JavaScript Inheritance and Object Programming

Preview Edition

Preview Edition

For the full book:

Amazon (http://www.amazon.com/JavaScript-Inheritance-Object-Programming-Rinehart/dp/1490463046)

The CreateSpace eStore (http://www.createspace.com/4327336, preview edition discount: MAMMBM3G)

The extended eBook from Software Developer's Journal (http://sdjournal.org/javascript-inheritance-and-object-programming-ebook/)

JavaScript Inheritance and Object Programming

Martin Rinehart

Ebook published by *Software Developer's Journal*. Print book printed on demand (POD) by CreateSpace, a subsidiary of Amazon. Designed and written by the author in the beautiful Hudson Valley, New York, USA.

Versions 2e and 2p completed 9 July, 2013.

All text, code and illustrations by Martin Rinehart, now a full-time frontend engineer. He can be contacted through http://www.martinrinehart.com/.

Many thanks to Technical Editors Robert Franzese, founder of Zen40.com, Eric Schmidt of Microsoft Corporation and Assaf Shemesh, an expert in software development and management. Without their efforts this book would have many more errors.

Dedicated to Brendan Eich, who wrote JavaScript and gave us object programming.

JavaScript Inheritance and Object Programming

A Note for the Implementers

JavaScript, or more exactly, the subset of JavaScript Crockford identifies as “The Good Parts,” is a beautiful language. It is small, yet expressive. It's combination of C with functional programming and object programming gives it extraordinary depth. In nearly a half century of programming I have used dozens of languages. Only two of them, JavaScript is one, have been languages I've loved.

Today there are people working to free JavaScript from the browser, to further empower JavaScript (FileWriter and WebGL, to mention personal favorites) and to bring it up to professional speed. My apologies to the latter group. In many ways, object programming is the enemy of compiled speed.

So a word of encouragement and advice to our courageous implementers. First, making JavaScript run at some reasonable fraction of C's speed is a magnificent goal. More power to you! (And yes, that's self-serving. You are giving more power to all of us who write JavaScript. We love it and we thank you for it.)

Second, removing object programming to gain speed cuts out the heart to save the patient. Object programming is not your enemy, it is the essence of the language. Look on it as a challenge, as the Everest of your profession. The view from the top will be spectacular. Object programming at half the speed of C will be breathtaking.

Martin Rinehart, 8 June 2013
Hudson Valley, New York

Website

This book is the sixth “knowit” at the author's website, MartinRinehart.com. From the home page, click “Knowits” and then “JIOP”.

The gateway page holds the notes (both footnotes and bibliography). Hyperlinks are inline in the eBook, online in the printed book (so the printed text of this book is uninterrupted, and so you can click on the references).

From the gateway page, sub menus link you to the “User Guide” to the *JSWindows* system, the engineer's reference material and the source code.

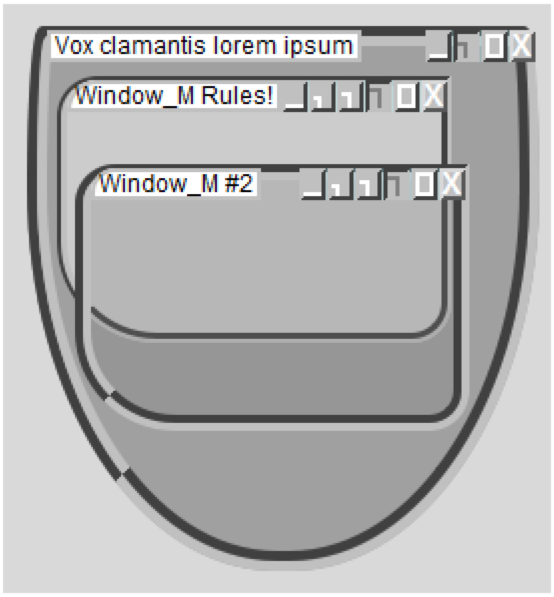
Introduction

Within the JavaScript community there is a great deal of misinformation regarding inheritance. Some comes from object-oriented programmers (C++ and its offspring like Java and C#) looking for, and not finding, their familiar classes. Some comes from JavaScript programmers looking for Self-style prototypal inheritance, which JavaScript's prototypes are not very good at.

In this small volume we begin at the beginning, discussing objects, their place in software and the benefits we expect from them. We continue to first define terms like “class” (three meanings) and “inheritance.” With a common terminology we then look at how class-based and prototypal inheritance can be implemented in JavaScript. To leap ahead, we will propose ways in which JavaScript is best served by minimizing the long inheritance chains associated with object-oriented programming languages.

OOP inheritance is very different from prototypal inheritance. They are widely misunderstood as two ways of achieving the same result. Their similarities are more superficial than profound. We have to understand both to understand JavaScript's hybrid class/prototypal object model.

Throughout the book we look at our *JSWindows* sample system. These are windows, JavaScript style:



These windows can be dragged to new positions, resized, minimized, maximized and so on. They are programmed in JavaScript but with OOP inheritance.

JSWindows does not use prototypal inheritance. It's goal, in addition to being a working window system, was to demonstrate class-based inheritance in JavaScript without additional library code. It was intended to demonstrate the use of long inheritance chains without the use of JavaScript's prototype chain.

Along the way, we also examine JavaScript's object programming (OP) abilities, abilities that are the true distinction between class-based OOP and JavaScript's hybrid object model. *JSWindows* library code (not inheritance-related) makes extensive use of object programming. JavaScript programmers almost all use object programming, though many are not aware of its profound importance.

Last, we show how *JSWindows* uses “capabilities,” a hybrid of Java's interfaces and JavaScript's mixins. We show that capabilities give us the benefits of inheritance without the problems associated with mapping complex real world relationships to OOP-limited models. (Java replaced multiple inheritance with interfaces, and we were big fans.) Object programming enables our capabilities.

Finally, we note that inheritance, specifically OOP inheritance, is a valuable tool for the advanced JavaScript programmer. (Prototypal inheritance may also be valuable, for those so inclined. If you understand that these are two different models, you can use both together.) Once you understand the importance of object programming, however, inheritance assumes a far less prominent part in your systems' architectures.

We make assertions about JavaScript's abilities along the way. The eBook version has two chapters that may interest those who accept nothing without proof. The first discusses JavaScript's relationship to traditional OOP concepts, such as encapsulation and polymorphism. The second elaborates on the details of the JavaScript constructor mechanism.

1 Objects

Objects were a software experiment that worked. They began in research labs in the late 1970s. They became a new, mainstream programming paradigm in the 1980s. By the '90s, languages that were being created (Java, JavaScript, Python, Ruby) were all object-based. Languages that predated objects were being retrofitted. Today even 50-year old languages (Basic, Cobol, Fortran) have adopted objects.

In this book we will be examining two forms of inheritance and alternatives to inheritance. We need to understand the benefits of objects to see how these alternatives provide (or not) the benefits of programming with objects.

We will also be drawing examples from our *JSWindows* demonstration system. It is introduced at the end of this chapter.

Reasons for Objects

There are a lot of reasons to prefer programming with objects. We'll discuss three here.

Objects Do Methods

First, the syntax has things doing things just as happens outside of the world of software. The things objects do are small software programs, functions that the object can perform, commonly called methods. If you program a dog (object) to speak (method) you express it directly:

```
dog.speak(); // says "Woof, woof"
```

Many objects can implement the same action (method) with variations appropriate for each, as Listing 1-1 shows.

Listing 1-1

```
parrot.speak(); // "Polly want a cracker."
kitten.speak(); // "mew, mew"
```

If your parrots and kittens use different methods for speaking, object systems will choose the appropriate method for you. For those who are academically inclined, the eBook version's Chapter 8 explains *subtype polymorphism*, the principle underlying the selection of object-appropriate methods.

Event-Driven Programming

Second, few programs now do anything except by user direction. Most programs paint an interface (menus, icons, buttons) and wait for a user command. This is called "event-driven" programming. When the user clicks the "save" icon, the user's document is written to disk. Internally, the user action could trigger a very small program:

```
user_data.save();
```

Again, it uses the `noun.verb()` syntax. This brings us to the main reason for the success of objects.

Taming Exponential Complexity

As programs grow in size, complexity increases exponentially. We have all used systems that never seemed to be robust. Fixing this broke that. If the complexity is ever successfully tamed, it is after immense expense. Objects help shrink the size and limit the complexity. That reduces the effort (expense).

Let's think about a simple example, a small non-object program might require a thousand lines of code. The same job using objects might require ten methods, each only a hundred lines of code. The ten small methods will be far cheaper to write and vastly cheaper to debug.

As we look at *JSWindows*' code, you will see many features that take surprisingly little code and live comfortably outside of other parts of the system.

Class-Based v. Prototypal

Objects come in many guises. We will be using class-based object-oriented programming (OOP, based on classes) and JavaScript's hybrid class/prototypal object model.

Object systems go back to Simula and SmallTalk, in the late '70s. These found the mainstream when Bjarne Stroustrup, at Bell Labs, wrote a compiler that output programs in plain C from an object-enhanced form he called C++. (During the '80s, C was the dominant language for professional programming on non-IBM mainframes and almost all minicomputers and microcomputers.) C++ was first available in 1983. By the end of the decade it had become ubiquitous.

To add objects to C, Stroustrup made some sacrifices. His enhanced C used class modules (more on these shortly) that were not objects, but that defined objects. Despite protests from purists, Stroustrup's limited features were well chosen and programmers were delighted with its improvements over original C. Other languages, such as Java (1995) adopted the C++ object model. Programming in C++ and its progeny is called "object-oriented programming" or "class-based" (aka "classical") OOP.

Class-based OOP relies on class software that defines the objects created. The programmer decides what features each group of objects will need and programs an appropriate class software module that will create objects with those features.

By contrast, the “prototypal” object paradigm does not use class modules. The programmer creates a prototype object and other objects are then cloned from the prototype. This was an experimental object model from Self, a language that evolved from SmallTalk.

JavaScript's original author, Brendan Eich, adopted the prototypal model, partially, in a scripting language for Netscape's then market-leading web browser, Netscape Navigator. JavaScript, born in 1995, combined class-based and prototypal forms. As JavaScript is, to date, the only language available for writing programs that run in all browsers, it has exploded in popularity as the web has exploded.

Class-based OOP and JavaScript perform similar functions and provide similar benefits.

Programmers have begun to discover that JavaScript supports almost complete object programming (defined in Chapter 3) and this is a major advance over the limitations of the class-based model, quite separate from and far more important than JavaScript's prototypal features.

So what are objects?

Objects Up Close

An object is a collection of properties (often a set, but “set” has a mathematical meaning we do not want here). Properties are named values. Values may be, depending on the language, simple values (boolean, integer, character), composite values built from other values (arrays, objects) or, in languages capable of functional programming, functions or other blocks of code. (JavaScript also implements functional programming borrowed from the Scheme language, a dialect of Lisp.)

Some authors use the word “property” to specifically mean what we call “data properties”—properties that are not methods. We find this misleading in a language where functions are first-class objects.

In JavaScript, property names must be strings. In most class-based OOP languages they must be strings that are limited by the restrictions imposed on variable names.

Objects also are permitted direct access to a collection of functions (commonly called "instance methods") that are part of the class software (in class-based languages) or the prototype (in JavaScript). These functions are separate from, but available to, the objects.

All dogs, in the example above, could access the `speak()` method:

```
dog.speak(); // says "Woof, woof"
```

Here, `speak()` is a property of dog objects. In the text, trailing parentheses indicate that the property is executable code, called a method. This is one of the two main categories of object properties.

Data Properties

Objects may have data properties. These are often said to describe an object's "state." A dog might have properties such as `name`, `breed` and `date-of-birth`. Each dog (called an "object instance" or, in class-based OOP, an "instance" of the dog class) will have space allocated to store each of these properties' values. The key point is that each instance has its own set of data properties.

Methods (Code Properties)

Unlike data, the methods are stored in the class software, in class-based OOP, or in the prototype, in JavaScript. (As methods require storage space, it would be extremely inefficient to store a separate copy of each method with each instance of the class.)

Methods operate on the data properties of each instance. If each dog had a "message" property, a small breed could say, "Yap, yap" while a large breed said "Woof, woof." If an application's dogs could speak a bit of English, the programmer might combine fixed values with the dog's `name` property to achieve a result like the one shown in Listing 1-2.

Listing 1-2

```
collie.speak(); // "My name is Lassie."

beagle.speak(); // "My name is Snoopy."
```

Next we put objects to use. Our *JSWindows* system is one example.

Introducing *JSWindows*

The result you see in Figure 1-1 takes exactly four lines of JavaScript code. One creates the page title and three create the three windows. (Is there a law that says windows must be rectangles?)

The gateway to the online portion of this book is http://www.martinrinehart.com/frontend-engineering/knowits/op/knowits-op.html. You can use the system your self (and enjoy the full color) by clicking "Using *JSWindows*" from the gateway page. Let's take a look.

Windows with Border Radii

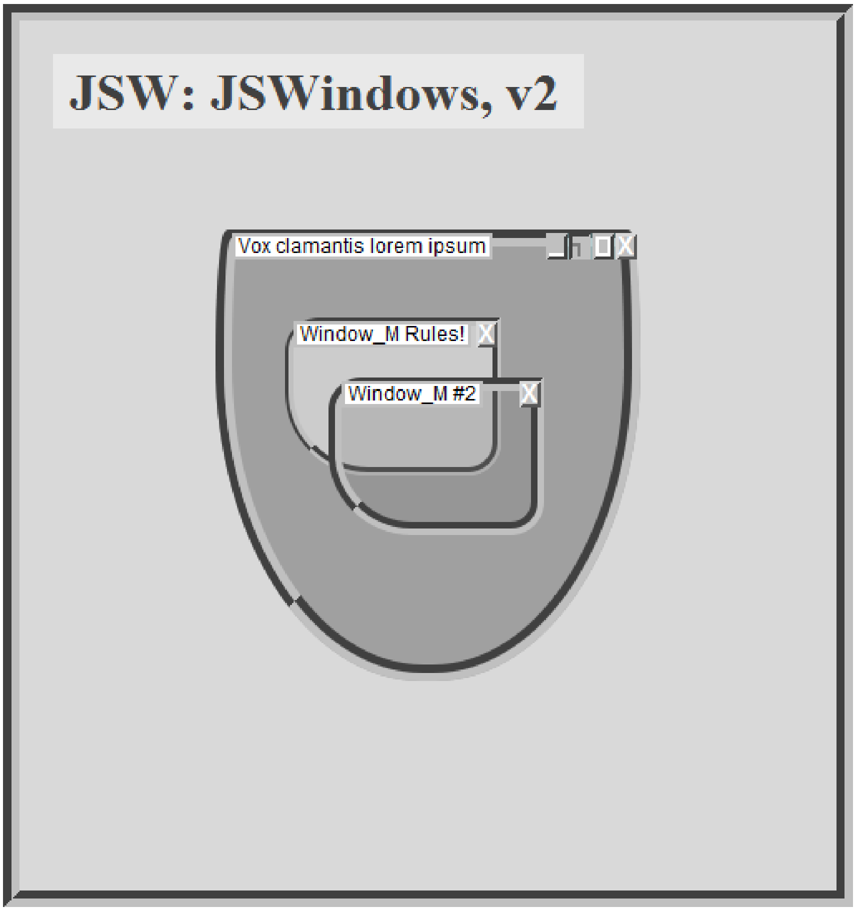


Figure 1-1

The shield in the center of Figure 1-1 is, as you can see from the motto, the frontend engineers' family crest. As you can see from the buttons on its upper right, it is also a window. You can close, maximize, restore and minimize with these buttons (right to left).

The title is also a drag handle.

JSWindows is written in JavaScript for applications on all devices that run HTML4 or 5 and CSS2 or 3. The shapes here are done with CSS3 border radii.

JSWindows Contain JSWindows

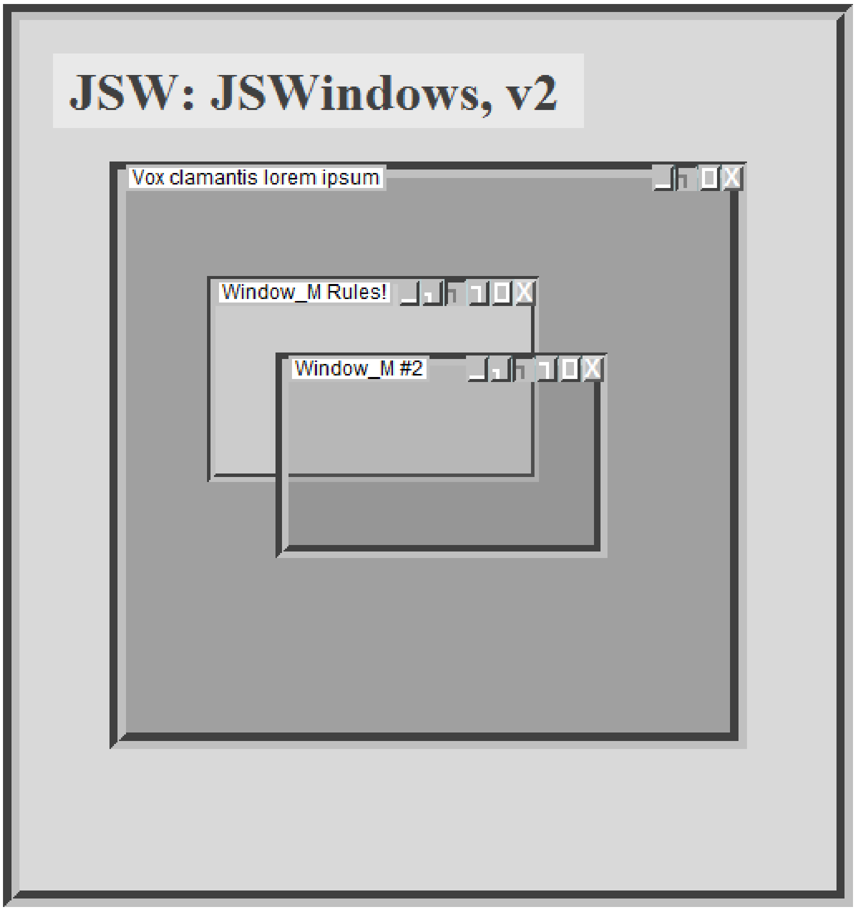


Figure 1-2

In Figure 1-2 we've changed the size of the shield, eliminated the border radii specifications from each window and changed the two inner windows from `Window_M` (window, movable) to `Window_M_BS` (window, movable with buttons for sizing).

Note that there are, by default, three intermediate sizes (between min and max), not just the single “restored” size of some other systems.

We are also using the CSS3 RGBA (transparency) capability for the #2 window.

Click Small, Click Large

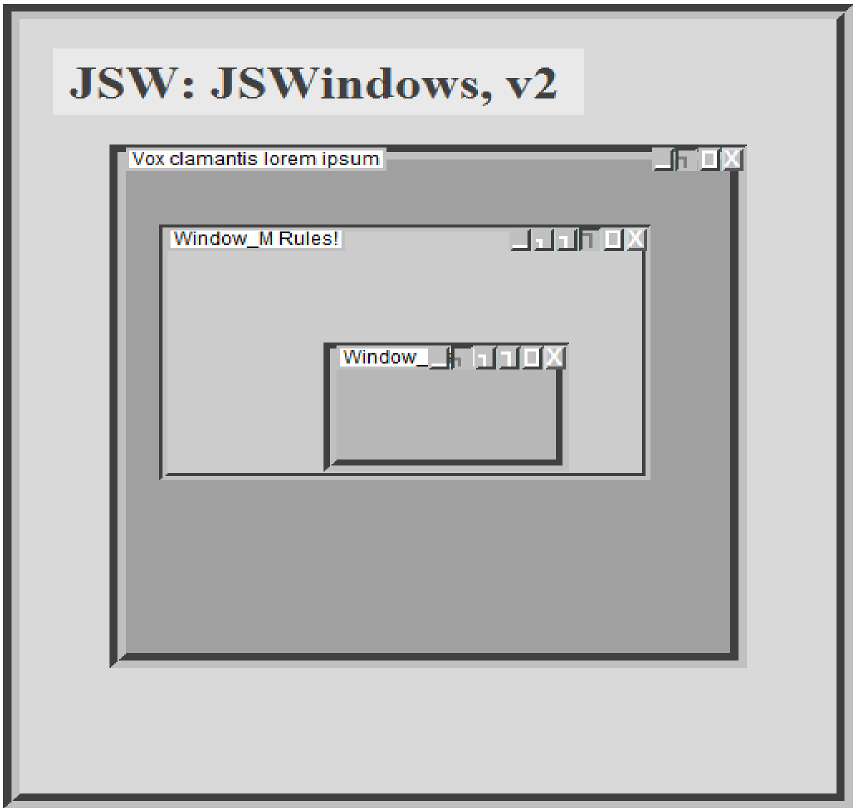


Figure 1-3

Figure 1-3 is exactly two button clicks away from Figure 1-2. Look at the sizing buttons. We clicked the “large” button for one small window, the “small” button for the other.

Next we go on to minimizing. Note that we are minimizing windows within their containers. The shield (the one with the motto) is contained within SCREEN. The two smaller windows are contained within the shield.

Minimized Windows in a Window

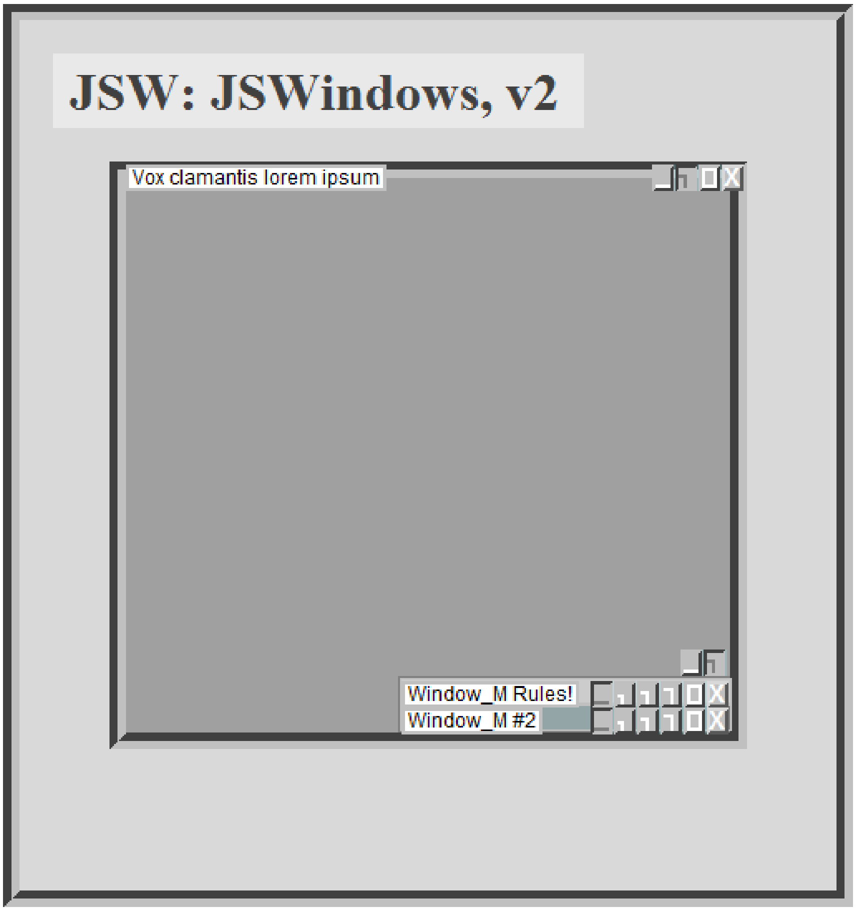


Figure 1-4

Notice that the “minbox” (Figure 1-4, holding the two minimized windows) also has a restore (now disabled) and a minimize sizing button. If you click its minimize button, the minbox is reduced to a one-pixel-tall box, topped by its own sizing buttons, taking almost no screen real estate.

But let's return to the windows with the CSS3 border radii. They're more fun.

The Original Windows, Almost

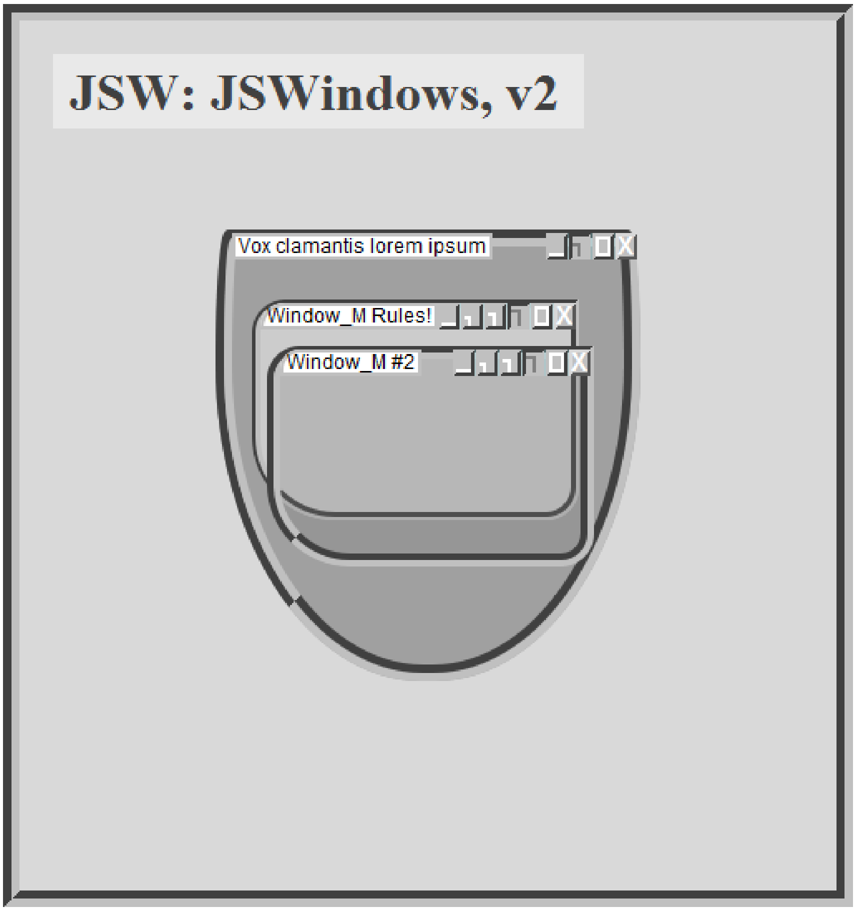


Figure 1-5

In Figure 1-5 the original windows are shown, except that the two smaller ones are still sizable. Here their “large” buttons have been clicked.

Maximizing in the Shield

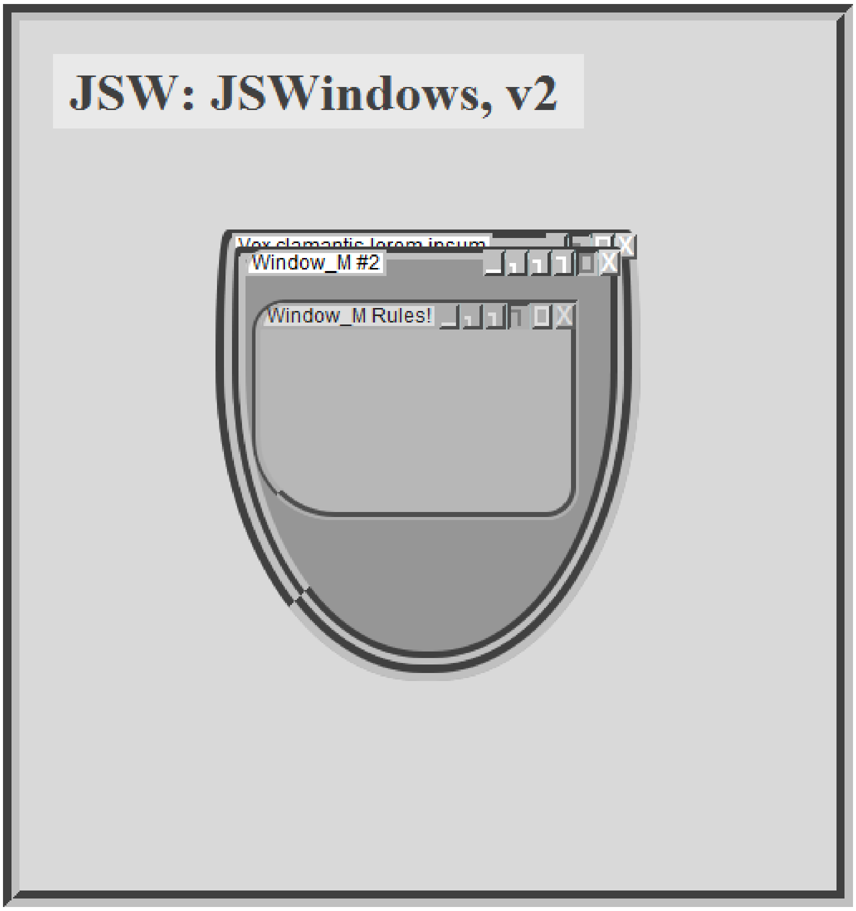


Figure 1-6

In Figure 1-6, we clicked the “maximize” button of the translucent window. (If you compare with Figure 1-5, you can see that the “Window_M Rules!” window is masked.)

Isn't this the way you would have coded “maximize”? (It's easier than you might guess. It works inside circles and ovals, as well as shields, too.)

JSWindows is more fun, and in full color, when you try it online http://www.martinrinehart.com/frontend-engineering/knowits/op/knowits-op.html.

The *JSWindows* system is programmed with class-based inheritance, using long inheritance chains typical of class-based languages. (It does not use OOP-emulating library code, nor the JavaScript prototype chains typical of library OOP implementations in JavaScript.) We return to this system frequently as we discuss the implementation of inheritance in JavaScript.

In Chapter 2 we back up, beginning at the beginning.

2 JavaScript Objects

We begin, in this chapter, by looking at JavaScript objects, how they are created and how their “prototypes” work.

Built-Ins

Programmers new to JavaScript quickly learn that there are common needs that are met by built-in JavaScript “objects.” For example, common mathematical operations are performed by the `Math` object's methods.

Math

You want to round a number to the nearest integer. Listing 2-1 shows an example.

Listing 2-1

```
var width =
    content_width + padding + border_width;

div.style.width = Math.round(width) + 'px';
```

The `Math` “object” is really just a collection of library functions, all related to mathematical operations. You should not manipulate it as an object.

Date

The `Date` constructor returns real objects, time stamps that include both date and time to the nearest millisecond. The `Date` also provides the methods you want to manipulate dates (to set a time for an appointment next week, for example). The bit of code in Listing 2-2 might be included when you were comparing the running times of alternative functions.

Listing 2-2

```
var start = new Date();
var result = sample_long_function();
elapsed = start - new Date();

alert('Time was: ' + elapsed);
```

The programmer is teased by these built-in objects. Hopefully, you decide that you want to go further and create your own objects.

Ex Nihilo Objects

In JavaScript, you can create objects from a constructor (from the equivalent of class software as in C++) or from a prototype (as in Self). JavaScript combines both. It also lets you create an object *ex nihilo* (from nothing).

There are two direct ways to create a JavaScript object from nothing. You can use the `Object` constructor or an object literal.

The Object Constructor

In JavaScript there is an object constructor, `Object`.

```
my_object = new Object();
```

As the name suggests, this creates an object. The initial capital letter tells you that the `Object()` method was intended for use after the `new` operator as a constructor. When you create an object this way you can assign properties, as Listing 2-3 shows:

Listing 2-3

```
my_object.size = 'large';
my_object.color = 'blue';
```

Veterans of static, class-based languages (such as C++, Java and many others) find this capability novel. They are accustomed to names and types of object properties being fixed at compile time. Dynamic properties (as in Python) remove this restriction. (Static properties give optimal performance. Just-in-time compilation—a technique pioneered in Self—minimizes any speed difference, however.)

The prototype (discussed below) of objects created from nothing is `Object.prototype`.

Object Literals

Listing 2-4 shows another way the above *ex nihilo* object could be created.

Listing 2-4

```
my_object = {
      size: 'large',
      color: 'blue'
};
```

As in C, use of whitespace is for readability. The following line does the same job.

```
my_object={size:'large',color:'blue'};
```

JavaScript object literals also create instances of the `Object` family (as if you used the `Object` constructor) and their prototype is, again, `Object.prototype`.

The object literal notation (documented by Crockford as JSON, JavaScript Object Notation) is still the preferred method for creating objects from nothing. The `Object` constructor has been assigned powerful new capabilities by the most recent versions of the JavaScript standard (ECMAScript). You will want to consider these new capabilities when they become universally available.

Objects created from the `Object` constructor and objects created from object literals all share `Object.prototype` as their prototype. This gives them all, for example, a `toString()` method that reports "[object Object]," telling you that the type is "object", its constructor is `Object`. (This is almost never helpful. Creating custom objects lets you supply useful `toString()` methods. If you `alert()` or `console.log()` an object you will be looking at its `toString()` version. You usually know that it is an object.)

More *ex nihilo* Objects

Functions and arrays are also objects in JavaScript. Whenever you create a function (named or anonymous, with the `function` keyword or the `Function` constructor), you create an object. When you create an array (via the `Array` constructor or the array literal notation) you are also creating objects.

This is another way of creating an *ex nihilo* object:

```
arr = ['dog', 'cat', 'mouse'];
```

From now on, however, we will reserve the adjective *ex nihilo* for objects that are neither functions nor arrays.

Custom Objects

To create your own objects, ones that will have their own prototypes (the next section), you write your own constructor functions, just as you do in class-based languages.

In JavaScript you can begin in much the same way that OOP programmers begin, by writing a constructor function.

Constructors

By convention, the name of a constructor begins with a capital letter. The constructor function assigns properties, commonly data properties, and adds initial values. Listing 2-5 shows a simple example.

Listing 2-5

```
function Dog(name, breed) {
    var new_dog = this; // the 'new' object

    new_dog.name = name;
    new_dog.breed = breed;
}
```

If you chose the parameter names carefully, chances are those same names are also good names for the object's properties. (Study the example to convince yourself that this does not lead to ambiguities.) The pattern in Listing 2-5, where the value of the parameter `name` becomes the value of the `name` property, is very common.

Assigning Initial Property Values

Using your `Dog` constructor, your `Dog` objects can start their lives with two properties, `name` and `breed`, and each `Dog` will have values for those properties. Listing 2-6 shows this constructor being used to create `Dog` objects.

Listing 2-6

```
var snoopy = new Dog('Snoopy', 'Beagle');
var lassie = new Dog('Lassie', 'Collie');
```

Familiar `object.property` notation can be used to access these properties' values.

```
alert(snoopy.breed); // 'Beagle'
```

Non-Trivial Property Values

In addition to assigning arguments passed into the constructor, the constructor can create any properties that you wish. This could include arbitrarily complex computations. In class-based OOP programming, the object's property names and types are specified by the class software at compile time. In JavaScript, the constructor function serves this requirement. (In Self, the properties, called "slots," could be added whenever the programmer chose. As in Self, in JavaScript, and other languages such as Python, the properties are dynamic and may be changed, added or deleted, during execution. In traditional class-based OOP languages, only the property values may be changed.)

In JavaScript, the values you assign to properties can include any JavaScript values, including other objects. You could assign functions as property values, but this would seldom be useful. If your constructor assigns functions, it will be assigning the same code to every object it creates. You normally want the function code to be part of the object's prototype (as it is part of the class software, in class-based OOP languages).

Methods as Prototype Values

Methods (functions) can be assigned as object properties, but that would be wasteful. The method code can be assigned to the prototype where it can be accessed by all objects that are members of the family (generated from the same constructor).

A data property could also be assigned to the prototype, but that, too, would be wasteful. Assume you have several types of animals and they have diets such as "carnivore" and "herbivore." Putting this data value into the prototype would make it available to all animals of each type, as Listing 2-7 shows.

Listing 2-7

```
Dog.prototype.diet = 'carnivore';
alert(lassie.diet); // She's a 'carnivore'.
```

The lookup, however, is not needed. In `lassie.diet`, JavaScript looks to the `lassie` object for a property named "diet." It does not find it. Then it looks to the prototype, where it finds the value it seeks. If the property value is the same for all family members, the property can be made a property of the constructor (shown in Listing 2-8), saving the wasted lookup.

Listing 2-8

```
Dog.diet = 'carnivore'; // not Dog.prototype
alert(Dog.diet); // same for Lassie, Snoopy...
```

The JavaScript usage of the constructor is almost identical to the OOP programmer's use of the class software. Instance data is assigned to each object. Methods, and data that applies to the whole family, are part of the class software. (`Dog.diet` is `'carnivore'`; `lassie.diet` is `undefined`.) Instance methods are part of the prototype for the convenience of using `object.method()` calls, for example, `lassie.speak()`.

Now we take a longer look at object prototypes, risking repetition in favor of being absolutely clear.

Object Prototypes

When JavaScript sees an object property name, such as "color" in `object.color` it looks to the object for a property of that name. If the name does not exist it looks to the object's prototype for a property of that name. (The prototype is another object.)

Assume that you created singers, as Listing 2-9 shows.

Listing 2-9

```
function Singer() {}; // no properties
var patty = new Singer(),
    maxene = new Singer(),
    laverne = new Singer();

patty.sing(); // error, undefined method
```

We have no `sing()` method, so these girls don't know how to sing. In Listing 2-10 we teach Patty, individually.

Listing 2-10

```
patty.sing = function () {
    alert('...boogie woogie bugle boy...');
}
patty.sing(); //...boogie woogie bugle boy...
```

Patty can sing, but her sisters cannot. We should have put this method in the prototype, as in Listing 2-11.

Listing 2-11

```
Singer.prototype.sing = function () {
    alert('...blows eight-to-the-bar...');
};
maxene.sing(); // she sings!
laverne.sing(); // she sings, too!
```

How did we know that `Singer.prototype` was the prototype for our singing sisters? The prototype for any object you create from a constructor is the property named "prototype" of the constructor function from which the object was instantiated. If `Singer` is the constructor, then `Singer.prototype` is the prototype for `Singer` family objects. (The eBook's Chapter 9 fills in all the details of the underlying mechanism by which `Singer.prototype` becomes the prototype for all `Singer` family instances.)

In the class-based model, instance methods are written in the class software. In Java you would have created a `sing()` method in the `Singer.java` module. In JavaScript you assign the `sing()` method to the `Singer` function's `prototype` property. The result is the same. The objects created from the constructor can `sing()`.

(The Andrews Sisters—Patty, Maxene and LaVerne—had their biggest hit, of many, with "Boogie Woogie Bugle Boy" in 1941. http://www.dump.com/andrewssisters/, 2:21)

The Prototype Chain

Now we repeat the paragraph that started the previous section:

> When JavaScript sees an object property name, such as “color” in `object.color` it looks to the object for a property of that name. If the name does not exist it looks to the object's prototype for a property of that name. (The prototype is another object.)

What happens if JavaScript does not find the name in the prototype? “The prototype is another object.” Simply read the paragraph again. JavaScript will look in the prototype's prototype. In JavaScript, all objects you create are either created from the `Object` constructor, and therefore `Object.prototype` is their prototype, or their constructor's prototype is an object created from `Object`, or their constructor's constructor is an object created from `Object`, and so on.

Searching prototypes stops at `Object.prototype`. This is called the prototype chain. Ultimately, `Object.prototype` is the prototype of every object you create, directly or indirectly.

Bear in mind the following two important facts as you consider the prototype chain. First, there are large families of objects (not ones that you create) that may not have prototypes. JavaScript does not know how to do input or output. It depends on a “host environment” for all I/O. Most commonly, JavaScript runs in a browser that provides the host environment through objects. These are called “host objects” and unless the browser's authors were meticulous (most weren't) the host objects do not have prototypes. Host objects seldom have prototype chains.

Second, you can see how the prototype chain could be used to implement inheritance. If you could connect your `Extend` family objects so that properties not found in `Extend.prototype` would be sought in `Base.prototype`, you would have inheritance. What does not follow is that this would be a good way to achieve inheritance. There are other ways that are preferable.

Many authors suggest you use the prototype chain to achieve “prototypal inheritance.” In a language such as Self, creating one object from a prototype object is referred to as “inheritance.” In JavaScript, creating objects *ex nihilo* involves no inheritance. Creating objects via a call to a constructor is analogous to creating objects in class-based OOP. (JavaScript's *ex nihilo* objects inherit from `Object.prototype`, so they aren't really *ex nihilo*.)

Is the term “inheritance” appropriate for creating objects from a constructor? In OOP, creating base objects from the `Base` constructor is not called inheritance. That term refers specifically to the creation of objects from a family such as `Extend` that “inherits” from a family such as `Base`. In fact, you can code JavaScript as if the prototype were part of an OOP family. *JSWindows* is coded this way. Object methods can be found in their prototypes but properties not in the prototype simply do not exist. (Assume that there is no prototype chain. A property not found in the prototype does not exist.)

The only extensive use of the prototype chain is made by programmers who wish to provide fundamental additions to a JavaScript built-in class. For example, you might wish to add a capability to all `Array` objects. You could add additional method(s) to `Array.prototype`.

Most experienced JavaScript programmers are opposed to this idea. What happens if one library adds extensions to `Array.prototype` and another also adds extensions to `Array.prototype`? We have lots of new capabilities and we're all happy until one library's author inadvertently (and inevitably) picks the same name chosen by another. Then we are in trouble. Whichever code is loaded last gets ownership of that duplicated name. The library that depended on the first-loaded (and therefore overwritten) version of that name becomes buggy. The history of JavaScript is replete with once reliable libraries that suddenly became buggy because of these naming conflicts.

It follows that if the prototype chain is not used for modifications to JavaScript basics and it is not used for inheritance, the prototype chain could be removed from JavaScript. It could and *JSWindows*, for one, would not miss it. (*JSWindows*, as we will see, uses extensive inheritance, none of it using the prototype chain.)

In you want to experiment with Self-like object prototypes, in *JavaScript: The Good Parts* Crockford advocates them and provides a function to create true prototypal inheritance. (It only takes a half dozen lines of JavaScript.)

Now that we have introduced JavaScript objects, we can look at JavaScript's object programming (OP) capability which we will use constantly. Before we go on, let's put this much to use.

Coding a Class

Ready to start writing code? In this chapter and the next four we have guided programming tutorials. The road map is this:

- Create a useful `Box` class (here in chapter 2).
- Add a `styles` configuration object using object programming (3).
- Create a `Borders` class and combine it with `Box` objects using composition (4).
- Create a `Button` class that inherits from `Box` (5).
- Add a `Maskable` capability and an `implements()` method that adds this capability to a `Box` (6).

A) Create Your Template

If you already have a template, or your editor/IDE can create one, test it against these requirements. Upgrade as needed. If need be, create a new template.

A minimal HTML template will include at least:

- A doctype
- File path and name comments
- `<html>`, `<head>` and `<body>` tags
- `<title>` in the head
- `<script>` at the end of the body
- `“use strict”;` in the script

Compare your template to the one at http://www.martinrinehart.com/frontend-engineering/knowits/op/op-tutorial/op-tut-2/op-tut-2a.html. Make sure that yours is better than ours. (But don't overload it with bits you might not need.)

B) Add a Class Skeleton

Turn your template into an HTML file. We'll be working continuously on this one file for all the tutorials in this book. Place it in some convenient folder.

Now we start to write JavaScript.

Every class will need a constructor, an `init()` method (property of the constructor, not in the prototype) and a t`oString()` method (in the prototype). Add these and compare against our sample at http://www.martinrinehart.com/frontend-engineering/knowits/op/op-tutorial/op-tut-2/op-tut-2b.html.

C) Add Parameters

Now we need to think about the actual work for our class. That means deciding on the parameters for our constructor. Think carefully here, but don't get stuck. (Think like a backpacker. “When in doubt, leave it out.”)

Our sample includes a

- `parent` (`document.body` or another Box)
- `id` (unique, for the DOM element and for our own use)
- `pos_size` (left,top, width,height array)
- `color` (background)

The `init()` method should copy these parameters into properties of the `new_box`. Writing these with the `toString()` method will let you test your work this far. (A simple test is in our sample.)

Our sample is at http://www.martinrinehart.com/frontend-engineering/knowits/op/op-tutorial/op-tut-2/op-tut-2c.html. You'll be able to find our samples (“Next” on the right, bottom or the letter on the menu) without any more links, we hope.

D) Finish `Box.init()`, Add a Mainline

To finish `init()`, `document.createElement()` a `div` as a `Box.delem` (DOM element) property. Give it an ID (with your `id` property). Add to the `delem`'s `style` property: a background color and styles for your position and size values. Don't forget to add a `position` style (`absolute` or `relative`). And don't forget to `appendChild()` it to the `parent` element.

Now convert that `alert()` message to an actual mainline, with sensible values in the `pos_size` property. Run. Fix typos. Repeat until you have a `Box` on the screen.

It's good to see your work on the screen. This is always the point where we start to smile. Test out your `pos_size` array with different positions and sizes. Convince yourself that your `color` works. (Create more than one `Box` as you test.)

E) Add Styles

Next chapter we'll do a little object programming, using a `styles` configuration object. For now, we just add styles to the mainline. We use `textAlign`, `fontSize` and `padding`. Go ahead and add any that you like. (We'll do a `Borders` class in Chapter 4, so you might skip border styles for now.)

F) Finish the Mainline

And finally, finish your work with some content in the `Box`. Extra credit if you don't slavishly follow our example.

You've now coded a constructor and an `init()`, so you can inherit from it easily. And you've added an instance method (`toString()`) to the prototype. Not quite ready to graduate from the School of JavaScript OOP, but well on your way.

Now, on to object programming. It's easier to do than to say.

3 Object Programming

We are now ready to consider object programming (OP), the fundamental difference between JavaScript and OOP languages. *JSWindows* makes continuous use of object programming. (An early attempt to write *JSWindows* without OP was abandoned. It may have been possible, but the extra effort was not in our time budget.)

In class-based object-oriented programming the objects are defined before they are created. Once an object exists, you can execute its methods or assign values to its data properties. But you cannot add or delete properties or change methods, for examples. Object programming gives you the ability to do these, and more.

OP Defined

The OOP model combines object instances, sets of name/value pairs, with class software. The class software provides a routine to create instances, to store instance methods and it provides other services. This is a subset of object programming, which includes these abilities.

- An object programming (OP) system allows the creation, modification and disposal of objects during program execution.
- Objects are collections of properties. Properties are name/value pairs.
- "Modification" of objects means the ability to create, modify and delete properties. Modifying properties includes modifying names and/or values.
- An OP system also provides for services provided by OOP classes, such as storing data and code at the class software level (to avoid duplicating either in each instance object).

Note that by this definition, the JavaScript OP model is incomplete as it does not allow direct modification of property names. (It takes a three-line utility function to modify a property name. A direct approach to modifying the name would be preferable, but you can live without it.)

Programming with Properties

JavaScript leads programmers into object programming, almost without their being aware of it. It's two ways of identifying object properties play a major role.

Dot Notation

In the familiar `object.property` notation, a period separates the object reference from the property name. The latter is a constant in source code.

```
var x = object.prop_name;
```

In the above, “object” is the name of an object reference. “prop_name” is the name of a property of that object (directly, or via its prototype or prototype chain).

This is the common OOP notation, as well. Few OOP languages have the equivalent of JavaScript's subscript notation.

Subscript Notation

Subscript notation allows the use of variable property names. First, with a constant, the above example is repeated here.

```
var x = object['prop_name'];
```

Listing 3-1 shows an example of an expression used to select a property.

Listing 3-1

```
var e = func_returning_string(args);
var x = object[e];
```

Listing 3-2 shows both notations being used to create a new property within an object. Constant and variable property names are shown.

Listing 3-2

```
Object.new_prop = value;
Object['new_prop'] = value;

var e = func_returning_string(args);
object[e] = value;
```

The pair of features above provides surprising power and grace when programming.

Object Programming Examples

Object Sum

JSWindows uses “styles” objects to hold lists of CSS styles. These are the JavaScript versions of CSS declaration lists. Each name/value pair in the object corresponds to a CSS property name and value. This is a styles object:

```
{borderWidth: '8px', borderColor: '#a0a0ff'}
```

As styles become known, they are added to an object's `styles` object. We often want to add the properties of a new object to an existing object in the process. Listing 3-3 shows this situation in pseudocode.

Listing 3-3

```
var bstyles = borders.get_styles();
all_styles += bstyles; // pseudocode
```

The `+=` operator does not apply to objects, but we can write a `sum` function to do the same job. We want the styles in the incoming `bstyles` object to add new properties to the `all_styles` object. If properties of the same name already exist in `all_styles`, we want to replace their values. As Listing 3-4 shows, this is more trouble to explain than to code.

Listing 3-4

```
sum = function (old_object, new_object) {
    var ret = {};

    for (var prop in old_object) {
            ret[prop] = old_object[prop]; }
    for (prop in new_object) {
            ret[prop] = new_object[prop]; }

      return ret;
} // end: sum()
```

This function creates a new, empty object. Then the first `for`/`in` loop copies the old object's property names and values into the new object. The second `for`/`in` loop copies the new property names and values into the new object. In the process it will create new properties, if required, or override existing properties where there is a name conflict.

If your project wants a `sum()` function where the values in the old object are preserved (not overridden by values in the new object) it is simple to modify the above. As you loop through the properties of `new_object`, just check:

```
if (old_object[prop] === undefined) ...
```

Do not overwrite the existing property if it is already defined.

OP for Inheriting Prototypes

We also use the `sum()` function to copy prototypes, underneath the `extends()` function. The latter provides a meaningful name and saves typing, two reasons worthy of our support. Listing 3-5 shows `extends()`.

Listing 3-5

```
extends = function (extend, base) {
    extend.prototype = sum(
        extend.prototype,
        base.prototype);
}
```

OP in the *JSWindows* Library

For those who have not totally understood the idea (and for those who are asking, “Is it really that simple?”) we provide additional examples of object programming from the *JSWindows* library.

DOM related

The library functions are divided into a “DOM related” group, for dealing with the browser's host environment, and a “Utility” group, for everything else. Almost 80% of both make use of some form of object programming. Listing 3-6 shows the function that deletes a single DOM element. This would be simpler if one browser's bugs did not make it necessary to assign null to the deleted reference.

Listing 3-6

```
delete_delem = function (delem) {

    while (delem.firstChild) {
        delete_delem(delem.firstChild);
    }
    delem.parentNode.removeChild(delem);
    delem = null; // Some MSIE needs this.

} // end: delete_delem()
```

The `delete_delem()` function calls itself recursively to remove children from the `delem` (DOM element). When the children are gone, it removes the selected element from its parent. As JavaScript's main use is for DOM manipulation, it shows how OP is an integral part of its job.

For a second example, Listing 3-7 shows the library function that attaches an event listener to a DOM element. As with so much DOM-related work, one of its jobs is to smooth over differences between browsers. As with so many of these differences, Internet Explorer, especially older versions, is the problem.

Listing 3-7

```
/** Add an event listener. */
listen_for = function (
        wobj, event_name, func) {
    var delem = wobj.delem;

    if (delem.addEventListener !==undefined) {
        delem.addEventListener(
                event_name, func, false); }
    else if (delem.attachEvent !==undefined) {
        delem.attachEvent(
                'on' + event_name, func); }
                // IE before 9
    else { delem['on' + event_name] = func; }
        // old school!

} // end: listen_for()
```

Again we are working with object properties, in this case, event listening methods. Here we use both dot notation and subscript notation as we fall back to progressively older ways of adding the event listener.

An interesting feature of this library function is that the `Wobj` reference (Window object, the root of our family hierarchy) is passed as an explicit parameter, Fortran style. We prefer to call these functions in the object style:

```
wobj.listen_for(event_name, func);
```

Listing 3-8 shows the method in Wobj.prototype that lets us use our preferred style.

Listing 3-8

```
Wobj.prototype.listen_for = function (
        event_name, listen_func) {

    listen_for(this, event_name,
            listen_func);
}
```

When you call `object.method()`, the `object` reference on the left is converted to the `this` parameter within the method. We simply put `this` back into the Fortran-style parameter list, explicitly. That gives us object-style method calling for our Fortran-style library functions.

Our third, and final, example from the DOM-related library functions removes an event listener that had been added by the old-fashioned method:

```
element.onclick = click_func;
```

Listing 3-9 shows the listener removing function.

Listing 3-9

```
stop_listening_on = function (wobj, type) {
    var delem = wobj.delem;

    delem['on' + type] = undefined;
}
```

JavaScript's subscript notation makes this job simple.

Utility

We have been big fans of object programming for a long time so it came as no surprise that most of our DOM-related utilities manipulated object's properties. After all, the DOM is an object tree. What we were surprised to find was how many of our utility (non-DOM) functions also used OP. We'll start with a simple example.

Whenever we write a constructor, we also write a `toString()` method. It seems you always want to have a readable version of an object as you are developing. But what about your *ex nihilo* objects? The default `toString()` (from `Object.prototype`) reports that you have "[object Object]" (an object created by the `Object` constructor). This is almost never helpful. Listing 3-10 shows a simple utility that creates a readable version of an object that is frequently helpful.

Listing 3-10

```
o2s = function (obj) {
    var ret = [];
    for (var pname in obj) {
        var prop = obj[pname];
        if (typeof prop !== 'function') {
            ret.push(pname + ': ' + prop);
        }
    }
    return 'object{' + ret.join(',') + '}';
}
```

This loops through the properties by their names (in `pname`) and, if they are not functions, pushes them onto an array of property name/value pairs, as strings. That array is used as the center of the returned string.

Listing 3-11 shows an example of object programming applied to arrays. (Arrays are objects, in JavaScript.) It removes an element from an array, shortening the array by one (ensuring that there are no undefined elements created).

Listing 3-11

```
remove_element = function(arr, element) {

    var index = find_index(arr, element);
    if (index === -1) { return; }

    ret = [];
    for (var i in arr) {
        if (i !== index) { ret.push( arr[i]; )
    return ret;

} // end: remove_element()
```

One of the most common mistakes in any object programming is to assign a second reference to an object when a second object is needed.

```
var not_really_second = first;
```

When `not_really_second` is changed, the same change appears in `first` as they are both references to a single object. Making a shallow copy of `first` is normally correct.

```
var really_second = shallow_copy(first);
```

Listing 3-12 shows our `shallow_copy()` function.

Listing 3-12

```
shallow_copy = function (obj) {
      var ret;

      if (obj instanceof Array) { ret = []; }
      else if (obj instanceof Function) {
               ret = obj; }
      else { ret = {}; }

      for (var name in obj) {
               ret[name] = obj[name]; }

      return ret;
} // end: shallow_copy()
```

The code that copies arrays and non-array objects is identical except that the return value is initialized differently. The calling code does not care. Note that using a `for`/`in` loop to copy the array correctly handles sparse arrays and arrays that have been modified via `splice()` calls.

For those new to shallow and deep copying, a "shallow" copy means that values that are references to other objects are copied. A "deep" copy would duplicate the objects and arrays referenced. During a shallow copy, statements such as the following are executed:

```
copy.prop_name = original.prop_name;
```

This creates a second property with the same name and value as the first, but it is totally separate. After the assignment, there is no connection between the two. It's as if we had `lassie.breed = 'Collie'` and `snoopy.breed = 'Beagle'`.

If the code subsequently assigns to the copy, it does not impact the original.

```
copy.prop_name = some_other_object;
```

The above statement replaces the value of `copy.prop_name` with another object reference. It has no effect on the value of `original.prop_name`.

We are now equipped to consider inheritance in JavaScript, both class-based and prototypal. Before we start on code, we will take a close look at what is really meant by “inheritance” when programming with objects. But first, let's do a little object programming. You will be an ace in a few minutes.

Coding Object Programs

We'll be working with objects. To test our work, we'll want to take a look at our objects. If we use an object where it will be coerced to a string (such as in an `alert()` or `console.log()`) JavaScript may tell us that it's an “[object Object].” (An object made from the `Object` costructor, as every *ex nihilo* object is.) This is seldom helpful. We want to look at the properties of the object. So we begin with a simple, hard-working utility to tell us more.

A) Object to String

We'll use an `o2s()` function (object to string) constantly as we develop. It's job is to take an object, any object, and list its properties. We want the name and value of each property. (You'll want a smarter `o2s()` if you need to show function objects, as their values—the function source code—tend to be very long.)

The `for (name in object)` loop iterates through any object, returning each property's name. Try using it to write an `o2s()` function. Test it with object literals:

```
alert( o2s({breed:'Beagle', name:'Snoopy'}) );
```

Don't fuss overly much with the beauty of the output. It's only for use during development. And it's only for use with objects that don't have a nice `toString()` method, which can do a much better job.

Our version is shown at http://www.martinrinehart.com/frontend-engineering/knowits/op/op-tutorial/op-tut-3/op-tut-3a.html.

B) A `get_ps_styles()` Function

Next, we'll want to put together our `Box`'s styles in the `init()` method, adding `styles` objects as we go along. Right now, however, the body of `Box.init()` is rather crowded with the dull work of turning a `pos_size` array into CSS style specifications. We've moved this into an inner function whose job is to take the `pos_size` array and return the CSS we need. (A value of 100 for `pos_size[0]`-—left—becomes “`left: '100px'`”.)

Take a peek at ours, if you like, and then write your own. Test it with an `alert()` or `console.log()` inside the `Box.init()` method.

C) A Library Function to `sum()` Obects

Our new `Box.init()` will do its work by the `sum()` library function. You want to write and test your `sum()` carefully, as this one will definitely be at the heart of your production code. Don't forget any of these:

- `sum( {}, {foo:'bar'} )`
- `sum( {foo:'bar'}, {} )`
- `sum( {foo:'bar'}, {foo:23} )`

You can look in the code (3, C) if you haven't a clue where to start. (And you can look back a few pages in this chapter, for a second opinion.) You'll find that `sum()` is hard-working, but that doesn't mean it's hard to code.

Note that the sum operation is not commutative. `sum(a, b)` is not equal to the `sum(b, a)` if `a` and `b` have like-named properties. In the third example, above, the value of `foo` will be 23 after the sum, as properties of the second object override like-named properties of the first object. This becomes important in `styles` objects as we want styles we explicitly define to override default styles.

D) Putting `sum()` to Use

Begin by replacing the `color` parameter with a `styles` parameter. (Three places: in the parameter list in the constructor, in the `Box.init()` parameters and in the argument list in the constructor's call to `Box.init()`.) Styles will be a configuration object you can use to specify any CSS styles you like. In your mainline, replace the color specification with an object, like `{backgroundColor: '#c0e0ff'}`. Don't forget to replace the line `new_box.color = …` with `new_box.styles = …`.

Replace the part of `Box.init()` that assigns values to your `delem`'s `styles` object. This takes more thinking than code. Ensure that the specific styles from the `styles` configuration object override any other styles. (Do you want to look at our version first? That's fine. But write your own! This is a very fine example of a little bit of OP making short work of a hard task.)

Our new core of the `Box.init()` method is woefully lacking in comments. That's your job. Feel free to copy our code but add enough comments so that a maintenance programmer will be able to make sense of it.

In the end, test by adding to the styles configuration object. Any style explicitly specified in `styles` should override anything else. For example, you should be able to right align text with appropriate styles.

E) Teach `toString()` About Styles

We've laid the groundwork and made this one easy. The `toString()` method should now be able to show the styles you include in your `styles` object. You've already got the `o2s()` method that will make this easy.

F) A Mainline with Styles

As a next-to-last step, replace those explicit assignments from the work you did in Chapter 2 with assignments to a `styles` configuration object. (This has very little real value with just one box. Picture dozens of boxes, each using one of a handful of different styles objects. That will give this approach lots of value.)

At this point you should have a system that relies on styles configuration objects and you can happily forget the fact that someplace, deep inside, these are assigned to a `delem.style` object.

G) An `alert()` to Test

Finally, if you `alert()` or `console.log()` your boxes, you'll see `toString()`s showing readable styles objects.

Now we can move on to inheritance.

End,
Preview Edition

For the full book:

Amazon (http://www.amazon.com/JavaScript-Inheritance-Object-Programming-Rinehart/dp/1490463046)

The CreateSpace eStore (http://www.createspace.com/4327336, preview edition discount: MAMMBM3G)

The extended eBook from Software Developer's Journal (http://sdjournal.org/javascript-inheritance-and-object-programming-ebook/)

Notes

The eBook's Chapters 8 and 9 disect OOP object principles (polymorphism, encapsulation and so on) and the more obscure details of the new operator/JavaScript constructor pair, respectively. The eBook is available from http://sdjournal.org/javascript-inheritance-and-object-programming-ebook/.

References, as in a bibliography, should be online, not in print. (At least until you can place your mouse on a printed page and click, online will be preferred.)

The gateway to the online portion of this book is http://www.martinrinehart.com/frontend-engineering/knowits/op/knowits-op.html.

The gateway page provides links to references.

Defined Terms

Often the most controversial portion of an analysis is the definitions. We have attempted to use terms that have (more or less) common definitions and avoid terms that are ambiguous (e.g., polymorphism). Wikipedia is particularly valuable in having a community editing process that often achieves consensus on definitions. We also check other sources, however. Our defined terms and their references include:

Object-oriented programming `Oop-W`, `Oop-1-5`

Prototype-based programming `Prtp-W`, `Prtp-1-5`

Classes `Clss-W`, `Clss-1-6`

Instances (objects) `Inst-W`, `Inst-1-4`

Methods `Mthd-W`, `Mthd-1-4`

Inheritance `Inhr-W`, `Inhr-1-6`

Composition `Cmp-W`, `Cmp-1-4`

(A second dash, as in `Oop-1-5`, denotes the full range: `Oop-1`, `Oop-2`, …, `Oop-5`.)

Support for Selected Statements

“Today even 50-year old languages (Basic, Cobol, Fortran) have adopted objects.” `Oop-3`, `Inhr-4`, `Clss-6`, `Mthd-4`

“Other languages, such as Java (1995) adopted the C++ object model.” `Clss-1-6`, particularly `Clss-2`

“By contrast, the “prototypal” object paradigm does not use classes. The programmer creates a prototype object and other objects are then copied from the prototype.” `Prtp1-4`

“An object is a collection of properties (often a set, but “set” has a mathematical meaning we do not want here). Properties are named values.” `Inst-1-4`

“Objects also are permitted direct access to a collection of functions (commonly called “instance methods”) that are part of the class (in class-based languages) or the prototype (in JavaScript).” `Inst-3`, `Mthd-1-4`, `Prtp-1`, `Prtp-2`, `Prtp-4`

> “In OOP, a class is the software that creates and supports a set of objects, including the constructor and methods that instances of the class can perform. A class may also have methods and data of its own (“class statics”, in Java).” `Clss-1-6`

'In class-based OOP, when `Bo` is empty we say that `E` “extends” `B`, or `E` “inherits from” `B`.' `Inhr-2-6`.

[In prototypal inheritance] “Objects inherit directly from each other. The base object is called the “prototype” of the inheriting object. “ `Prtp-1-4`

“If an object includes another type of object as a property, it is using composition.” `Inhr-1-2`, but see `Inhr-3` and `Inhr-4` for qualifiers.

“[JavaScript] also lets you create an object *ex nihilo* (from nothing).” `Prtp-2`

“We are not endorsing inheritance-based architecture;” `Inhr-1`

'We extend interfaces to “capabilities” which borrow from and extend both Java's interfaces and JavaScript's “mixins.”' `Trts-W`,