Phil Ballard

Sixth Edition
New coverage of
ECMAScript 6

Sams **Teach Yourself**

JavaScript™

in **24 Hours**

SAMS

Phil Ballard

Sams **Teach Yourself**

JavaScript®

Sixth Edition

in 24 Hours

SAMS 800 East 96th Street, Indianapolis, Indiana, 46240 USA

Notice that in the `<head>` section of the page, you create the object `myNewObject` and assign it the property `info` and the method `showInfo`, as described earlier.

Loading this page into your browser, you are confronted with three buttons.

Clicking on the first button makes a call to the `showInfo` method of the newly created object:

```
<input type="button" value="Good showInfo Call" onclick="myNewObject.showInfo()" />
```

As you would hope, the value of the `info` property is passed to the `alert()` dialog, as shown in Figure 8.1.

FIGURE 8.1
The `info` property is correctly called

The second button attempts to make a call directly to the function `myFunc()`:

```
<input type="button" value="myFunc Call" onclick="myFunc()" />
```

Because `myFunc()` is a method of the global object (having been defined without reference to any other object as parent), it attempts to pass to the `alert()` dialog the value of a nonexistent property `window.info`, with the result shown in Figure 8.2.

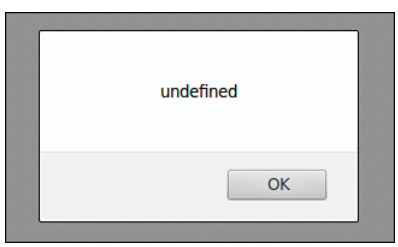

FIGURE 8.2
The global object has no property called `info`

Finally, your third button attempts to call `showInfo` without reference to its parent object:

```
<input type="button" value="Bad showInfo Call" onclick="showInfo()" />
```

Because the method does not exist outside the object `myNewObject`, JavaScript reports an error, as shown in Figure 8.3.

NOTE

We'll talk more about showing JavaScript errors using your browser's JavaScript Console or Error Console later in the book.

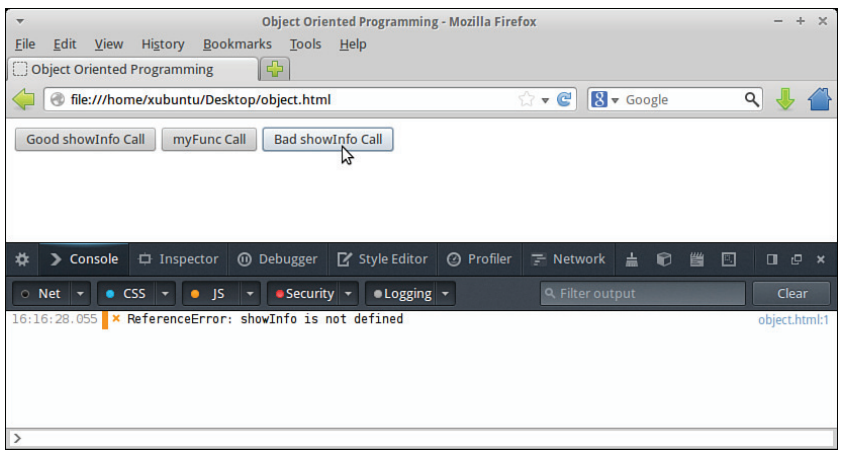

FIGURE 8.3
JavaScript reports that `showInfo` is not defined

Anonymous Functions

There is a more convenient and elegant way to assign a value to your object's `showInfo` method, without having to create a separate, named function and then later assign it by name to the required method. Instead of this code:

```
function myFunc() {
        alert(this.info);
    };
myNewObject.showInfo = myFunc;
```

you could simply have written the following:

```
myNewObject.showInfo = function() {
    alert(this.info);
}
```

Because you haven't needed to give a name to your function prior to assigning it, this technique is referred to as using an *anonymous function*.

By using similar assignment statements you can add as many properties and methods as you need to your instantiated object.

TIP

JavaScript offers a further way to create a direct instance of an object; the technique uses JSON (JavaScript Object Notation). It isn't covered here, as we explore JSON in detail in Hour 10, "Meet JSON."

Using a Constructor Function

Directly creating an instance of an object is fine if you think you'll only need one object of that type. Unfortunately, if you need to create another instance of the same type of object, you'll have to go through the same process again—creating the object, adding properties, defining methods, and so on.

TIP

An object with only one global instance is sometimes called a *singleton* object. These objects are useful sometimes; for example, a user of your program might have only one associated `user-Profile` object, perhaps containing his or her user name, URL of last page viewed, and similar properties.

A better way to create objects that will have multiple instances is by using an *object constructor function*. An object constructor function creates a kind of "template" from which further objects can be instantiated.

Take a look at the following code. Instead of using `new Object()`, you first declare a function, `myObjectType()`, and in its definition you can add properties and methods using the `this` keyword.

```
function myObjectType(){
    this.info = 'I am a shiny new object';
    this.showInfo = function(){
        alert(this.info);  // show the value of the property info
    }
     this.setInfo = function (newInfo) {
        this.info = newInfo; // overwrite the value of the property info
    }
}
```

In the preceding code you added a single property, `info`, and two methods: `showInfo`, which simply displays the value currently stored in the `info` property, and `setInfo`. The `setInfo` method takes an argument, `newInfo`, and uses its value to overwrite the value of the `info` property.

Instantiating an Object

You can now create as many instances as you want of this object type. All will have the properties and methods defined in the `myObjectType()` function. Creating an object instance is known as *instantiating* an object.

Having defined your constructor function, you can create an instance of your object simply by using the constructor function:

```
var myNewObject = new myObjectType();
```

NOTE

Note that this syntax is identical to using `new Object()`, except you use your purpose-designed object type in place of JavaScript's general-purpose `Object()`. In doing so, you instantiate the object complete with the properties and methods defined in the constructor function.

Now you can call its methods and examine its properties:

```
var x = myNewObject.info // x now contains 'I am a shiny new object'
myNewObject.showInfo();  // alerts 'I am a shiny new object'
myNewObject.setInfo("Here is some new information");  // overwrites the info
property
```

Creating multiple instances is as simple as calling the constructor function as many times as you need to:

```
var myNewObject1 = new myObjectType();
var myNewObject2 = new myObjectType();
```

Let's see this in action. The code in Listing 8.2 defines an object constructor function the same as the one described previously.

Two instances of the object are instantiated; clearly, both objects are initially identical. You can examine the value stored in the `info` property for each object by clicking on one of the buttons labeled Show Info 1 or Show Info 2.

A third button calls the `setInfo` method of object `myNewObject2`, passing a new string literal as an argument to the method. This overwrites the value stored in the `info` property of object `myNewObject2`, but of course leaves `myNewObject` unchanged. The revised values can be checked by once again using Show Info 1 and Show Info 2.

LISTING 8.2 Creating Objects with a Constructor Function

```
<!DOCTYPE html>
<html>
<head>
    <title>Object Oriented Programming</title>
    <script>
        function myObjectType(){
            this.info = 'I am a shiny new object';
            this.showInfo = function(){
                alert(this.info);
            }
            this.setInfo = function (newInfo) {
                this.info = newInfo;
            }
        }
        var myNewObject1 = new myObjectType();
        var myNewObject2 = new myObjectType();
    </script>
</head>
<body>
    <input type="button" value="Show Info 1" onclick="myNewObject1.showInfo()" />
    <input type="button" value="Show Info 2" onclick="myNewObject2.showInfo()" />
    <input type="button" value="Change info of object2"
    ➥onclick="myNewObject2.setInfo('New Information!')" />
</body>
</html>
```

Using Constructor Function Arguments

There is nothing to stop you from customizing your objects at the time of instantiation, by passing one or more arguments to the constructor function. In the following code, the definition of the constructor function includes one argument, `personName`, which is assigned to the `name` property by the constructor function. As you instantiate two objects, you pass a name as an argument to the constructor function for each instance.

```
function Person(personName){
    this.name = personName;
    this.info = 'I am called ' + this.name;
    this.showInfo = function(){
        alert(this.info);
    }
}
var person1 = new Person('Adam');
var person2 = new Person('Eve');
```

TIP

You can define the constructor function to accept as many or as few arguments as you want:

```
function Car(Color, Year, Make, Miles) {
    this.color = Color;
    this.year = Year;
    this.make = Make;
    this.odometerReading = Miles;
    this.setOdometer = function(newMiles) {
         this.odometerReading = newMiles;
    }
var car1 = new Car("blue", "1998", "Ford", 79500);
var car2 = new Car("yellow", "2004", "Nissan", 56350);
car1.setOdometer(82450);
```

Extending and Inheriting Objects Using `prototype`

A major advantage of using objects is the capability to reuse already written code in a new context. JavaScript provides a means to modify objects to include new methods and/or properties or even to create brand-new objects based on ones that already exist.

These techniques are known, respectively, as *extending* and *inheriting* objects.

Extending Objects

What if you want to extend your objects with new methods and properties after the objects have already been instantiated? You can do so using the keyword `prototype`. The prototype object allows you to quickly add a property or method that is then available for all instances of the object.

▼ TRY IT YOURSELF

Extend an Object Using `prototype`

Let's extend the `Person` object of the previous example with a new method, `sayHello`:

```
Person.prototype.sayHello = function() {
    alert(this.name + " says hello");
}
```

Create a new HTML file in your editor, and enter the code from Listing 8.3.