (' + houseOfFish.manager.phone + ')')
```

Like the objects in an HTML page, objects can be chained across multiple levels in this way, so that the `manager` properties (such as `phone`) can also be objects.

Functions as objects

In actuality, each object method that has been shown in this tutorial has been an object, albeit a special type of object.

The `Function` object constructor takes two parameters*: the name of any arguments to be passed to the function, and the code for the function itself. For example:

```
var replaceManager =
```

```
    new Function('defVal',
                 'document.write("Default manager: "+defVal)')

replaceManager(geena.firstName)
```

Understanding the object nature of functions allows you to create them programmatically at runtime, because they are simply text passed as an argument to the `Function` constructor. Granted, it would be inconvenient to create a particularly long function this way, but this ability creates significant flexibility.

* The actual definition of the `Function` object allows much more flexibility than this, but for simplicity's sake, this tutorial will stay with just these two parameters.

Dynamic methods

The ability to create a function, and thus a method, dynamically can make object definition all that much easier. For example, suppose it were policy to designate a backup project manager to be assigned in the event that the current project manager leaves without a replacement. There are several ways to code this.

One way is to create a property for the backup project manager, then create a method that replaces the project manager with the backup manager if necessary. There's nothing terribly wrong with that approach, but the same thing can be accomplished using dynamic methods:

```
function Project(projName, projMgr, projStatus, projDefMgr) {
    this.name = projName || "Miracle Preso"
    this.manager = projMgr || alex
    this.status = projStatus || "0%"
    this.team = ["John Smith", "Mary Jones", "August McKendrick"]

    this.replaceManager = new Function('newMgr', 'this.manager = newMgr || '+projDefMgr)
 }

 var geena = new Employee('Geena', 'Tungsten', 'x2322')

 var houseOfFish = new Project('House Of Fish', null, '0%', 'alex')
```

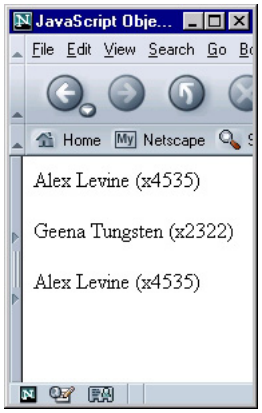
Notice that the default manager (`projDefMgr`) is fed to the constructor as a string, and not as an object, because the goal is to create a text argument for `Function` of:

```
this.manager = newMgr || alex
```

Passing `projDefMgr` as an object would cause the script to attempt to combine an object and a string, which causes an error.

The end result is that the object now has a default manager to use if none is provided to the `replaceManager()` method:

```
document.write(houseOfFish.manager.firstName + ' ')
 document.write(houseOfFish.manager.lastName)
 document.write('  (' + houseOfFish.manager.phone + ')')
 document.write('<br /><br />')

 houseOfFish.replaceManager(geena)

 document.write(houseOfFish.manager.firstName + ' ')
 document.write(houseOfFish.manager.lastName)
 document.write('  (' + houseOfFish.manager.phone + ')')
 document.write('<br /><br />')

 houseOfFish.replaceManager()

 document.write(houseOfFish.manager.firstName + ' ')
 document.write(houseOfFish.manager.lastName)
 document.write('  (' + houseOfFish.manager.phone + ')')
```



The `houseOfFish` object was created without a manager, so the default manager, `alex`, was used. Later, `replaceManager()` changed the value of the `manager` property to point to the `geena` object. Finally, `replaceManager` executed one more time, this time using the default manager, `alex`.

Arrays of objects

Adding an array of objects as an object property is just like adding an array of strings or other values. For example:

```
                    var alex = new Employee('Alex', 'Levine', 'x4535')
var geena = new Employee('Geena', 'Tungsten', 'x2322')
var art = new Employee('Art', 'Franklin', 'x4223')
var daniel = new Employee('Daniel', 'Gardst', 'x2234')

function Project(projName, projMgr, projStatus, projDefMgr) {
```

```
   this.name = projName || "Miracle Preso"
   this.manager = projMgr || alex
   this.status = projStatus || "0%"
   this.team = [geena, art, daniel]

   this.replaceManager = new Function('newMgr', 'this.manager = newMgr || '+projDefMgr)
}
```

A script can also pass the array of objects directly:

```
function Project(projName, projMgr, projStatus, projDefMgr, projTeam) {
   this.name = projName || "Miracle Preso"
   this.manager = projMgr || alex
   this.status = projStatus || "0%"
   this.team = projTeam

   this.replaceManager = new Function('newMgr', 'this.manager = newMgr || '+projDefMgr)
}

var houseOfFish = new Project('House Of Fish', null, '0%', 'alex',  [geena, art, daniel])
```

Either way, the `team` property now consists of an array of objects.

Accessing an object in an array

To access an object that is part of an array, include the array index:

```
document.write(houseOfFish.team[0].firstName + ' ')
document.write(houseOfFish.team[0].lastName)
document.write('  (' + houseOfFish.team[0].phone + ')')

document.write('<br />')

document.write(houseOfFish.team[1].firstName + ' ')
document.write(houseOfFish.team[1].lastName)
document.write('  (' + houseOfFish.team[1].phone + ')')

document.write('<br />')

document.write(houseOfFish.team[2].firstName + ' ')
document.write(houseOfFish.team[2].lastName)
document.write('  (' + houseOfFish.team[2].phone + ')')
```

In this case, the `team` property represents an array of objects, so for example:

```
houseOfFish.team[1]
```

corresponds to the `art` object. That means:

```
houseOfFish.team[1].lastName
```

returns "Franklin," the `lastName` property of the `art` object.

Section 7. JavaScript objects summary

Summary

Objects are, in many ways, the foundation of JavaScript. This prototype-based object oriented language not only allows you to work with built-in objects such as `document` and the objects that are its properties, but also custom objects.

Custom objects are created using constructors. Objects can inherit properties and methods from each other by determining the prototype for an object. Properties can consist of simple values, or of other objects, including functions.

Any object operation carried out during the course of working with the built-in objects, such as executing methods, assigning objects to object properties, and assigning arrays of objects to object properties, can be used with custom objects, allowing for a traditional object oriented programming approach to client-side and server-side JavaScript programming.

Resources

For good information on object oriented programming in general and JavaScript in particular, see these resources:

* Read *What is Object-Oriented Software?* , by Terry Montlick.
* Find object oriented programming information at *Cetus Links: Architecture and Design* .
* Read a *JavaScript Tutorial for Programmers* by Aaron Weiss.
* Read *A Primer on JavaScript Arrays* by Danny Goodman.
* Read *Creating Robust Functions* .
* Read *Object Hierarchy and Inheritance in JavaScript* , on the Netscape site.
* Read *The prototype object of JavaScript 1.1* .
* Read *Creating custom objects in JavaScript* at Website Abstraction.
* Read *All About JavaScript* by Robert W. Husted for a look at how JavaScript on the client compares to JavaScript on the server.
* For a variety of JavaScript documentation, including reference manuals for JavaScript 1.5, read *Netscape's JavaScript Documentation* .
* Explore a wealth of information at *Cetus Links: Object-Oriented Language: JavaScript / ECMAScript* .

Downloads

* Download an *HTML file with the sample code* presented in this tutorial.
* Download *IBM Web Browser for OS/2* .
* Download *Microsoft Internet Explorer 5.5* , *Internet Explorer 6* , or *Internet Explorer 5.0 for Macintosh* .
* Download *Netscape 6* , with improved compliance over earlier versions.

Feedback

We welcome your feedback on this tutorial -- let us know what you think. We look forward to hearing from you!

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The Toot-O-Matic tool is a short Java program that uses XSLT stylesheets to convert the XML source into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML.