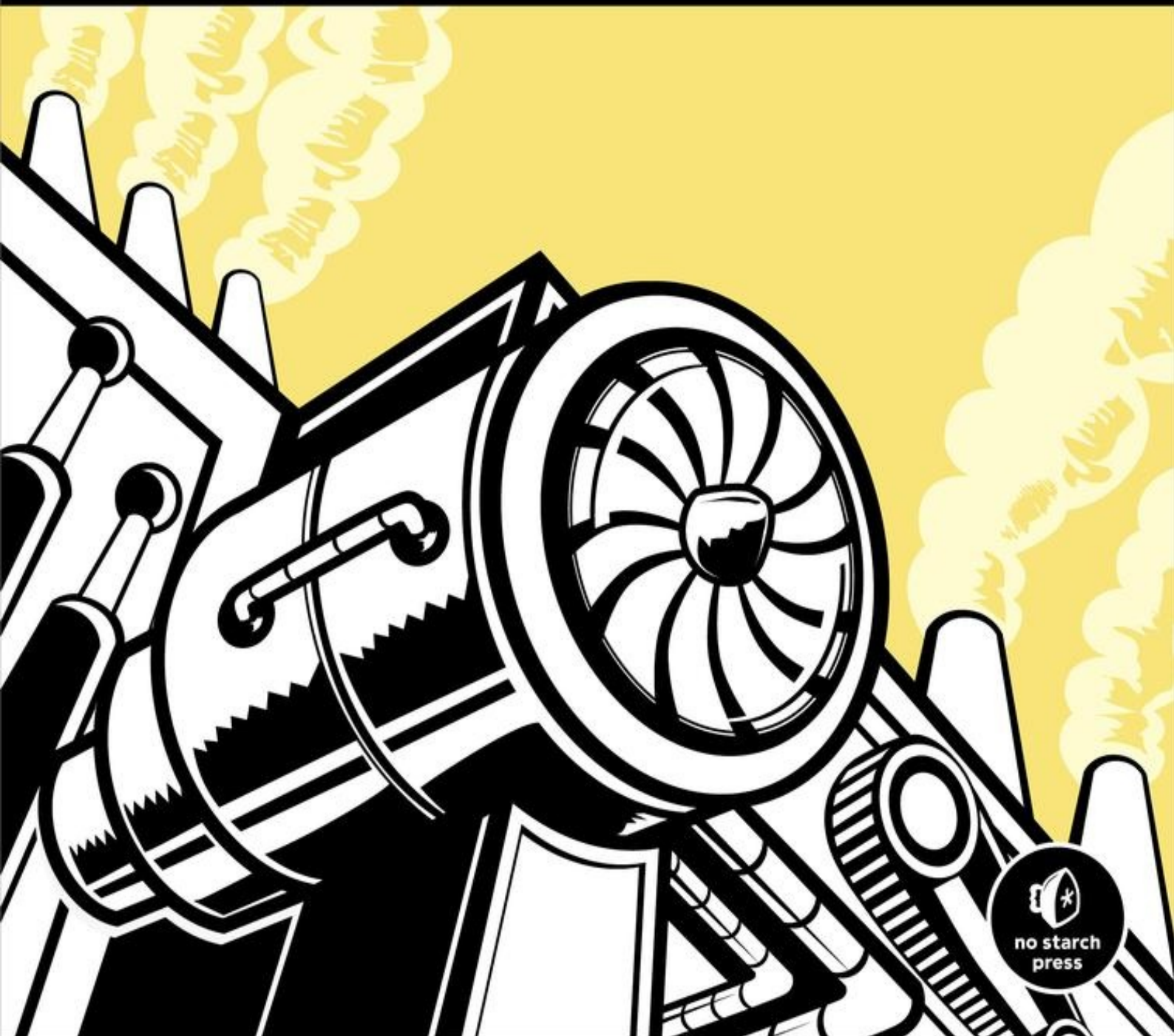
THE PRINCIPLES OF OBJECT-ORIENTED JAVASCRIPT

NICHOLAS C. ZAKAS



no starch press

The Principles of Object-Oriented Javascript

Nicholas C. Zakas

Published by No Starch Press

About the Author

Nicholas C. Zakas is a software engineer at Box and is known for writing on and speaking about the latest in JavaScript best practices. He honed his experience during his five years at Yahoo!, where he was principal frontend engineer for the Yahoo! home page. He is the author of several books, including *Maintainable JavaScript* (O'Reilly Media, 2012) and *Professional JavaScript for Web Developers* (Wrox, 2012).

About the Technical Reviewer

Originally from the UK, Angus Croll is now part of Twitter's web framework team in San Francisco and is the co-author and principal maintainer of Twitter's open source Flight framework. He's obsessed with JavaScript and literature in equal measure and is a passionate advocate for the greater involvement of artists and creative thinkers in software development. Angus is a frequent speaker at conferences worldwide and is currently working on two books for No Starch Press. He can be reached on Twitter at @angustweets.

Foreword

The name Nicholas Zakas is synonymous with JavaScript development itself. I could ramble on for pages with his professional accolades, but I am not going to do that. Nicholas is well-known as a highly skilled JavaScript developer and author, and he needs no introduction. However, I would like to offer some personal thoughts before praising the contents of this book.

My relationship with Nicholas comes from years of studying his books, reading his blog posts, watching him speak, and monitoring his Twitter updates as a JavaScript pupil. We first met in person when I asked him to speak at a jQuery conference several years ago. He treated the jQuery community to a high-quality talk, and since then, we have spoken publicly and privately over the Internet. In that time, I have come to admire him as more than just a leader and developer in the JavaScript community. His words are always gracious and thoughtful, his demeanor always kind.

His intent as a developer, speaker, and author is always to help, to educate, and to improve. When he speaks, you should listen, not just because he is a JavaScript expert, but because his character rises above his professional status.

This book's title and introduction make Nicholas's intentions clear: he has written it to help class-minded (that is, C++ or Java) programmers transition to a language without classes. In the book, he explains how encapsulation, aggregation, inheritance, and polymorphism can be accomplished when writing JavaScript. This is the ideal text to bring a knowledgeable programmer into the fold of object-oriented JavaScript development. If you are reading this book as a developer from another language, you are about to be treated to a concise and skillfully worded JavaScript book.

However, this book also stands to serve programmers coming from within the JavaScript fold. Many JavaScript developers have only an ECMAScript 3 (ES3) understanding of objects, and they are in need of a proper introduction to ECMAScript 5 (ES5) object features. This book can serve as that introduction, bridging a knowledge gap between ES3 objects and ES5 objects.

Now, you might be thinking, "Big deal. Several books have included chapters or notes on the additions to JavaScript found in ES5." Well, that is true. However, I believe this to be the only book written to date that focuses on the nature of objects by giving ES5 objects first-class citizenship in the entire narrative. This book brings a cohesive introduction to not only ES5 objects, but also the bits of ES3 that you need to grok while learning many of the new additions found in ES5.

As an author myself, I strongly believe this is the one book, given its focus on object-oriented principles and ES5 object updates, that needed to be written as we await ES6

updates to scripting environments.

Cody Lindley (*www.codylindley.com*)

Author of *JavaScript Enlightenment*, *DOM Enlightenment*, and *jQuery Enlightenment*

Boise, Idaho

December 16, 2013

Acknowledgments

I’d like to thank Kate Matsudaira for convincing me that self-publishing an ebook was the best way to get this information out. Without her advice, I’d probably still be trying to figure out what I should do with the information contained in this book.

Thanks to Rob Friesel for once again providing excellent feedback on an early copy of this book, and Cody Lindley for his suggestions. Additional thanks to Angus Croll for his technical review of the finished version— his nitpicking made this book much better.

Thanks as well to Bill Pollock, whom I met at a conference and who started the ball rolling on publishing this book.

Introduction

Most developers associate object-oriented programming with languages that are typically taught in school, like C++ and Java, which base object-oriented programming around classes. Before you can do anything in these languages, you need to create a class, even if you're just writing a simple command-line program.

Common design patterns in the industry reinforce class-based concepts as well. But JavaScript doesn't use classes, and this is part of the reason people get confused when they try learning it after C++ or Java.

Object-oriented languages have several characteristics:

Encapsulation. Data can be grouped together with functionality that operates on that data. This, quite simply, is the definition of an object.

Aggregation. One object can reference another object.

Inheritance. A newly created object has the same characteristics as another object without explicitly duplicating its functionality.

Polymorphism. One interface may be implemented by multiple objects.

JavaScript has all these characteristics, though because the language has no concept of classes, some aren't implemented in quite the way you might expect. At first glance, a JavaScript program might even look like a procedural program you would write in C. If you can write a function and pass it some variables, you have a working script that seemingly has no objects. A closer look at the language, however, reveals the existence of objects through the use of dot notation.

Many object-oriented languages use dot notation to access properties and methods on objects, and JavaScript is syntactically the same. But in JavaScript, you never need to write a class definition, import a package, or include a header file. You just start coding with the data types that you want, and you can group those together in any number of ways. You could certainly write JavaScript in a procedural way, but its true power emerges when you take advantage of its object-oriented nature. That's what this book is about.

Make no mistake: A lot of the concepts you may have learned in more traditional object-oriented programming languages don't necessarily apply to JavaScript. While that often confuses beginners, as you read, you'll quickly find that JavaScript's weakly typed nature allows you to write less code to accomplish the same tasks as other languages. You can just start coding without planning the classes that you need ahead of time. Need an object with specific fields? Just create an ad hoc object wherever you want. Did you forget to add a method to that object? No problem—just add it later.

Inside these pages, you'll learn the unique way that JavaScript approaches object-oriented programming. Leave behind the notions of classes and class-based inheritance and learn about prototype-based inheritance and constructor functions that behave similarly. You'll learn how to create objects, define your own types, use inheritance, and otherwise manipulate objects to get the most out of them. In short, you'll learn everything you need to know to understand and write JavaScript professionally. Enjoy!

Who This Book Is For

This book is intended as a guide for those who already understand object-oriented programming but want to know exactly how the concept works in JavaScript. Familiarity with Java, C#, or object-oriented programming in other languages is a strong indicator that this book is for you. In particular, this book is aimed at three groups of readers:

- Developers who are familiar with object-oriented programming concepts and want to apply them to JavaScript
- Web application and Node.js developers trying to structure their code more effectively
- Novice JavaScript developers trying to gain a deeper understanding of the language

This book is not for beginners who have never written JavaScript. You will need a good understanding of how to write and execute JavaScript code to follow along.

Overview

Chapter 1 introduces the two different value types in JavaScript: primitive and reference. You'll learn what distinguishes them from each other and how understanding their differences is important to an overall understanding of JavaScript.

Chapter 2 explains the ins and outs of functions in JavaScript. First-class functions are what makes JavaScript such an interesting language.

Chapter 3 details the makeup of objects in JavaScript. JavaScript objects behave differently than objects in other languages, so a deep understanding of how objects work is vital to mastering the language.

Chapter 4 expands on the previous discussion of functions by looking more specifically at constructors. All constructors are functions, but they are used a little bit differently. This chapter explores the differences while also talking about creating your own custom types.

Chapter 5 explains how inheritance is accomplished in JavaScript. Though there are no classes in JavaScript, that doesn't mean inheritance isn't possible. In this chapter, you'll learn about prototypal inheritance and how it differs from class-based inheritance.

Chapter 6 walks through common object patterns. There are many different ways to build and compose objects in JavaScript, and this chapter introduces you to the most popular patterns for doing so.

Help and Support

If you have questions, comments, or other feedback about this book, please visit the mailing list at *http://groups.google.com/group/zakasbooks*.

Chapter 1. Primitive and Reference Types

Most developers learn object-oriented programming by working with class-based languages such as Java or C#. When these developers start learning JavaScript, they get disoriented because JavaScript has no formal support for classes. Instead of defining classes from the beginning, with JavaScript you can just write code and create data structures as you need them. Because it lacks classes, JavaScript also lacks class groupings such as packages. Whereas in languages like Java, package and class names define both the types of objects you use and the layout of files and folders in your project, programming in JavaScript is like starting with a blank slate: You can organize things any way you want. Some developers choose to mimic structures from other languages, while others take advantage of JavaScript's flexibility to come up with something completely new. To the uninitiated, this freedom of choice can be overwhelming, but once you get used to it, you'll find JavaScript to be an incredibly flexible language that can adapt to your preferences quite easily.

To ease the transition from traditional object-oriented languages, JavaScript makes objects the central part of the language. Almost all data in JavaScript is either an object or accessed through objects. In fact, even functions (which languages traditionally make you jump through hoops to get references to) are represented as objects in JavaScript, which makes them *first-class functions*.

Working with and understanding objects is key to understanding JavaScript as a whole. You can create objects at any time and add or remove properties from them whenever you want. In addition, JavaScript objects are extremely flexible and have capabilities that create unique and interesting patterns that are simply not possible in other languages.

This chapter focuses on how to identify and work with the two primary JavaScript data types: primitive types and reference types. Though both are accessed through objects, they behave in different ways that are important to understand.

What Are Types?

Although JavaScript has no concept of classes, it still uses two kinds of *types*: primitive and reference. *Primitive types* are stored as simple data types. *Reference types* are stored as objects, which are really just references to locations in memory.

The tricky thing is that JavaScript lets you treat primitive types like reference types in

order to make the language more consistent for the developer.

While other programming languages distinguish between primitive and reference types by storing primitives on the stack and references in the heap, JavaScript does away with this concept completely: It tracks variables for a particular scope with a *variable object*. Primitive values are stored directly on the variable object, while reference values are placed as a pointer in the variable object, which serves as a reference to a location in memory where the object is stored. However, as you'll see later in this chapter, primitive values and reference values behave quite differently although they may initially seem the same.

Of course, there are other differences between primitive and reference types.

Primitive Types

Primitive types represent simple pieces of data that are stored as is, such as `true` and 25. There are five primitive types in JavaScript:

Boolean	`true` or `false`
Number	Any integer or floating-point numeric value
String	A character or sequence of characters delimited by either single or double quotes (JavaScript has no separate character type)
Null	A primitive type that has only one value, `null`
Undefined	A primitive type that has only one value, `undefined` (`undefined` is the value assigned to a variable that is not initialized)

The first three types (Boolean, number, and string) behave in similar ways, while the last two (null and undefined) work a bit differently, as will be discussed throughout this chapter. All primitive types have literal representations of their values. *Literals* represent values that aren't stored in a variable, such as a hardcoded name or price. Here are some examples of each type using its literal form:

```
// strings
var name = "Nicholas";
var selection = "a";

// numbers
var count = 25;
var cost = 1.51;

// boolean
var found = true;

// null
var object = null;
```

```
// undefined
var flag = undefined;
var ref;    // assigned undefined automatically
```

In JavaScript, as in many other languages, a variable holding a primitive directly contains the primitive value (rather than a pointer to an object). When you assign a primitive value to a variable, the value is copied into that variable. This means that if you set one variable equal to another, each variable gets its own copy of the data. For example:

```
var color1 = "red";
var color2 = color1;
```

Here, `color1` is assigned the value of `"red"`. The variable `color2` is then assigned the value `color1`, which stores `"red"` in `color2`. Even though `color1` and `color2` contain the same value, they are completely separate from each other, and you can change the value in `color1` without affecting `color2` and vice versa. That's because there are two different storage locations, one for each variable. Figure 1-1 illustrates the variable object for this snippet of code.

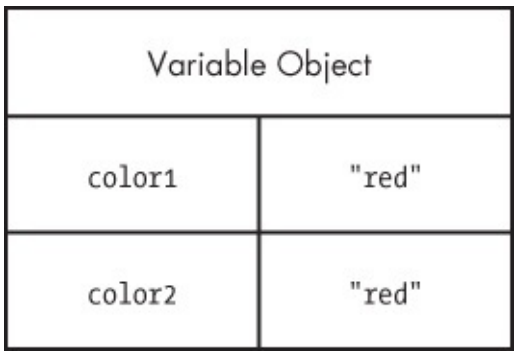


Figure 1-1. Variable object

Because each variable containing a primitive value uses its own storage space, changes to one variable are not reflected on the other. For example:

```
var color1 = "red";
var color2 = color1;

console.log(color1);    // "red"
console.log(color2);    // "red"

color1 = "blue";

console.log(color1);    // "blue"
```

```
console.log(color2);    // "red"
```

In this code, `color1` is changed to `"blue"` and `color2` retains its original value of `"red"`.

Identifying Primitive Types

The best way to identify primitive types is with the `typeof` operator, which works on any variable and returns a string indicating the type of data. The `typeof` operator works well with strings, numbers, Booleans, and `undefined`. The following shows the output when using `typeof` on different primitive values:

```
console.log(typeof "Nicholas");     // "string"
console.log(typeof 10);             // "number"
console.log(typeof 5.1);            // "number"
console.log(typeof true);           // "boolean"
console.log(typeof undefined);      // "undefined"
```

As you might expect, `typeof` returns `"string"` when the value is a string; `"number"` when the value is a number (regardless of integer or floating-point values); `"boolean"` when the value is a Boolean; and `"undefined"` when the value is undefined.

The tricky part involves `null`.

You wouldn't be the first developer to be confused by the result of this line of code:

```
console.log(typeof null);       // "object"
```

When you run `typeof null`, the result is `"object"`. But why an object when the type is `null`? (In fact, this has been acknowledged as an error by TC39, the committee that designs and maintains JavaScript. You could reason that `null` is an empty object pointer, making `"object"` a logical return value, but that's still confusing.)

The best way to determine if a value is `null` is to compare it against `null` directly, like this:

```
console.log(value === null);        // true or false
```

COMPARING WITHOUT COERCION

Notice that this code uses the triple equals operator (===) instead of the double equals operator. The reason is that triple equals does the comparison without coercing the variable to another type. To understand why this is important, consider the following:

```
console.log("5" == 5);              // true
```

```
console.log("5" === 5);             // false

console.log(undefined == null);     // true
console.log(undefined === null);    // false
```

When you use the double equals, the string `"5"` and the number `5` are considered equal because the double equals converts the string into a number before it makes the comparison. The triple equals operator doesn't consider these values equal because they are two different types. Likewise, when you compare `undefined` and `null`, the double equals says that they are equivalent, while the triple equals says they are not. When you're trying to identify `null`, use triple equals so that you can correctly identify the type.

Primitive Methods

Despite the fact that they're primitive types, strings, numbers, and Booleans actually have methods. (The `null` and `undefined` types have no methods.) Strings, in particular, have numerous methods to help you work with them. For example:

```
var name = "Nicholas";
var lowercaseName = name.toLowerCase();     // convert to lowercase
var firstLetter = name.charAt(0);           // get first character
var middleOfName = name.substring(2, 5);    // get characters 2-4

var count = 10;
var fixedCount = count.toFixed(2);          // convert to "10.00"
var hexCount = count.toString(16);          // convert to "a"

var flag = true;
var stringFlag = flag.toString();           // convert to "true"
```

NOTE

Despite the fact that they have methods, primitive values themselves are not objects. JavaScript makes them look like objects to provide a consistent experience in the language, as you'll see later in this chapter.

Reference Types

Reference types represent objects in JavaScript and are the closest things to classes that you will find in the language. Reference values are *instances* of reference types and are synonymous with objects (the rest of this chapter refers to reference values simply as *objects*). An object is an unordered list of *properties* consisting of a name (always a string) and a value. When the value of a property is a function, it is called a *method*. Functions themselves are actually reference values in JavaScript, so there's little difference between a property that contains an array and one that contains a function except that a function can be executed.

Of course, you must create objects before you can begin working with them.

Creating Objects

It sometimes helps to think of JavaScript objects as nothing more than hash tables, as shown in Figure 1-2.

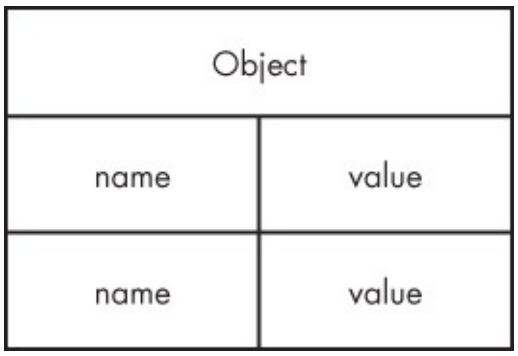


Figure 1-2. Structure of an object

There are a couple of ways to create, or *instantiate*, objects. The first is to use the `new` operator with a *constructor*. (A constructor is simply a function that uses `new` to create an object—any function can be a constructor.) By convention, constructors in JavaScript begin with a capital letter to distinguish them from nonconstructor functions. For example, this code instantiates a generic object and stores a reference to it in `object`:

```
var object = new Object();
```

Reference types do not store the object directly into the variable to which it is assigned, so the `object` variable in this example doesn't actually contain the object instance. Instead, it holds a pointer (or reference) to the location in memory where the object exists. This is the primary difference between objects and primitive values, as the primitive is stored directly in the variable.

When you assign an object to a variable, you're actually assigning a pointer. That means if you assign one variable to another, each variable gets a copy of the pointer, and both still reference the same object in memory. For example:

```
var object1 = new Object();
var object2 = object1;
```

This code first creates an object (with `new`) and stores a reference in `object1`. Next, `object2` is assigned the value of `object1`. There is still only the one instance of the

object that was created on the first line, but both variables now point to that object, as illustrated in Figure 1-3.

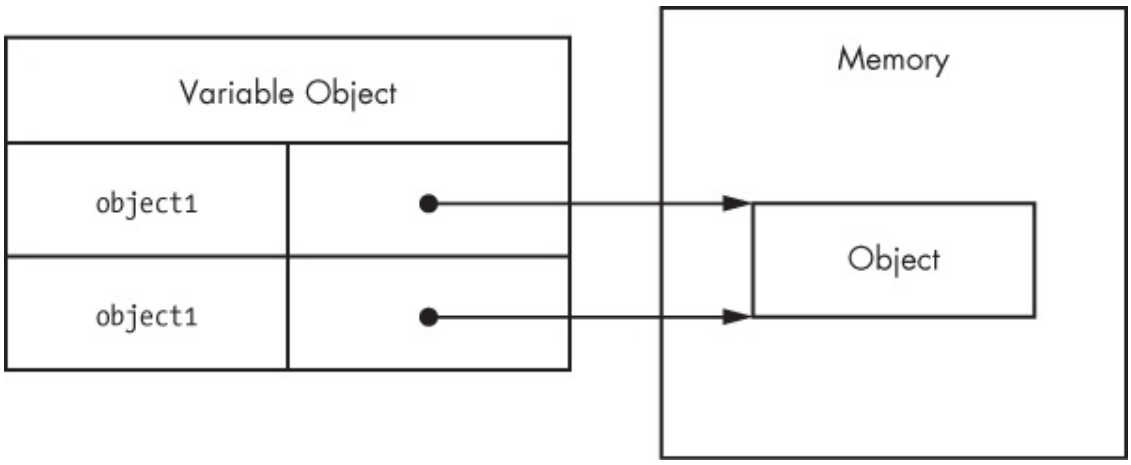


Figure 1-3. Two variables pointing to one object

Dereferencing Objects

JavaScript is a garbage-collected language, so you don't really need to worry about memory allocations when you use reference types. However, it's best to *dereference* objects that you no longer need so that the garbage collector can free up that memory. The best way to do this is to set the object variable to `null`.

```
var object1 = new Object();

// do something

object1 = null;     // dereference
```

Here, `object1` is created and used before finally being set to `null`. When there are no more references to an object in memory, the garbage collector can use that memory for something else. (Dereferencing objects is especially important in very large applications that use millions of objects.)

Adding or Removing Properties

Another interesting aspect of objects in JavaScript is that you can add and remove properties at any time. For example:

```
var object1 = new Object();
var object2 = object1;

object1.myCustomProperty = "Awesome!";
```

```
console.log(object2.myCustomProperty);      // "Awesome!"
```

Here, `myCustomProperty` is added to `object1` with a value of `"Awesome!"`. That property is also accessible on `object2` because both `object1` and `object2` point to the same object.

> **NOTE**
>
> *This example demonstrates one particularly unique aspect of JavaScript: You can modify objects whenever you want, even if you didn't define them in the first place. And there are ways to prevent such modifications, as you'll learn later in this book.*

In addition to generic object reference types, JavaScript has several other built-in types that are at your disposal.

Instantiating Built-in Types

You've seen how to create and interact with generic objects created with `new Object()`. The `Object` type is just one of a handful of built-in reference types that JavaScript provides. The other built-in types are more specialized in their intended usage and can be instantiated at any time.

The built-in types are:

`Array`	An ordered list of numerically indexed values
`Date`	A date and time
`Error`	A runtime error (there are also several more specific error subtypes)
`Function`	A function
`Object`	A generic object
`RegExp`	A regular expression

You can instantiate each built-in reference type using `new`, as shown here:

```
var items = new Array();
var now = new Date();
var error = new Error("Something bad happened.");
var func = new Function("console.log('Hi');");
var object = new Object();
var re = new RegExp("\\d+");
```

Literal Forms

Several built-in reference types have literal forms. A *literal* is syntax that allows you to define a reference value without explicitly creating an object, using the `new` operator and the object's constructor. (Earlier in this chapter, you saw examples of primitive literals including string literals, numeric literals, Boolean literals, the `null` literal, and the `undefined` literal.)

Object and Array Literals

To create an object with *object literal* syntax, you can define the properties of a new object inside braces. Properties are made up of an identifier or string, a colon, and a value, with multiple properties separated by commas. For example:

```
var book = {
    name: "The Principles of Object-Oriented JavaScript",
    year: 2014
};
```

You can also use string literals as property names, which is useful when you want a property name to have spaces or other special characters:

```
var book = {
    "name": "The Principles of Object-Oriented JavaScript",
    "year": 2014
};
```

This example is equivalent to the previous one despite the syntactic differences. Both examples are also logically equivalent to the following:

```
var book = new Object();
book.name = "The Principles of Object-Oriented JavaScript";
book.year = 2014;
```

The outcome of each of the previous three examples is the same: an object with two properties. The choice of pattern is up to you because the functionality is ultimately the same.

> **NOTE**
>
> *Using an object literal doesn't actually call new Object(). Instead, the JavaScript engine follows the same steps it does when using new Object() without actually calling the constructor. This is true for all reference literals.*

You can define an *array literal* in a similar way by enclosing any number of comma-separated values inside square brackets. For example:

```
var colors = [ "red", "blue", "green" ];
console.log(colors[0]);     // "red"
```

This code is equivalent to the following:

```
var colors = new Array("red", "blue", "green")
console.log(colors[0]);     // "red"
```

Function Literals

You almost always define functions using their literal form. In fact, using the `Function` constructor is typically discouraged given the challenges of maintaining, reading, and debugging a string of code rather than actual code, so you'll rarely see it in code.

Creating functions is much easier and less error prone when you use the literal form. For example:

```
function reflect(value) {
    return value;
}

// is the same as

var reflect = new Function("value", "return value;");
```

This code defines the `reflect()` function, which returns any value passed to it. Even in the case of this simple function, the literal form is easier to write and understand than the constructor form. Further, there is no good way to debug functions that are created in the constructor form: These functions aren't recognized by JavaScript debuggers and therefore act as a black box in your application.

Regular Expression Literals

JavaScript also has *regular expression literals* that allow you to define regular expressions without using the `RegExp` constructor. Regular expression literals look very similar to regular expressions in Perl: The pattern is contained between two slashes, and any additional options are single characters following the second slash. For example:

```
var numbers = /\d+/g;

// is the same as

var numbers = new RegExp("\\d+", "g");
```