Understanding Prototypes

As stated many times over in this book, JavaScript is a different kind of OOP language in that it is *prototypical*. This means that the variables you create are derived from a predefined model, called a *prototype*, not a class definition. This prototypical nature is true whether the prototype is built into JavaScript or defined by you:

```
var lang = 'JavaScript';
var e1 = new Employee('Jane', 'Doe', 'Accounting');
```

In that code, the `lang` variable is an instance of the `String` prototype and `e1` is an instance of `Employee`. (One technicality: the value of `lang` is a literal string, but it will be automatically converted to a `String` object when it is used like one.)

Prototypical Inheritance

Every JavaScript object inherits the properties and methods defined in its prototype. If you were to inspect a custom object you created, you'd find methods and properties that you did not create (**Figure 14.7**).

Figure 14.7. The empty `Test` object instance already has some methods defined, inherited from the `Object` prototype.

And, as you've already seen in the `Employee` example, objects can also be assigned their own properties and methods, which would not be found in the original prototype. For example, `Department` has an `addEmployee()` method, but `Object`, `Department`'s prototype, does not.

Some variables will have a prototype that in turn has its own prototype. For example, the `e1` object is based upon the `Employee` prototype, which is based upon `Object`. This is known as the *prototype chain*. When you reference any object property or method, JavaScript looks through the object's prototype chain to find a corresponding attribute. JavaScript will stop looking through the prototype chain when it gets to the root prototype—that from which all other prototypes stem, which is `Object`. If no corresponding property or method is found in the chain, then `undefined` is returned.

To differentiate between properties or methods defined within an object or within its prototype chain, you can call the `hasOwnProperty()` method, providing it with the property in question. This method is defined in `Object`, and is therefore inherited by all other objects. For example:

```
var test = { thing: 1 };
test.hasOwnProperty('thing'); // true
test.valueOf(); // Object: i.e., there is a valueOf() method
test.hasOwnProperty('valueOf'); // false
```

The *Math* object is one of the few that you cannot create an instance of (i.e., it cannot be the prototype for any other objects).

Adding Prototype Methods

In a class-based OOP language, you can change the properties and methods of every instance object by altering the underlying class definition. In JavaScript, which doesn't have classes, you change the prototype's properties and methods by editing the object's `prototype` property. For example, the `Employee` object can be altered after its original definition. Here, a new method is added to it:

```
function Employee(firstName, lastName, department) {
   /* Actual code. */
}
Employee.prototype.getNameBackwards = function() {
   return this.lastName + ', ' + this.firstName:
}
```

Now that the new method has been added to the prototype, you can call it on any instance objects: `e1.getNameBackwards()`. In fact, you can even call this method if the `e1` variable was created *prior to* adding the method to the `Employee` prototype (**Figure 14.8**)!

Figure 14.8. By changing the prototype, any instance of that prototype can make use of the modifications.

The ability to retroactively change a prototype even allows you to change the definition of objects built into JavaScript, such as `String`. This next bit of code adds a `trim()` method to the `String` object, if it doesn't already have one:

```
if (typeof String.prototype.trim == 'undefined') {
   String.prototype.trim = function() {
      return this.replace(/^\s+|\s+$/g,'');
   };
}
```

(The ability to actually trim blank space from the beginning and end of a string requires a regular expression.) After executing that code, your JavaScript will have a `trim()` method for `String` objects, whether that method was native to the browser (as the method was added in ECMAScript 5) or not.

As another example, you could extend the `Date` object so that it has a `getMonthName()` method, which returns the textual version of the month represented by the date.

Although JavaScript allows for you to modify prototypes, this concept is sometimes called *monkey patching*, and should be used cautiously. Understand that modifying a prototype is a global change that impacts every instance of that prototype. Adding unusual methods and properties to built-in JavaScript objects, such as `String`, could create bugs in code that expects the prototype to be untainted. The best use of this concept is to create backwards-functional objects, as in the `String.prototype.trim()` example (i.e., creating a `String` object that can be used reliably regardless of the browser type or version).

Each method added to a prototype is therefore added to every instance of that prototype, whether it is needed or not. If you only need a function for a specific instance, you can create that function separately and call it while providing the object:

```
function doSomething(obj) {
   // Do something with obj.
}
```

Or you could add the function definition to just the single instance:

```
var obj = {};
obj.doSomething = function() {
    // Do something with this.
}
```

Changing a prototype can also be used to prevent closures from being created, although this is a much more advanced topic.