JavaScript

Succinctly

by Cody Lindley

Syncfusion | Technology Resource Portal

JavaScript Succinctly

By
Cody Lindley

Foreword by Daniel Jebaraj

Important licensing information. Please read.

Edited by

This publication was edited by Jay Natarajan, senior product manager, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Cody Lindley is a client-side engineer (aka front-end developer) and recovering Flash developer. He has an extensive background working professionally (11+ years) with HTML, CSS, JavaScript, Flash, and client-side performance techniques as they pertain to web development. If he is not wielding client-side code he is likely toying with interface/interaction design or authoring material and speaking at various conferences. When not sitting in front of a computer, it is a sure bet he is hanging out with his wife and kids in Boise, Idaho—training for triathlons, skiing, mountain biking, road biking, alpine climbing, reading, watching movies, or debating the rational evidence for a Christian worldview.

Introduction

This book is not about JavaScript design patterns or implementing an object-oriented paradigm with JavaScript code. It was not written to distinguish the good features of the JavaScript language from the bad. It is not meant to be a complete reference guide. It is not targeted at people new to programming or those completely new to JavaScript. Nor is this a cookbook of JavaScript recipes. Those books have been written.

It was my intention to write a book that gives the reader an accurate JavaScript worldview through an examination of native JavaScript objects and supporting nuances: complex values, primitive values, scope, inheritance, the head object, etc. I intend this book to be a short and digestible summary of the ECMA-262, Edition 3 specification, focused on the nature of objects in JavaScript.

If you are a designer or developer who has only used JavaScript under the mantle of libraries (such as jQuery, Prototype, etc.), it is my hope that the material in this book will transform you from a JavaScript library user into a JavaScript developer.

Why did I write this book?

First, I must admit that I wrote this book for myself. Truth be told, I crafted this material so I could drink my own Kool-Aid and always remember what it tastes like. In other words, I wanted a reference written in my own words used to jog my memory as needed. Additionally:

- Libraries facilitate a "black box" syndrome that can be beneficial in some regards but detrimental in others. Things may be completed fast and efficiently, but you have no idea how or why. And the *how* and *why* really matter when things go wrong or performance becomes an issue. The fact is that anyone who intends to implement a JavaScript library or framework when building a web application (or just a good sign-up form) ought to look under the hood and understand the engine. This book was written for those who want to pop the hood and get their hands dirty in JavaScript itself.

- Mozilla has provided the most up-to-date and complete reference guide for JavaScript 1.5. I believe what is missing is a digestible document, written from a single point of view, to go along with their reference guide. It is my hope that this book will serve as a "what you need to know" manual for JavaScript values, detailing concepts beyond what the Mozilla reference covers.

- Version 1.5 of JavaScript is going to be around for a fair amount of time, but as we move toward the new additions to the language found in ECMA Edition 5, I wanted to document the cornerstone concepts of JavaScript that will likely be perennial.

- Advanced technical books written about programing languages are often full of monolithic code examples and pointless meanderings. I prefer short explanations that get to the point, backed by real code that I can run instantly. I coined a term, "technical thin-slicing," to describe what I am attempting to employ in this book. This entails reducing complex topics into smaller, digestible concepts taught with minimal words and backed with comprehensive and focused code examples.

- Most JavaScript books worth reading are three inches thick. Definitive guides like David Flanagan's certainly have their place, but I wanted to create a book that hones in on the important stuff without being exhaustive.

Who should read this book?

This book is targeted at two types of people. The first is an advanced beginner or intermediate JavaScript developer who wishes to solidify his or her understanding of the language through an in-depth look at JavaScript objects. The second type is a JavaScript library veteran who is ready to look behind the curtain. This book is not ideal for newbies to programming, JavaScript libraries, or JavaScript itself.

Why JavaScript 1.5 and ECMA-262 Edition 3?

In this book, I focus on version 1.5 of JavaScript (equivalent to ECMA-262 Edition 3) because it is the most widely implemented version of JavaScript to date. The next version of this book will certainly be geared toward the up-and-coming ECMA-262 Edition 5.

Why didn't I cover the `Date()`, `Error()`, or `RegEx()` objects?

Like I said, this book is not an exhaustive reference guide to JavaScript. Rather, it focuses on objects as a lens through which to understand JavaScript. So I have decided not to cover the `Date()`, `Error()`, or `RegEx()` objects because, as useful as they are, grasping the details of these objects will not make or break your general understanding of objects in JavaScript. My hope is that you simply apply what you learn here to all objects available in the JavaScript environment.

Preface

Before you begin, it is important to understand various styles employed in this book. Please do not skip this section because it contains important information that will aid you as you read the book.

More code, less words

Please examine the code examples in detail. The text should be viewed as secondary to the code itself. It is my opinion that a code example is worth a thousand words. Do not worry if you're initially confused by explanations. Examine the code. Tinker with it. Reread the code comments. Repeat this process until the concept being explained becomes clear. I hope you achieve a level of expertise such that well-documented code is all you need to understand a programming concept.

Exhaustive code and repetition

You will probably curse me for repeating myself and for being so comprehensive with my code examples. And while I might deserve it, I prefer to err on the side of being exact, verbose, and repetitive, rather than make false assumptions some authors often make about their readers. Yes, both approaches can be annoying depending upon what knowledge the author brings to the subject, but they can also serve a useful purpose for those who want to learn a subject in detail.

Color-coding conventions

Code will be colored using the normal JavaScript syntax highlighting in Visual Studio. This will help you understand the code, but you will be just fine reading this material on a monochrome e-book reader such as the Kindle Touch.

```
<!DOCTYPE html><html lang="en"><body><script>

    // This is a comment about a specific part of the code.
    var foo = 'calling out this part of the code';

</script></body></html>
```

In addition to syntax highlighting the code, the text in this book is colored so as to distinguish between JavaScript words and keywords, JavaScript code, and regular text. The following excerpt from the book demonstrates this coloring semantic.

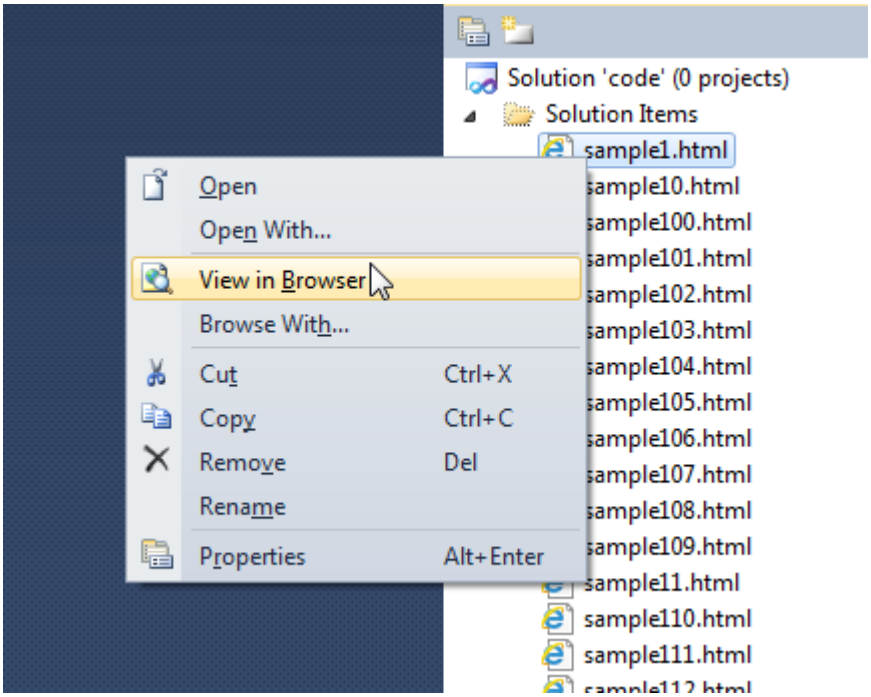
“Consider that the *cody* object created from the `Object()` constructor function is not really different from a string object created via the `String()` constructor function. To drive this fact home, examine and contrast the following code:”

Notice the use of gray italicized text for code references, orange text for JavaScript words and keywords, and regular black text for everything in-between.

Code examples

This book relies heavily on code examples to express JavaScript concepts. The code samples are available at https://bitbucket.org/syncfusion/javascript-succinctly.

The code samples are provided as individual HTML files. A Visual Studio 2010 project is also provided for easy navigation. You can select any file, right-click, and select the **View in Browser** option to test the code.



The name of the sample file is always included above its code block in the format **Sample: $file-name.html**.

Before reading this book, make sure you are comfortable with the usage and purpose of `console.log`. You can open the JavaScript console window in different browsers using the following keyboard shortcuts.

Browser	Keyboard shortcut to open JavaScript console window
Internet Explorer	F12 to open the developer tools, then Ctrl+3 to open the console window
Chrome	Ctrl+Shift+J
Firefox	Ctrl+Shift+K
Safari	Ctrl+Alt+I

I encourage you to download the code and follow along. I authored this book counting on the fact that you will need to tinker with the code while you are reading and learning.

Chapter 1 JavaScript Objects

Creating objects

In JavaScript, objects are king: Almost everything is an object or acts like an object. Understand objects and you will understand JavaScript. So let's examine the creation of objects in JavaScript.

An object is just a container for a collection of named values (aka properties). Before we look at any JavaScript code, let's first reason this out. Take myself, for example. Using plain language, we can express in a table, a "cody":

cody	
property	**property value**
living	True
age	33
gender	Male

The word "cody" in the table is just a label for the group of property names and corresponding values that make up exactly what a cody is. As you can see from the table, I am living, 33, and a male.

JavaScript, however, does not speak in tables. It speaks in objects, which are similar to the parts contained in the "cody" table. Translating the cody table into an actual JavaScript object would look like this:

Sample: sample1.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // Create the cody object
    var cody = new Object();

    // then fill the cody object with properties (using dot notation).
    cody.living = true;
    cody.age = 33;
    cody.gender = 'male';
```

```
    console.log(cody); // Logs Object {living = true, age = 33, gender =
'male'}

</script></body></html>
```

Keep this at the forefront of your mind: objects are really just containers for properties, each of which has a name and a value. This notion of a container of properties with named values (i.e. an object) is used by JavaScript as the building blocks for expressing values in JavaScript. The ***cody*** object is a value which I expressed as a JavaScript object by creating an object, giving the object a name, and then giving the object properties.

Up to this point, the ***cody*** object we are discussing has only static information. Since we are dealing with a programing language, we want to program our ***cody*** object to actually do something. Otherwise, all we really have is a database akin to JSON. In order to bring the ***cody*** object to life, I need to add a property *method*. Property methods perform a function. To be precise, in JavaScript, methods are properties that contain a **`Function()`** object, whose intent is to operate on the object the function is contained within.

If I were to update the ***cody*** table with a ***getGender*** method, in plain English it would look like this:

cody object	
property	**property value**
living	True
age	33
gender	Male
getGender	return the value of gender

Using JavaScript, the ***getGender*** method from the updated cody table would look like so:

Sample: sample2.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var cody = new Object();
    cody.living = true;
    cody.age = 33;
    cody.gender = 'male';
    cody.getGender = function () { return cody.gender; };

    console.log(cody.getGender()); // Logs 'male'.

</script></body></html>
```

The ***getGender*** method, a property of the ***cody*** object, is used to return one of ***cody***'s other property values: the value "male" stored in the ***gender*** property. What you must realize is that without methods, our object would not do much except store static properties.

The ***cody*** object we have discussed thus far is what is known as an **Object()** object. We created the ***cody*** object using a blank object that was provided to us by invoking the **Object()** constructor function. Think of constructor functions as a template or cookie cutter for producing predefined objects. In the case of the ***cody*** object, I used the **Object()** constructor function to produce an empty object which I named ***cody***. Because ***cody*** is an object constructed from the **Object()** constructor, we call ***cody*** an **Object()** object. What you really need to understand, beyond the creation of a simple **Object()** object like ***cody***, is that the majority of values expressed in JavaScript are objects (primitive values like "foo", 5, and **true** are the exception but have equivalent wrapper objects).

Consider that the ***cody*** object created from the **Object()** constructor function is not really different from a string object created via the **String()** constructor function. To drive this fact home, examine and contrast the following code:

Sample: sample3.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myObject = new Object(); // Produces an Object() object.
    myObject['0'] = 'f';
    myObject['1'] = 'o';
    myObject['2'] = 'o';

    console.log(myObject); // Logs Object { 0="f", 1="o", 2="o"}
```

```
    var myString = new String('foo'); // Produces a String() object.

    console.log(myString); // Logs foo { 0="f", 1="o", 2="o"}

</script></body></html>
```

As it turns out, *myObject* and *myString* are both . . . objects! They both can have properties, inherit properties, and are produced from a constructor function. The *myString* variable containing the *'foo'* string value seems to be as simple as it goes, but amazingly it's got an object structure under its surface. If you examine both of the objects produced you will see that they are identical objects in substance but not in type. More importantly, I hope you begin to see that JavaScript uses objects to express values.

Notes

You might find it odd to see the string value *'foo'* in object form because typically a string is represented in JavaScript as a primitive value (e.g., **var myString = 'foo';**). I specifically used a string object value here to highlight that anything can be an object, including values that we might not typically think of as an object (e.g., string, number, Boolean). Also, I think this helps explain why some say that everything in JavaScript can be an object.

JavaScript bakes the **String()** and **Object()** constructor functions into the language itself to make the creation of a **String()** object and **Object()** object trivial. But you, as a coder of the JavaScript language, can also create equally powerful constructor functions. In the following sample, I demonstrate this by defining a non-native custom *Person()* constructor function so that I can create people from it.

Sample: sample4.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // Define Person constructor function in order to create custom Person()
objects later.
    var Person = function (living, age, gender) {
        this.living = living;
        this.age = age;
        this.gender = gender;
        this.getGender = function () { return this.gender; };
    };

    // Instantiate a Person object and store it in the cody variable.
    var cody = new Person(true, 33, 'male');

    console.log(cody);
```

```
    /* The String() constructor function that follows, having been defined by
JavaScript, has the same pattern. Because the string constructor is native to
JavaScript, all we have to do to get a string instance is instantiate it. But
the pattern is the same whether we use native constructors like String() or
user-defined constructors like Person(). */

    // Instantiate a String object stored in the myString variable.
    var myString = new String('foo');

    console.log(myString);

</script></body></html>
```

The user-defined ***Person()*** constructor function can produce ***Person*** objects, just as the native **String()** constructor function can produce string objects. The ***Person()*** constructor is no less capable, and is no more or less malleable, than the native **String()** constructor or any of the native constructors found in JavaScript.

Remember how the ***cody*** object we first looked at was produced from an **Object()**. It's important to note that the **Object()** constructor function and the new ***Person()*** constructor shown in the previous code example can give us identical outcomes. Both can produce an identical object with the same properties and property methods. Examine the two sections of code that follow, showing that ***codyA*** and ***codyB*** have the same object values even though they are produced in different ways.

Sample: sample5.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // Create a codyA object using the Object() constructor.

    var codyA = new Object();
    codyA.living = true;
    codyA.age = 33;
    codyA.gender = 'male';
    codyA.getGender = function () { return codyA.gender; };

    console.log(codyA); // Logs Object {living=true, age=33, gender="male",
...}

    /* The same cody object is created below, but instead of using the native
Object() constructor to create a one-off cody, we first define our own
Person() constructor that can create a cody object (and any other Person
object we like) and then instantiate it with "new". */

    var Person = function (living, age, gender) {
        this.living = living;
```

```
        this.age = age;
        this.gender = gender;
        this.getGender = function () { return this.gender; };
    };

    var codyB = new Person(true, 33, 'male');

    console.log(codyB); // Logs Object {living=true, age=33, gender="male",
...}

</script></body></html>
```

The main difference between the ***codyA*** and ***codyB*** objects is not found in the object itself, but in the constructor functions used to produce the objects. The ***codyA*** object was produced using an instance of the **`Object()`** constructor. The ***`Person()`*** constructor produced ***codyB***, but can also be used as a powerful, centrally defined object "factory" to be used for creating more ***`Person()`*** objects. Crafting your own constructors for producing custom objects also sets up prototypal inheritance for ***`Person()`*** instances.

Both solutions resulted in the same complex object being created. It's these two patterns that are most commonly used for constructing objects.

JavaScript is really just a language that is prepackaged with a few native object constructors used to produce complex objects which express a very specific type of value (e.g., numbers, strings, functions, objects, arrays, etc.), as well as the raw materials via **`Function()`** objects for crafting user-defined object constructors (e.g., ***`Person()`***). The end result—no matter the pattern for creating the object—is typically the creation of a complex object.

Understanding the creation, nature, and usage of objects and their primitive equivalents is the focus of the rest of this book.

JavaScript constructors create and return object instances

The role of a constructor function is to create multiple objects that share certain qualities and behaviors. Basically, a constructor function is a cookie cutter for producing objects that have default properties and property methods.

If you said, "A constructor is nothing more than a function," then I would reply, "You are correct—unless that function is invoked using the **`new`** keyword." (For example, **`new String('foo')`**). When this happens, a function takes on a special role, and JavaScript treats the function as special by setting the value of **`this`** for the function to the new object that is being constructed. In addition to this special behavior, the function will return the newly created object (i.e. **`this`**) by default instead of the value **`false`**. The

new object that is returned from the function is considered to be an instance of the constructor function that constructs it.

Consider the ***Person()*** constructor again, but this time read the comments in the following code sample carefully, as they highlight the effect of the **new** keyword.

Sample: sample6.html

```
<!DOCTYPE html><html lang="en"><body><script>

    /* Person is a constructor function. It was written with the intent of
being used with the new keyword. */
    var Person = function Person(living, age, gender) {
        // "this" below is the new object that is being created (i.e. this =
new Object();)
        this.living = living;
        this.age = age;
        this.gender = gender;
        this.getGender = function () { return this.gender; };
        // When the function is called with the new keyword, "this" is
returned instead of false.
    };

    // Instantiate a Person object named cody.
    var cody = new Person(true, 33, 'male');

    // cody is an object and an instance of Person()
    console.log(typeof cody); // Logs object.
    console.log(cody); // Logs the internal properties and values of cody.
    console.log(cody.constructor); // Logs the Person() function.

</script></body></html>
```

The sample6.html code leverages a user-defined constructor function (i.e. **Person()**) to create the ***cody*** object. This is no different from the **Array()** constructor creating an **Array()** object (e.g., **new Array()**) in the following code.

Sample: sample7.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // Instantiate an Array object named myArray.
    var myArray = new Array(); // myArray is an instance of Array.

    // myArray is an object and an instance of the Array() constructor.
    console.log(typeof myArray); // Logs object! What? Yes, arrays are a type
of object.
```

```
    console.log(myArray); // Logs [ ]

    console.log(myArray.constructor); // Logs Array()

</script></body></html>
```

In JavaScript, most values (excluding primitive values) involve objects being created, or *instantiated*, from a constructor function. An object returned from a constructor is called an *instance*. Make sure you are comfortable with these semantics, as well as the pattern of leveraging constructors to produce objects.

The native JavaScript object constructors

The JavaScript language contains nine native (or built-in) object constructors. These objects are used by JavaScript to construct the language, and by "construct" I mean these objects are used to express object values in JavaScript code, as well as orchestrate several features of the language. Thus, the native object constructors are multifaceted in that they produce objects, but are also leveraged in facilitating many of the language's programming conventions. For example, functions are objects created from the `Function()` constructor, but are also used to create other objects when called as constructor functions using the `new` keyword.

The nine native object constructors that come prepackaged with JavaScript are:

- `Number()`
- `String()`
- `Boolean()`
- `Object()`
- `Array()`
- `Function()`
- `Date()`
- `RegExp()`
- `Error()`

JavaScript is mostly constructed from these nine objects (as well as string, number, and Boolean primitive values). Understanding these objects in detail is key to taking advantage of JavaScript's unique programming power and language flexibility.

Notes
The `Math` object is the oddball here. It's a static object rather than a constructor function, meaning you can't do this: `var x = new Math()`. But you can use it as if it has already been instantiated (e.g., `Math.PI`). Truly, `Math` is just an object namespace set up by JavaScript to house math functions.

The native objects are sometimes referred to as "global objects" since they are the objects that JavaScript has made natively available for use. Do not confuse the term *global object* with the "head" global object that is the topmost level of the scope chain, for example, the `window` object in all web browsers.

The `Number()`, `String()`, and `Boolean()` constructors not only construct objects; they also provide a primitive value for a string, number, and Boolean, depending upon how the constructor is leveraged. If you call these constructors directly, then a complex object is returned. If you simply express a number, string, or Boolean value in your code (primitive values like 5, "foo", and `true`), then the constructor will return a primitive value instead of a complex object value.

User-defined/non-native object constructor functions

As you saw with the *`Person()`* constructor, we can make our own constructor functions from which we can produce not just one, but *multiple* custom objects.

In the following sample, I present the familiar `Person()` constructor function:

Sample: sample8.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var Person = function (living, age, gender) {
        this.living = living;
        this.age = age;
        this.gender = gender;
        this.getGender = function () { return this.gender; };
    };

    var cody = new Person(true, 33, 'male');
    console.log(cody); // Logs Object {living=true, age=33, gender="male",
...}

    var lisa = new Person(true, 34, 'female');
    console.log(lisa); // Logs Object {living=true, age=34, gender="female",
...}

</script></body></html>
```

As you can see, by passing unique parameters and invoking the *`Person()`* constructor function, you could easily create a vast number of unique people objects. This can be

pretty handy when you need more than two or three objects that possess the same properties, but with different values. Come to think of it, this is exactly what JavaScript does with the native objects. The `Person()` constructor follows the same principles as the `Array()` constructor. So `new Array('foo','bar')` is really not that different than `new Person(true, 33, 'male')`. Creating your own constructor functions is just using the same pattern that JavaScript itself uses for its own native constructor functions.

Notes
It is not required, but when creating custom constructor functions intended to be used with the `new` operator, it's best practice to make the first character of the constructor name uppercase: `Person()` rather than `person()`.

One tricky thing about constructor functions is the use of the `this` value inside of the function. Remember, a constructor function is just a cookie cutter. When used with the `new` keyword, it will create an object with properties and values defined inside of the constructor function. When `new` is used, the value `this` literally means the new object or instance that will be created based on the statements inside the constructor function. On the other hand, if you create a constructor function and call it without the use of the `new` keyword, the `this` value will refer to the "parent" object that contains the function. More detail about this topic can be found in Chapter 6.

It's possible to forgo the use of the `new` keyword and the concept of a constructor function by explicitly having the function return an object. The function would have to be written explicitly to build an `Object()` object and return it: `var myFunction = function() {return {prop: val}};`.

Instantiating constructors using the `new` operator

A constructor function is basically a cookie-cutter template used to create pre-configured objects. Take `String()` for example. This function, when used with the `new` operator (`new String('foo')`), creates a string instance based on the `String()` "template." Let's look at an example.

Sample: sample9.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myString = new String('foo');

    console.log(myString); // Logs foo {0 = "f", 1 = "o", 2 = "o"}

</script></body></html>
```

In this snippet, we created a new string object that is an instance of the `String()` constructor function. Just like that, we have a string value expressed in JavaScript.

Notes
I'm not suggesting that you use constructor functions instead of their literal/primitive equivalents—like `var string="foo";`. I am, however, suggesting that you understand what is going on behind literal/primitive values.

As previously mentioned, the JavaScript language has the following native predefined constructors: `Number()`, `String()`, `Boolean()`, `Object()`, `Array()`, `Function()`, `Date()`, `RegExp()`, and `Error()`. We can instantiate an object instance from any of these constructor functions by applying the `new` operator. In the following sample, I construct these nine native JavaScript objects.

Sample: sample10.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // Instantiate an instance for each native constructor using the new
keyword.
    var myNumber = new Number(23);
    var myString = new String('male');
    var myBoolean = new Boolean(false);
    var myObject = new Object();
    var myArray = new Array('foo', 'bar');
    var myFunction = new Function("x", "y", "return x*y");
    var myDate = new Date();
    var myRegExp = new RegExp('\bt[a-z]+\b');
    var myError = new Error('Darn!');

    // Log/verify which constructor created the object.
    console.log(myNumber.constructor); // Logs Number()
    console.log(myString.constructor); // Logs String()
    console.log(myBoolean.constructor); // Logs Boolean()
    console.log(myObject.constructor); // Logs Object()
    console.log(myArray.constructor); // Logs Array() in modern browsers.
    console.log(myFunction.constructor); // Logs Function()
    console.log(myDate.constructor); // Logs Date()
    console.log(myRegExp.constructor); // Logs RegExp()
    console.log(myError.constructor); // Logs Error()

</script></body></html>
```

By using the `new` operator, we are telling the JavaScript interpreter that we would like an object that is an instance of the corresponding constructor function. For example, in the code sample, the `Date()` constructor function is used to create date objects. The

`Date()` constructor function is a cookie cutter for date objects. That is, it produces date objects from a default pattern defined by the `Date()` constructor function.

At this point, you should be well acquainted with creating object instances from native constructor functions (e.g., `new String('foo')`) and user-defined constructor functions (e.g, `new Person(true, 33, 'male')`).

Notes
Keep in mind that `Math` is a static object—a container for other methods—and is not a constructor that uses the `new` operator.

Creating shorthand or literal values from constructors

JavaScript provides shortcuts—called "literals"—for manufacturing most of the native object values without having to use `new Foo()` or `new Bar()`. For the most part, the literal syntax accomplishes the same thing as using the `new` operator. The exceptions are: `Number()`, `String()`, and `Boolean()`—see the notes after the following sample.

If you come from other programming backgrounds, you are likely more familiar with the literal way of creating objects. In the following sample, I instantiate the native JavaScript constructors using the `new` operator and then create corresponding literal equivalents.

Sample: sample11.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myNumber = new Number(23); // An object.
    var myNumberLiteral = 23; // Primitive number value, not an object.

    var myString = new String('male'); // An object.
    var myStringLiteral = 'male'; // Primitive string value, not an object.

    var myBoolean = new Boolean(false); // An object.
    var myBooleanLiteral = false; // Primitive boolean value, not an object.

    var myObject = new Object();
    var myObjectLiteral = {};

    var myArray = new Array('foo', 'bar');
    var myArrayLiteral = ['foo', 'bar'];

    var myFunction = new Function("x", "y", "return x*y");
    var myFunctionLiteral = function (x, y) { return x * y };

    var myRegExp = new RegExp('\bt[a-z]+\b');
    var myRegExpLiteral = /\bt[a-z]+\b/;
```

```
    // Verify that literals are created from same constructor.
    console.log(myNumber.constructor, myNumberLiteral.constructor);
    console.log(myString.constructor, myStringLiteral.constructor);
    console.log(myBoolean.constructor, myBooleanLiteral.constructor);
    console.log(myObject.constructor, myObjectLiteral.constructor);
    console.log(myArray.constructor, myArrayLiteral.constructor);
    console.log(myFunction.constructor, myFunctionLiteral.constructor);
    console.log(myRegExp.constructor, myRegExpLiteral.constructor);

</script></body></html>
```

What you need to take away here is the fact that, in general, using literals simply conceals the underlying process identical to using the **`new`** operator. Maybe more importantly, it's much more convenient!

Okay, things are a little more complicated with respect to the primitive string, number, and Boolean values. In these cases, literal values take on the characteristics of primitive values rather than complex object values. See the notes that follow.

Notes
When using literal values for **`String()`**, **`Number()`**, and **`Boolean()`**, an actual complex object is never created until the value is treated as an object. In other words, you are dealing with a primitive data type until you attempt to use methods or retrieve properties associated with the constructor (e.g., **`var charactersInFoo = 'foo'.length`**). When this happens, JavaScript creates a wrapper object for the literal value behind the scenes, allowing the value to be treated as an object. Then, after the method is called, JavaScript discards the wrapper object and the value returns to a literal type. This is why string, number, and Boolean are considered primitive (or simple) data types. I hope this clarifies the misconception "everything in JavaScript is an object" from the concept "everything in JavaScript can act like an object."

Primitive (aka simple) values

The JavaScript values ***`5`***, ***`'foo'`***, **`true`**, and **`false`** , as well as **`null`** and **`undefined`**, are considered primitive because they are *immutable*. That is, a number is a number, a string is a string, a Boolean is either true or false, and **`null`** and **`undefined`** are just that, **`null`** and **`undefined`**. These values are inherently simple and do not represent values that can be made up of other values.

Examine the following code and ask yourself if the string, number, Boolean, **`null`**, and **`undefined`** values could be more complex. Contrast this to what you know of an **`Object()`**instance, **`Array()`**instance, or really any complex object.

Sample: sample12.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myString = 'string'
    var myNumber = 10;
    var myBoolean = false; // Could be true or false, but that is it.
    var myNull = null;
    var myUndefined = undefined;

    console.log(myString, myNumber, myBoolean, myNull, myUndefined);

    /* Consider that a complex object like array or object can be made up of
multiple primitive values, and thus becomes a complex set of multiple values.
*/

    var myObject = {
        myString: 'string',
        myNumber: 10,
        myBoolean: false,
        myNull: null,
        myUndefined: undefined
    };

    console.log(myObject);

    var myArray = ['string', 10, false, null, undefined];

    console.log(myArray);

</script></body></html>
```

Quite simply, primitive values represent the lowest form (i.e. simplest) of data and information available in JavaScript.

Notes

As opposed to creating values with literal syntax, when a **`String()`**, **`Number()`**, or **`Boolean()`** value is created using the **`new`** keyword, the object created is actually a complex object.

It's critical that you understand the fact that the **`String()`**, **`Number()`**, and **`Boolean()`** constructors are dual-purpose constructors used to create literal/primitive values as well as complex values. These constructors do not always return objects, but instead, when used without the "**`new`**" operator, can return a primitive representation of the actual complex object value.

The primitive values `null`, `undefined`, "string", 10, `true`, and `false` are not objects

The `null` and `undefined` values are such trivial values that they do not require a constructor function, nor the use of the `new` operator to establish them as a JavaScript value. To use `null` or `undefined`, all you do is use them as if they were an operator. The remaining primitive values—string, number, and Boolean—while technically returned from a constructor function, are not objects.

In the following sample, I contrast the difference between primitive values and the rest of the native JavaScript objects.

Sample: sample13.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // No object is created when producing primitive values; notice no use of
the "new" keyword.
    var primitiveString1 = "foo";
    var primitiveString2 = String('foo');
    var primitiveNumber1 = 10;
    var primitiveNumber2 = Number('10');
    var primitiveBoolean1 = true;
    var primitiveBoolean2 = Boolean('true');

    // Confirm the typeof is not object.
    console.log(typeof primitiveString1, typeof primitiveString2); // Logs
'string,string'.
    console.log(typeof primitiveNumber1, typeof primitiveNumber2); // Logs
'number,number'.
    console.log(typeof primitiveBoolean1, typeof primitiveBoolean2); // Logs
'Boolean,Boolean'.

    // Using a constructor and the "new" keyword for creating objects.

    var myNumber = new Number(23);
    var myString = new String('male');
    var myBoolean = new Boolean(false);
    var myObject = new Object();
    var myArray = new Array('foo', 'bar');
    var myFunction = new Function("x", "y", "return x * y");
    var myDate = new Date();
    var myRegExp = new RegExp('\\bt[a-z]+\\b');
    var myError = new Error('Darn!');

    // Logs 'object object object object object function object function
object'.
    console.log(
        typeof myNumber,
```

```
        typeof myString,
        typeof myBoolean,
        typeof myObject,
        typeof myArray,
        typeof myFunction, // BE AWARE typeof returns function for all
function objects.
        typeof myDate,
        typeof myRegExp, // BE AWARE typeof returns function for RegExp()
        typeof myError
        );

</script></body></html>
```

What I would like you to learn from the previous code example is that primitive values are not objects. Primitive values are special in that they are used to represent simple values.

How primitive values are stored/copied in JavaScript

It is extremely important to understand that primitive values are stored and manipulated at "face value." It might sound simple, but this means that if I store the string value *"foo"* in a variable called *myString*, then the value *"foo"* is literally stored in memory as such. Why is this important? Once you begin manipulating (e.g., copying) values, you have to be equipped with this knowledge, because primitive values are copied literally.

In the following example, we store a copy of the *myString* value (*'foo'*) in the variable *myStringCopy*, and its value is literally copied. Even if we change the original value, the copied value, referenced by the variable *myStringCopy*, remains unchanged.

Sample: sample14.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myString = 'foo' // Create a primitive string object.
    var myStringCopy = myString; // Copy its value into a new variable.
    var myString = null; // Manipulate the value stored in the myString
variable.

    /*The original value from myString was copied to myStringCopy. This is
confirmed by updating the value of myString then checking the value of
myStringCopy.*/

    console.log(myString, myStringCopy); // Logs 'null foo'

</script></body></html>
```

The concept to take away here is that primitive values are stored and manipulated as immutable *values*. Referring to them transfers their value. In the previous example, we *copied, or cloned,* the ***myString*** value to the variable ***myStringCopy***. When we updated the ***myString*** value, the ***myStringCopy*** value still had a copy of the old ***myString*** value. Remember this and contrast the mechanics here with complex objects (discussed in the following section).

Primitive values are equal by value

Primitives can be compared to see if their values are literally the same. As logic would suggest, if you compare a variable containing the numeric value ***10*** with another variable containing the numeric value ***10***, JavaScript will consider these equal because ***10*** is the same as ***10*** (i.e. ***10 === 10***). The same, of course, would apply if you compare the primitive string ***'foo'*** to another primitive string with a value of ***'foo'***. The comparison would say that they are equal to each other based on their value (i.e. ***'foo' === 'foo'***).

In the following code, I demonstrate the "equal by value" concept using primitive numbers, as well as contrast this with a complex number object.

Sample: sample15.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var price1 = 10;
    var price2 = 10;
    var price3 = new Number('10'); // A complex numeric object because new
was used.
    var price4 = price3;

    console.log(price1 === price2); // Logs true.

    /* Logs false because price3 contains a complex number object and price 1
is a primitive value. */
    console.log(price1 === price3);

    // Logs true because complex values are equal by reference, not value.
    console.log(price4 === price3);

    // What if we update the price4 variable to contain a primitive value?
    price4 = 10;

    console.log(price4 === price3); // Logs false: price4 is now primitive
rather than complex.

</script></body></html>
```

The concept to take away here is that primitives, when compared, will check to see if the expressed *values* are equal. When a string, number, or Boolean value is created using the `new` keyword (e.g., `new Number('10')`), the value is no longer primitive. As such, comparison does not work the same as if the value had been created via literal syntax. This is not surprising, given that primitive values are stored by value (i.e. does `10 === 10`), while complex values are stored by reference (i.e. does price3 and price4 contain a reference to the same value).

The string, number, and Boolean primitive values act like objects when used like objects

When a primitive value is used as if it were an object created by a constructor, JavaScript converts it to an object in order to respond to the expression at hand, but then discards the object qualities and changes it back to a primitive value. In the code that follows, I take primitive values and showcase what happens when the values are treated like objects.

Sample: sample16.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // Produce primitive values.
    var myNull = null;
    var myUndefined = undefined;
    var primitiveString1 = "foo";
    var primitiveString2 = String('foo'); // Did not use new, so we get
primitive.
    var primitiveNumber1 = 10;
    var primitiveNumber2 = Number('10'); // Did not use new, so we get
primitive.
    var primitiveBoolean1 = true;
    var primitiveBoolean2 = Boolean('true'); // Did not use new, so we get
primitive.

    /* Access the toString() property method (inherited by objects from
object.prototype) to demonstrate that the primitive values are converted to
objects when treated like objects. */

    // Logs "string string"
    console.log(primitiveString1.toString(), primitiveString2.toString());

    // Logs "number number"
    console.log(primitiveNumber1.toString(), primitiveNumber2.toString());

    // Logs "boolean boolean"
    console.log(primitiveBoolean1.toString(), primitiveBoolean2.toString());
```

```
    /* This will throw an error and not show up in Firebug Lite, as null and
undefined do not convert to objects and do not have constructors. */

    console.log(myNull.toString());
    console.log(myUndefined.toString());

</script></body></html>
```

In this code example, all of the primitive values (except **`null`** and **`undefined`**) are converted to objects, so as to leverage the **`toString()`** method, and then are returned to primitive values once the method is invoked and returned.

Complex (aka composite) values

The native object constructors **`Object()`**, **`Array()`**, **`Function()`**, **`Date()`**, **`Error()`**, and **`RegExp()`** are complex because they can contain one or more primitive or complex values. Essentially, complex values can be made up of many different types of JavaScript objects. It could be said that complex objects have an unknown size in memory because complex objects can contain any value and not a specific known value. In the following code, we create an object and an array that houses all of the primitive objects.

Sample: sample17.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var object = {
        myString: 'string',
        myNumber: 10,
        myBoolean: false,
        myNull: null,
        myUndefined: undefined
    };

    var array = ['string', 10, false, null, undefined];

    /* Contrast this to the simplicity of the primitive values below. In a
primitive form, none of the values below can be more complex than what you
see while complex values can encapsulate any of the JavaScript values (seen
above). */

    var myString = 'string';
    var myNumber = 10;
    var myBoolean = false;
    var myNull = null;
    var myUndefined = undefined;
```

```
</script></body></html>
```

The concept to take away here is that complex values are a composite of values and differ in complexity and composition to primitive values.

Notes
The term "complex object" has also been expressed in other writings as "composite objects" or "reference types." If it's not obvious, all these names describe the nature of a JavaScript value excluding primitive values. Primitive values are not "referenced by value" and cannot represent a composite (i.e. a thing made up of several parts or elements) of other values, while complex objects are "referenced by value" and can contain or encapsulate other values.

How complex values are stored/copied in JavaScript

It is extremely important to understand that complex values are stored and manipulated by *reference*. When creating a variable containing a complex object, the value is stored in memory at an address. When you reference a complex object, you're using its name (i.e. variable or object property) to retrieve the value at that address in memory. The implications are significant when you consider what happens when you attempt to copy a complex value. In the next sample, we create an object stored in the variable ***myObject***. The value in ***myObject*** is then copied to the variable ***copyOfMyObject***. Really, it is not a copy of the object—more like a copy of the address of the object.

Sample: sample18.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myObject = {};

    var copyOfMyObject = myObject; // Not copied by value, just the reference
is copied.

    myObject.foo = 'bar'; // Manipulate the value stored in myObject.

    /* If we log myObject and copyOfMyObject, they will have a foo property
because they reference the same object. */

    console.log(myObject, copyOfMyObject); // Logs 'Object { foo="bar"}
Object { foo="bar"}'

</script></body></html>
```

What you need to realize is that, unlike primitive values that would copy a value, objects (aka complex values) are stored by reference. As such, the reference (aka address) is

copied, but not the actual value. This means that objects are not copied at all. Like I said, what is copied is the address or reference to the object in the memory stack. In our code example, *myObject* and *copyOfMyObject* point to the same object stored in memory.

The idea to take away here is that when you change a complex value—because it is stored by reference—you change the value stored in all variables that reference the complex value. In our code example, both *myObject* and *copyOfMyObject* are changed when you update the object stored in either variable.

Notes
When the values **String()**, **Number()**, and **Boolean()** are created using the **new** keyword, or converted to complex objects behind the scenes, the values continue to be stored/copied by value. So, even though primitive values can be treated like complex values, they do not take on the quality of being copied by reference.

To truly make a copy of an object, you have to extract the values from the old object and inject them into a new object.

Complex objects are equal by reference

When comparing complex objects, they are equal only when they reference the same object (i.e. have the same address). Two variables containing identical objects are not equal to each other since they do not actually point at the same object.

In the following sample, *objectFoo* and *objectBar* have the same properties and are, in fact, identical objects, but when asked if they are equal via ===, JavaScript tells us they are not.

Sample: sample19.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var objectFoo = { same: 'same' };
    var objectBar = { same: 'same' };

    // Logs false, JS does not care that they are identical and of the same
object type.
    console.log(objectFoo === objectBar);

    // How complex objects are measured for equality.
    var objectA = { foo: 'bar' };
    var objectB = objectA;

    console.log(objectA === objectB); // Logs true because they reference the
same object.
```

```
</script></body></html>
```

The concept to take away here is that variables that point to a complex object in memory are equal only because they are using the same "address." Conversely, two independently created objects are not equal even if they are of the same type and possess the exact same properties.

Complex objects have dynamic properties

A new variable that points to an existing complex object does not copy the object. This is why complex objects are sometimes called reference objects. A complex object can have as many references as you want, and they will always refer to the same object, even as the object being referenced changes.

Sample: sample20.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var objA = { property: 'value' };
    var pointer1 = objA;
    var pointer2 = pointer1;

    // Update the objA.property, and all references (pointer1 and pointer2)
are updated.
    objA.property = null;

    // Logs 'null null null' because objA, pointer1, and pointer2 all
reference the same object.
    console.log(objA.property, pointer1.property, pointer2.property);

</script></body></html>
```

This allows for dynamic object properties because you can define an object, create references, update the object, and all of the variables referring to the object will "get" that update.

The `typeof` operator used on primitive and complex values

The **`typeof`** operator can be used to return the type of value you are dealing with. But the values returned from it are not exactly consistent or what some might say, logical. The following code exhibits the returned values from using the **`typeof`** operator.

Sample: sample21.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // Primitive values.
    var myNull = null;
    var myUndefined = undefined;
    var primitiveString1 = "string";
    var primitiveString2 = String('string');
    var primitiveNumber1 = 10;
    var primitiveNumber2 = Number('10');
    var primitiveBoolean1 = true;
    var primitiveBoolean2 = Boolean('true');

    console.log(typeof myNull); // Logs object? WHAT? Be aware...
    console.log(typeof myUndefined); // Logs undefined.
    console.log(typeof primitiveString1, typeof primitiveString2); // Logs
string string.
    console.log(typeof primitiveNumber1, typeof primitiveNumber2); // Logs
number number
    console.log(typeof primitiveBoolean1, typeof primitiveBoolean2); // Logs
boolean boolean.

    // Complex values.
    var myNumber = new Number(23);
    var myString = new String('male');
    var myBoolean = new Boolean(false);
    var myObject = new Object();
    var myArray = new Array('foo', 'bar');
    var myFunction = new Function("x", "y", "return x * y");
    var myDate = new Date();
    var myRegExp = new RegExp('\\bt[a-z]+\\b');
    var myError = new Error('Darn!');

    console.log(typeof myNumber); // Logs object.
    console.log(typeof myString); // Logs object.
    console.log(typeof myBoolean); // Logs object.
    console.log(typeof myObject); // Logs object.
    console.log(typeof myArray); // Logs object.
    console.log(typeof myFunction); // Logs function? WHAT? Be aware...
    console.log(typeof myDate); // Logs object.
    console.log(typeof myRegExp); // Logs function? WHAT? Be aware...
    console.log(typeof myError); // Logs object.

</script></body></html>
```

When using this operator on values, you should be aware of the potential values returned given the type of value (primitive or complex) that you are dealing with.

Dynamic properties allow for mutable objects

Complex objects are made up of dynamic properties. This allows user-defined objects, and most of the native objects, to be mutated. This means that the majority of objects in JavaScript can be updated or changed at any time. Because of this, we can change the native pre-configured nature of JavaScript itself by augmenting its native objects. However, I am not telling you to do this; in fact I do not think you should. But let's not cloud what is possible with opinions.

This means it's possible to store properties on native constructors and add new methods to the native objects with additions to their prototype objects.

In the following code, I mutate the **`String()`** constructor function and **`String.prototype`**.

Sample: sample22.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // Augment the built-in String constructor Function() with the
augmentedProperties property.
    String.augmentedProperties = [];

    if (!String.prototype.trimIT) { // If the prototype does not have
trimIT() add it.
        String.prototype.trimIT = function () {
            return this.replace(/^\s+|\s+$/g, '');
        }

        // Now add trimIT string to the augmentedProperties array.
        String.augmentedProperties.push('trimIT');
    }
    var myString = '  trim me  ';
    console.log(myString.trimIT()); // Invoke our custom trimIT string
method, logs 'trim me'.

    console.log(String.augmentedProperties.join()); // Logs 'trimIT'.

</script></body></html>
```

I want to drive home the fact that objects in JavaScript are dynamic. This allows objects in JavaScript to be mutated. Essentially, the entire language can be mutated into a custom version (e.g., ***`trimIT`*** string method). Again, I am not recommending this—I am just pointing out that it is part of the nature of objects in JavaScript.

Notes
Careful! If you mutate the native inner workings of JavaScript, you potentially have a custom version of JavaScript to deal with. Proceed with caution, as most people will assume that JavaScript is the same wherever it's available.

All constructor instances have constructor properties that point to their constructor function

When any object is instantiated, the **`constructor`** property is created behind the scenes as a property of that object or instance. This property points to the constructor function that created the object. In the next code sample, we create an **`Object()`** object, stored in the ***`foo`*** variable, and then verify that the **`constructor`** property is available for the object we created.

Sample: sample23.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var foo = {};

    console.log(foo.constructor === Object) // Logs true, because object()
constructed foo.
    console.log(foo.constructor) // Points to the Object() constructor
function.

</script></body></html>
```

This can be useful: If I'm working with some instance and I can't see who or what created it (especially if it is someone else's code), I can determine if it's an array, an object, or whatever.

In the following sample, you can see that I have instantiated most of the pre-configured objects that come included with the JavaScript language. Note that using literal or primitive values does not mean that the **`constructor`** pointer is not resolved when the primitive literal value is treated as an object.

Sample: sample24.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myNumber = new Number('23');
    var myNumberL = 23; // Literal shorthand.
    var myString = new String('male');
    var myStringL = 'male'; // Literal shorthand.
    var myBoolean = new Boolean('true');
    var myBooleanL = true; // Literal shorthand.
    var myObject = new Object();
```

```
    var myObjectL = {}; // Literal shorthand.
    var myArray = new Array();
    var myArrayL = []; // Literal shorthand.
    var myFunction = new Function();
    var myFunctionL = function () { }; // Literal shorthand.
    var myDate = new Date();
    var myRegExp = new RegExp('/./');
    var myRegExpL = /./; // Literal shorthand.
    var myError = new Error();

    console.log( // All of these return true.
          myNumber.constructor === Number,
          myNumberL.constructor === Number,
          myString.constructor === String,
          myStringL.constructor === String,
          myBoolean.constructor === Boolean,
          myBooleanL.constructor === Boolean,
          myObject.constructor === Object,
          myObjectL.constructor === Object,
          myArray.constructor === Array,
          myArrayL.constructor === Array,
          myFunction.constructor === Function,
          myFunctionL.constructor === Function,
          myDate.constructor === Date,
          myRegExp.constructor === RegExp,
          myRegExpL.constructor === RegExp,
          myError.constructor === Error
    );

</script></body></html>
```

The **constructor** property also works on user-defined constructor functions. In the following sample, we define a ***CustomConstructor()*** constructor function, then using the keyword **new**, we invoke the function to produce an object. Once we have our object, we can then leverage the **constructor** property.

Sample: sample25.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var CustomConstructor = function CustomConstructor() { return 'Wow!'; };
    var instanceOfCustomObject = new CustomConstructor();

    // Logs true.
    console.log(instanceOfCustomObject.constructor === CustomConstructor);

    // Returns a reference to CustomConstructor() function.
    // Returns 'function() { return 'Wow!'; };'
```

```
    console.log(instanceOfCustomObject.constructor);

</script></body></html>
```

Notes
You might be confused as to why primitive values have constructor properties that point to constructor functions when objects are not returned. By using a primitive value, the constructor is still called, so there is still a relationship with primitive values and constructor functions. However, the end result is a primitive value.

If you would like the **`constructor`** property to log the actual name of the constructor for user-defined constructor functions, you have to give the constructor function an actual name (e.g., **`var Person = function Person(){};`**).

Verify that an object is an instance of a particular constructor function

By using the **`instanceof`** operator, we can determine (true or false) if an object is an instance of a particular constructor function.

In the next sample, we are verifying if the object ***`InstanceOfCustomObject`*** is an instance of the ***`CustomConstructor`*** constructor function. This works with user-defined objects as well as native objects created with the **`new`** operator.

Sample: sample26.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // User-defined object constructor.
    var CustomConstructor = function () { this.foo = 'bar'; };

    // Instantiate an instance of CustomConstructor.
    var instanceOfCustomObject = new CustomConstructor();

    console.log(instanceOfCustomObject instanceof CustomConstructor); // Logs
true.

    // Works the same as a native object.
    console.log(new Array('foo') instanceof Array) // Logs true.

</script></body></html>
```

Notes
One thing to watch out for when dealing with the **`instanceof`** operator is that it will return ***`true`*** any time you ask if an object is an instance of **`Object`**, since all objects inherit from the **`Object()`** constructor.

The `instanceof` operator will return false when dealing with primitive values that leverage object wrappers (e.g., `'foo' instanceof String // returns false`). Had the string `'foo'` been created with the `new` operator, the `instanceof` operator would have returned true. So, keep in mind that `instanceof` really only works with complex objects and instances created from constructor functions that return objects.

An instance created from a constructor can have its own independent properties (aka instance properties)

In JavaScript, objects can be augmented at any time (i.e. dynamic properties). As previously mentioned, and to be exact, JavaScript has *mutable objects*. This means that objects created from a constructor function can be augmented with properties.

In the following code sample, I create an instance from the `Array()` constructor and then augment it with its own property.

Sample: sample27.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myArray = new Array();
    myArray.prop = 'test';

    console.log(myArray.prop) // Logs 'test'.

</script></body></html>
```

This could be done with `Object()`, `RegExp()`, or any of the other non-primitive constructors—even `Boolean()`.

Sample: sample28.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // This can be done with any of the native constructors that actually
produce an object.
    var myString = new String();
    var myNumber = new Number();
    var myBoolean = new Boolean(true);
    var myObject = new Object();
    var myArray = new Array();
    var myFunction = new Function('return 2+2');
    var myRegExp = new RegExp('\bt[a-z]+\b');

    myString.prop = 'test';
    myNumber.prop = 'test';
    myBoolean.prop = 'test';
```

```
    myObject.prop = 'test';
    myArray.prop = 'test';
    myFunction.prop = 'test';
    myRegExp.prop = 'test';

    // Logs 'test', 'test', 'test', 'test', 'test', 'test', 'test'.
    console.log(myString.prop, myNumber.prop, myBoolean.prop, myObject.prop,
myArray.prop, myFunction.prop, myRegExp.prop);

    // Be aware: Instance properties do not work with primitive/literal
values.
    var myString = 'string';
    var myNumber = 1;
    var myBoolean = true;

    myString.prop = true;
    myNumber.prop = true;
    myBoolean.prop = true;

    // Logs undefined, undefined, undefined.
    console.log(myString.prop, myNumber.prop, myBoolean.prop);

</script></body></html>
```

Adding properties to objects created from a constructor function sometimes occurs. Remember, object instances created from constructor functions are just plain old objects.

Notes
Keep in mind that besides their own properties, instances can have properties inherited from the prototype chain. Or, as we just saw in the previous code sample, properties added to the constructor after instantiation. This highlights the dynamic nature of objects in JavaScript.

The semantics of "JavaScript objects" and "`Object()` objects"

Do not confuse the general term "JavaScript objects," which refers to the notion of objects in JavaScript, with **`Object()`** objects. An **`Object()`** object (e.g., **`var myObject = new Object()`**) is a very specific type of value expressed in JavaScript. Just as an **`Array()`** object is a type of object called *array*, an **`Object()`** object is a type of object called *object*. The gist is that the **`Object()`** constructor function produces an empty generic object container, which is referred to as an **`Object()`** object. Similarly, the **`Array()`** constructor function produces an array object, and we refer to these objects as **`Array()`** objects.

In this book, the term "JavaScript objects" is used to refer to all objects in JavaScript, because most of the values in JavaScript can act like objects. This is due to the fact that the majority of JavaScript values are created from a native constructor function which produces a very specific type of object.

What you need to remember is that an `Object()` object is a very specific kind of value. It's a generic empty object. Do not confuse this with the term "JavaScript objects" used to refer to most of the values that can be expressed in JavaScript as an object.

Complex objects can contain most of the JavaScript values as properties

A complex object can hold any permitted JavaScript value. In the following code, I create an **`Object()`** object called ***`myObject`*** and then add properties representing the majority of values available in JavaScript.

Sample: sample29.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myObject = {};

    // Contain properties inside of myObject representing most of the native
JavaScript values.
    myObject.myFunction = function () { };
    myObject.myArray = [];
    myObject.myString = 'string';
    myObject.myNumber = 33;
    myObject.myDate = new Date();
    myObject.myRegExp = /a/;
    myObject.myNull = null;
    myObject.myUndefined = undefined;
    myObject.myObject = {};
    myObject.myMath_PI = Math.PI;
    myObject.myError = new Error('Darn!');

    console.log(myObject.myFunction, myObject.myArray, myObject.myString,
myObject.myNumber, myObject.myDate, myObject.myRegExp, myObject.myNull,
myObject.myNull, myObject.myUndefined, myObject.myObject, myObject.myMath_PI,
myObject.myError);

    /* Works the same with any of the complex objects, for example a
function. */
    var myFunction = function () { };

    myFunction.myFunction = function () { };
    myFunction.myArray = [];
    myFunction.myString = 'string';
    myFunction.myNumber = 33;
    myFunction.myDate = new Date();
    myFunction.myRegExp = /a/;
    myFunction.myNull = null;
    myFunction.myUndefined = undefined;
    myFunction.myObject = {};
```

```
    myFunction.myMath_PI = Math.PI;
    myFunction.myError = new Error('Darn!');

    console.log(myFunction.myFunction, myFunction.myArray,
myFunction.myString, myFunction.myNumber, myFunction.myDate,
myFunction.myRegExp, myFunction.myNull, myFunction.myNull,
myFunction.myUndefined, myFunction.myObject, myFunction.myMath_PI,
myFunction.myError);

</script></body></html>
```

The simple concept to learn here is that complex objects can contain—or refer to—anything you can nominally express in JavaScript. You should not be surprised when you see this done, as all of the native objects can be mutated. This even applies to **`String()`**, **`Number()`**, and **`Boolean()`** values in their object form—i.e. when they are created with the **`new`** operator.

Encapsulating complex objects in a programmatically beneficial way

The **`Object()`**, **`Array()`**, and **`Function()`** objects can contain other complex objects. In the following sample, I demonstrate this by setting up an object tree using **`Object()`** objects.

Sample: sample30.html

```
<!DOCTYPE html><html lang="en"><body><script>

// Encapsulation using objects creates object chains.
var object1 = {
      object1_1: {
            object1_1_1: {foo: 'bar'},
            object1_1_2: {},
      },
      object1_2: {
            object1_2_1: {},
            object1_2_2: {},
      }
};

console.log(object1.object1_1.object1_1_1.foo); // Logs 'bar'.

</script></body></html>
```

The same thing can be done with an **`Array()`** object (aka multidimensional array), or with a **`Function()`** object.

Sample: sample31.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // Encapsulation using arrays creates a multidimensional array chain.
    var myArray = [[[]]]; // An empty array, inside an empty array, inside an
empty array.

    /* Here is an example of encapsulation using functions: An empty function
inside an empty function inside an empty function. */
    var myFunction = function () {
        // Empty function.
        var myFunction = function () {
            // Empty function.
            var myFunction = function () {
                // Empty function.
            };
        };
    };

    // We can get crazy and mix and match too.
    var foo = [{ foo: [{ bar: { say: function () { return 'hi'; } }}]}];
    console.log(foo[0].foo[0].bar.say()); // Logs 'hi'.

</script></body></html>
```

The main concept to take away here is that some of the complex objects are designed to encapsulate other objects in a programmatically beneficial way.

Getting, setting, and updating an object's properties using dot notation or bracket notation

We can get, set, or update an object's properties using either *dot notation* or *bracket notation*.

In the following sample, I demonstrate dot notation, which is accomplished by using the object name followed by a period, and then followed by the property to get, set, or update (e.g., ***`objectName.property`***).

Sample: sample32.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // Create a cody Object() object.
    var cody = new Object();

    // Setting properties.
```

```
    cody.living = true;
    cody.age = 33;
    cody.gender = 'male';
    cody.getGender = function () { return cody.gender; };

    // Getting properties.
    console.log(
        cody.living,
        cody.age,
        cody.gender,
        cody.getGender()
        ); // Logs 'true 33 male male'.

    // Updating properties, exactly like setting.
    cody.living = false;
    cody.age = 99;
    cody.gender = 'female';
    cody.getGender = function () { return 'Gender = ' + cody.gender; };

    console.log(cody);

</script></body></html>
```

Dot notation is the most common notation for getting, setting, or updating an object's properties.

Bracket notation, unless required, is not as commonly used. In the following sample, I replace the dot notation used in the previous sample with bracket notation. The object name is followed by an opening bracket, the property name (in quotes), and then a closing bracket:

Sample: sample33.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // Creating a cody Object() object.
    var cody = new Object();

    // Setting properties.
    cody['living'] = true;
    cody['age'] = 33;
    cody['gender'] = 'male';
    cody['getGender'] = function () { return cody.gender; };

    // Getting properties.
    console.log(
        cody['living'],
        cody['age'],
        cody['gender'],
```

```
        cody['getGender']() // Just slap the function invocation on the end!
        ); // Logs 'true 33 male male'.

    // Updating properties, very similar to setting.
    cody['living'] = false;
    cody['age'] = 99;
    cody['gender'] = 'female';
    cody['getGender'] = function () { return 'Gender = ' + cody.gender; };

    console.log(cody);

</script></body></html>
```

Bracket notation can be very useful when you need to access a property key and what you have to work with is a variable that contains a string value representing the property name. In the next sample, I demonstrate the advantage of bracket notation over dot notation by using it to access the property ***foobar***. I do this using two variables that, when joined, produce the string version of the property key contained in ***foobarObject***.

Sample: sample34.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var foobarObject = { foobar: 'Foobar is code for no code' };

    var string1 = 'foo';
    var string2 = 'bar';

    console.log(foobarObject[string1 + string2]); // Let's see dot notation
do this!

</script></body></html>
```

Additionally, bracket notation can come in handy for getting at property names that are invalid JavaScript identifiers. In the following code, I use a number and a reserved keyword as a property name (valid as a string) that only bracket notation can access.

Sample: sample35.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myObject = { '123': 'zero', 'class': 'foo' };

    // Let's see dot notation do this! Keep in mind 'class' is a keyword in
JavaScript.
    console.log(myObject['123'], myObject['class']); //Logs 'zero foo'.
```

```
    // It can't do what bracket notation can do, in fact it causes an error.
    // console.log(myObject.0, myObject.class);

</script></body></html>
```

Notes

Because objects can contain other objects, `cody.object.object.object.object` or `cody['object']['object']['object']['object']` can be seen at times. This is called object chaining. The encapsulation of objects can go on indefinitely.

Objects are mutable in JavaScript, meaning that getting, setting, or updating them can be performed on most objects at any time. By using the bracket notation (e.g., `cody['age']`), you can mimic associative arrays found in other languages.

If a property inside an object is a method, all you have to do is use the `()` operators (e.g., `cody.getGender()`) to invoke the property method.

Deleting object properties

The `delete` operator can be used to completely remove properties from an object. In the following code snippet, we delete the *`bar`* property from the *`foo`* object.

Sample: sample36.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var foo = { bar: 'bar' };
    delete foo.bar;
    console.log('bar' in foo); // Logs false, because bar was deleted from
foo.

</script></body></html>
```

Notes

`delete` will not delete properties that are found on the prototype chain.

Deleting is the only way to actually remove a property from an object. Setting a property to `undefined` or `null` only changes the value of the property. It does not remove the property from the object.

How references to object properties are resolved

If you attempt to access a property that is not contained in an object, JavaScript will attempt to find the property or method using the prototype chain. In the following sample, I create an array and attempt to access a property called *`foo`* that has not yet

been defined. You might think that because ***myArray.foo*** is not a property of the ***myArray*** object, JavaScript will immediately return **undefined**. But JavaScript will look in two more places (**Array.prototype** and then **Object.prototype**) for the value of ***foo*** before it returns **undefined**.

Sample: sample37.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myArray = [];

    console.log(myArray.foo); // Logs undefined.

    /* JS will look at Array.prototype for Array.prototype.foo, but it is not
there. Then it will look for it at Object.prototype, but it is not there
either, so undefined is returned! */

</script></body></html>
```

When I attempt to access a property of an object, it will check that object instance for the property. If it has the property, it will return the value of the property, and there is no inheritance occurring because the prototype chain is not leveraged. If the instance does not have the property, JavaScript will then look for it on the object's constructor function **prototype** object.

All object instances have a property that is a secret link (aka __proto__) to the constructor function that created the instance. This secret link can be leveraged to grab the constructor function, specifically the *prototype property* of the instance's constructor function.

This is one of the most confusing aspects of objects in JavaScript. But let's reason this out. Remember that a function is also an object with properties. It makes sense to allow objects to *inherit* properties from other objects. Just like saying: "Hey *object B*, I would like you to share all the properties that *object A* has." JavaScript wires this all up for native objects by default via the **prototype** object. When you create your own constructor functions, you can leverage prototype chaining as well.

How exactly JavaScript accomplishes this is confusing until you see it for what it is: just a set of rules. Let's create an array to examine the **prototype** property closer.

Sample: sample38.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // myArray is an Array object.
    var myArray = ['foo', 'bar'];
```

```
    console.log(myArray.join()); // join() is actually defined at
Array.prototype.join

</script></body></html>
```

Our **`Array()`** instance is an object with properties and methods. As we access one of the array methods, such as **`join()`**, let's ask ourselves: Does the ***`myArray`*** instance created from the **`Array()`** constructor have its own **`join()`** method? Let's check.

Sample: sample39.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myArray = ['foo', 'bar'];

    console.log(myArray.hasOwnProperty('join')); // Logs false.

</script></body></html>
```

No it does not. Yet ***`myArray`*** has access to the **`join()`** method as if it were its own property. What happened here? Well, you just observed the prototype chain in action. We accessed a property that, although not contained in the ***`myArray`*** object, could be found by JavaScript somewhere else. That somewhere else is very specific. When the **`Array()`** constructor was created by JavaScript, the **`join()`** method was added (among others) as a property of the **`prototype`** property of **`Array()`**.

To reiterate, if you try to access a property on an object that does not contain it, JavaScript will search the **`prototype`** chain for this value. First it will look at the constructor function that created the object (e.g., **`Array`**), and inspect its prototype (e.g., **`Array.prototype`**) to see if the property can be found there. If the first prototype object does not have the property, then JavaScript keeps searching up the chain at the constructor behind the initial constructor. It can do this all the way up to the end of the chain.

Where does the chain end? Let's examine the example again, invoking the **`toLocaleString()`** method on ***`myArray`***.

Sample: sample40.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // myArray and Array.prototype contain no toLocaleString() method.
    var myArray = ['foo', 'bar'];

    // toLocaleString() is actually defined at
Object.prototype.toLocaleString
```

```
    console.log(myArray.toLocaleString()); // Logs 'foo,bar'.

</script></body></html>
```

The **`toLocaleString()`** method is not defined within the ***`myArray`*** object. So, the prototype chaining rule is invoked and JavaScript looks for the property in the **`Array`** constructor's prototype property (e.g., **`Array.prototype`**). It is not there either, so the chain rule is invoked again and we look for the property in the **`Object()`** prototype property (**`Object.prototype`**). And yes, it is found there. Had it not been found there, JavaScript would have produced an error stating that the property was **`undefined`**.

Since all prototype properties are objects, the final link in the chain is **`Object.prototype`**. There is no other constructor prototype property that can be examined.

There is an entire chapter ahead that breaks down the prototype chain into smaller parts, so if this was completely lost on you, read that chapter and then come back to this explanation to solidify your understanding. From this short read on the matter, I hope you understand that when a property is not found (and deemed **`undefined`**), JavaScript will have looked at several prototype objects to determine that a property is **`undefined`**. A lookup always occurs, and this lookup process is how JavaScript handles inheritance as well as simple property lookups.

Using `hasOwnProperty` to verify that an object property is not from the prototype chain

While the **`in`** operator can check for properties of an object, including properties from the prototype chain, the **`hasOwnProperty`** method can check an object for a property that is not from the prototype chain.

In the following sample, we want to know if ***`myObject`*** contains the property ***`foo`***, and that it is not inheriting the property from the prototype chain. To do this, we ask if ***`myObject`*** has its own property called ***`foo`***.

Sample: sample41.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myObject = {foo: 'value'};

    console.log(myObject.hasOwnProperty('foo')) // Logs true.

    // Versus a property from the prototype chain.
    console.log(myObject.hasOwnProperty('toString'));  // Logs false.

</script></body></html>
```

The **hasOwnProperty** method should be leveraged when you need to determine whether a property is local to an object or inherited from the prototype chain.

Checking if an object contains a given property using the in operator

The **in** operator is used to verify (true or false) if an object contains a given property. In this sample, we are checking to see if *foo* is a property in *myObject*.

Sample: sample42.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myObject = { foo: 'value' };
    console.log('foo' in myObject); // Logs true.

</script></body></html>
```

You should be aware that the **in** operator not only checks for properties contained in the object referenced, but also for any properties that object inherits via the **prototype** chain. Thus, the same property lookup rules apply and the property, if not in the current object, will be searched for on the **prototype** chain.

This means that *myObject* in the previous sample actually contains a **toString** property method via the **prototype** chain (**Object.prototype.toString**), even if we did not specify one (e.g., *myObject.toString = 'foo'*).

Sample: sample43.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myObject = { foo: 'value' };
    console.log('toString' in myObject); // Logs true.

</script></body></html>
```

In the last code example, the *toString* property is not literally inside of the *myObject* object. However, it is inherited from **Object.prototype**, and so the **in** operator concludes that *myObject* does in fact have an inherited **toString()** property method.

Enumerate (loop over) an object's properties using the for in loop

By using **for in**, we can loop over each property in an object. In the following sample, we are using the **for in** loop to retrieve the property names from the *cody* object.

Sample: sample44.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var cody = {
        age: 23,
        gender: 'male'
    };

    for (var key in cody) { // key is a variable used to represent each
property name.
        // Avoid properties inherited from the prototype chain.
        if (cody.hasOwnProperty(key)) {
            console.log(key);
        }
    }

</script></body></html>
```

Notes

The **for in** loop has a drawback. It will not only access the properties of the specific object being looped over. It will also include in the loop any properties inherited (via the prototype chain) by the object. Thus, if this is not the desired result, and most of the time it is not, we have to use a simple **if** statement inside of the loop to make sure we only access the properties contained within the specific object we are looping over. This can be done by using the **hasOwnProperty()** method inherited by all objects.

The order in which the properties are accessed in the loop is not always the order in which they are defined within the loop. Additionally, the order in which you defined properties is not necessarily the order they are accessed.

Only properties that are enumerable (i.e. available when looping over an object's properties) show up with the **for in** loop. For example, the constructor property will not show up. It is possible to check which properties are enumerable with the **propertyIsEnumerable()** method.

Host objects and native objects

You should be aware that the environment (e.g., a web browser) in which JavaScript is executed typically contains what are known as *host objects*. Host objects are not part of the ECMAScript implementation, but are available as objects during execution. Of course, the availability and behavior of a host object depends completely on what the host environment provides.

For example, in the web browser environment the window/head object and all of its containing objects (excluding what JavaScript provides) are considered host objects.

In the following example, I examine the properties of the **window** object.

Sample: sample45.html

```
<!DOCTYPE html><html lang="en"><body><script>

    for (x in window) {
        console.log(x); // Logs all of the properties of the window/head
object.
    }

</script></body></html>
```

You might have noticed that native JavaScript objects are not listed among the host objects. It's fairly common that a browser distinguishes between host objects and native objects.

As it pertains to web browsers, the most famous of all hosted objects is the interface for working with HTML documents, also known as the DOM. The following sample is a method to list all of the objects contained inside the ***window.document*** object provided by the browser environment.

Sample: sample46.html

```
<!DOCTYPE html><html lang="en"><body><script>

    for (x in window.document) {
        console.log();
    }

</script></body></html>
```

What I want you to learn here is that the JavaScript specification does not concern itself with host objects and vice versa. There is a dividing line between what JavaScript provides (e.g., JavaScript 1.5, ECMA-262, Edition 3 versus Mozilla's JavaScript 1.6, 1.7, 1.8, 1.8.1, 1.8.5) and what the host environment provides, and these two should not be confused.

Notes
The host environment (e.g., a web browser) that runs JavaScript code typically provides the *head object* (e.g., window object in a web browser) where the native portions of the language are stored along with host objects (e.g., ***window.location*** in a web browser) and user-defined objects (e.g., the code you write to run in the web browser).

Sometimes a web browser manufacturer, as the host of the JavaScript interpreter, will push forward a version of JavaScript or add future specifications to JavaScript before they have been approved (e.g., Mozilla's Firefox JavaScript 1.6, 1.7, 1.8, 1.8.1, 1.8.5).

Enhancing and extending objects with Underscore.js

JavaScript 1.5 is lacking when it comes time to seriously manipulate and manage objects. If you are running JavaScript in a web browser, I would like to be bold here and suggest the usage of Underscore.js when you need more functionality than is provided by JavaScript 1.5. Underscore.js provides the following functionality when dealing with objects.

These functions work on all objects and arrays:

- `each()`
- `map()`
- `reduce()`
- `reduceRight()`
- `detect()`
- `select()`
- `reject()`
- `all()`
- `any()`
- `include()`
- `invoke()`
- `pluck()`
- `max()`
- `min()`
- `sortBy()`
- `sortIndex()`

- `toArray()`
- `size()`

These functions work on all objects:

- `keys()`
- `values()`
- `functions()`
- `extend()`
- `clone()`
- `tap()`
- `isEqual()`
- `isEmpty()`
- `isElement()`
- `isArray()`
- `isArguments`
- `isFunction()`
- `isString()`
- `isNumber`
- `isBoolean`
- `isDate`
- `isRegExp`
- `isNaN`
- `isNull`
- `isUndefined`

I like this library because it takes advantage of the new native additions to JavaScript where browsers support them, but also provides the same functionality to browsers that do not, all without changing the native implementation of JavaScript unless it has to.

Notes

Before you start to use Underscore.js, make sure the functionality you need is not already provided by a JavaScript library or framework that might already be in use in your code.

Chapter 3 `String()`

Conceptual overview of using the `String()` object

The `String()` constructor function is used to create string objects and string primitive values.

In the following sample, I detail the creation of string values in JavaScript.

Sample: sample47.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // Create a string object using the new keyword and the String()
constructor.
    var stringObject = new String('foo');
    console.log(stringObject); // Logs foo {0 = 'f', 1 = 'o', 2 = 'o'}
    console.log(typeof stringObject); // Logs 'object'.

    // Create string literal/primitive by directly using the String
constructor.
    var stringObjectWithOutNewKeyword = String('foo'); // Without new
keyword.
    console.log(stringObjectWithOutNewKeyword); // Logs 'foo'.
    console.log(typeof stringObjectWithOutNewKeyword); // Logs 'string'.

    // Create string literal/primitive (constructor leveraged behind the
scene).
    var stringLiteral = 'foo';
    console.log(stringLiteral); // Logs foo.
    console.log(typeof stringLiteral); // Logs 'string'.

</script></body></html>
```

`String()` parameters

The `String()` constructor function takes one parameter: the string value being created. In the following sample, we create a variable, `stringObject`, to contain the string value "foo".

Sample: sample48.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // Create string object.
    var stringObject = new String('foo');
```

```
    console.log(stringObject); // Logs 'foo {0="f", 1="o", 2="o"}'

</script></body></html>
```

Notes
When used with the `new` keyword, instances from the `String()` constructor produce an actual complex object. You should avoid doing this (use literal/primitive numbers) due to the potential problems associated with the `typeof` operator. The `typeof` operator reports complex string objects as 'object' instead of the primitive label ('string') you might expect. Additionally, the literal/primitive value is just faster to write and is more concise.

`String()` properties and methods

The `String` object has the following properties and methods (not including inherited properties and methods):

Properties (e.g., `String.prototype;`)

- `prototype`

Methods (e.g., `String.fromCharChode();`)

- `fromCharCode()`

String object instance properties and methods

String object instances have the following properties and methods (not including inherited properties and methods):

Instance Properties (e.g., `var myString = 'foo'; myString.length;`)

- `constructor`
- `length`

Instance Methods (e.g., `var myString = 'foo'; myString.toLowerCase();`)

- `charAt()`
- `charCodeAt()`
- `concat()`

- indexOf()
- lastIndexOf()
- localeCompare()
- match()
- quote()
- replace()
- search()
- slice()
- split()
- substr()
- substring()
- toLocaleLowerCase()
- toLocaleUpperCase()
- toLowerCase()
- toString()
- toUpperCase()
- valueOf()

Chapter 4 `Number()`

Conceptual overview of using the `Number()` object

The `Number()` constructor function is used to create numeric objects and numeric primitive values.

In the following sample, I detail the creation of numeric values in JavaScript.

Sample: sample49.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // Create number object using the new keyword and the Number()
constructor.
    var numberObject = new Number(1);
    console.log(numberObject); // Logs 1.
    console.log(typeof numberObject) // Logs 'object'.

    // Create number literal/primitive using the number constructor without
new.
    var numberObjectWithOutNew = Number(1); // Without using new keyword.
    console.log(numberObjectWithOutNew); // Logs 1.
    console.log(typeof numberObjectWithOutNew) // Logs 'number'.

    // Create number literal/primitive (constructor leveraged behind the
scenes).
    var numberLiteral = 1;
    console.log(numberLiteral); // Logs 1.
    console.log(typeof numberLiteral); // Logs 'number'.

</script></body></html>
```

Integers and floating-point numbers

Numbers in JavaScript are typically written as either integer values or floating-point values. In the following code, I create a primitive integer number and a primitive floating-point number. This is the most common usage of number values in JavaScript.

Sample: sample50.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var integer = 1232134;
    console.log(integer); // Logs '1232134'.

    var floatingPoint = 2.132;
```

```
    console.log(floatingPoint); // Logs '2.132'.

</script></body></html>
```

Notes
A numeric value can be a hexadecimal value or octal value in JavaScript, but this is typically not done.

`Number()` parameters

The `Number()` constructor function takes one parameter: the numeric value being created. In the following snippet, we create a number *object* for the value ***456*** called `numberOne`.

Sample: sample51.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var numberOne = new Number(456);

    console.log(numberOne); // Logs '456{}'.

</script></body></html>
```

Notes
When used with the `new` keyword, instances from the `Number()` constructor produce a complex object. You should avoid creating number values using the `Number()` constructor (use literal/primitive numbers) due to the potential problems associated with the `typeof` operator. The `typeof` operator reports number objects as 'object' instead of the primitive label ('number') you might expect. The literal/primitive value is just more concise.

`Number()` properties

The `Number()` object has the following properties:

Properties (e.g., `Number.prototype;`)

- `MAX_VALUE`
- `MIN_VALUE`
- `NaN`

- NEGATIVE_INFINITY
- POSITIVE_INFINITY
- prototype

Number object instance properties and methods

Number object instances have the following properties and methods (not including inherited properties and methods):

Instance Properties (e.g., `var myNumber = 5; myNumber.constructor;`)

- constructor

Instance Methods (e.g., `var myNumber = 1.00324; myNumber.toFixed();`)

- toExponential()
- toFixed()
- toLocaleString()
- toPrecision()
- toString()
- valueOf()

Chapter 5 `Boolean()`

Conceptual overview of using the `Boolean()` object

The **`Boolean()`** constructor function can be used to create Boolean objects, as well as Boolean primitive values, that represent either a **`true`** or a **`false`** value.

In the following code, I detail the creation of Boolean values in JavaScript.

Sample: sample52.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // Create a Boolean object using the new keyword and the Boolean()
constructor.
    var myBoolean1 = new Boolean(false); // Using new keyword.
    console.log(typeof myBoolean1); // Logs 'object'.

    // Create a Boolean literal/primitive by directly using the number
constructor without new.
    var myBoolean2 = Boolean(0); // Without new keyword.
    console.log(typeof myBoolean2); // Logs 'boolean'.

    // Create Boolean literal/primitive (constructor leveraged behind the
scenes).
    var myBoolean3 = false;
    console.log(typeof myBoolean3); // Logs 'boolean'.
    console.log(myBoolean1, myBoolean2, myBoolean3); // Logs false false
false.

</script></body></html>
```

`Boolean()` parameters

The **`Boolean()`** constructor function takes one parameter to be converted to a Boolean value (i.e. **`true`** or **`false`**). Any valid JavaScript value that is not **`0`**, **`-0`**, **`null`**, **`false`**, **`NaN`**, **`undefined`**, or an empty string (**`""`**) will be converted to **`true`**. In the following sample, we create two Boolean object values: One **`true`** and one **`false`**.

Sample: sample53.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // Parameter passed to Boolean() = 0 = false, thus foo = false
    var foo = new Boolean(0)
    console.log(foo);
```

```
    // Parameter passed to Boolean() = Math = true, thus bar = true
    var bar = new Boolean(Math)
    console.log(bar);

</script></body></html>
```

Notes
When used with the `new` keyword, instances from the `Boolean()` constructor produce an actual complex object. You should avoid creating Boolean values using the `Boolean()` constructor (instead, use literal/primitive numbers) due to the potential problems associated with the `typeof` operator. The `typeof` operator reports Boolean objects as 'object', instead of the primitive label ('boolean') you might expect. Additionally, the literal/primitive value is faster to write.

`Boolean()` properties and methods

The `Boolean()` object has the following properties:

Properties (e.g., `Boolean.prototype;`):

- `prototype`

Boolean object instance properties and methods

Boolean object instances have the following properties and methods (not including inherited properties and methods):

Instance Properties (e.g., `var myBoolean = false; myBoolean.constructor;`):

- `constructor`

Instance Methods (e.g., `var myNumber = false; myBoolean.toString();`):

- `toSource()`
- `toString()`
- `valueOf()`

Non-primitive false Boolean objects convert to true

A `false` Boolean object (as opposed to a primitive value) created from the `Boolean()` constructor is an object, and objects convert to `true`. Thus, when creating a `false`

Boolean object via the **`Boolean()`** constructor, the value itself converts to **`true`**. In the following sample, I demonstrate how a **`false`** Boolean object is always "truthy."

Sample: sample54.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var falseValue = new Boolean(false);

    console.log(falseValue); // We have a false Boolean object, but objects
are truthy.

    if (falseValue) { // Boolean objects, even false Boolean objects, are
truthy.
        console.log('falseValue is truthy');
    }

</script></body></html>
```

If you need to convert a non-Boolean value into a Boolean, just use the **`Boolean()`** constructor without the **`new`** keyword and the value returned will be a primitive value instead of a Boolean object.

Certain things are false, everything else is true

It has already been mentioned, but is worth mentioning again because it pertains to conversions: If a value is **`0`**, **`-0`**, **`null`**, **`false`**, **`NaN`**, **`undefined`**, or an empty string(**`""`**), it is **`false`**. Any value in JavaScript except the aforementioned values will be converted to **`true`** if used in a Boolean context (i.e. **`if (true) {};`**).

Sample: sample55.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // All of these return a false Boolean value.
    console.log(Boolean(0));
    console.log(Boolean(-0));
    console.log(Boolean(null));
    console.log(Boolean(false));
    console.log(Boolean(''));
    console.log(Boolean(undefined));
    console.log(Boolean(null));

    // All of these return a true Boolean value.
    console.log(Boolean(1789));
    console.log(Boolean('false')); // 'false' as a string is not false the
Boolean value.
```

```
    console.log(Boolean(Math));
    console.log(Boolean(Array()));

</script></body></html>
```

It's critical that you understand which JavaScript values are reduced to **`false`** so you are aware that all other values are considered **`true`**.

Chapter 6 Working with Primitive String, Number, and Boolean Values

Primitive/literal values are converted to objects when properties are accessed

Do not be mystified by the fact that string, number, and Boolean literals can be treated like an object with properties (e.g., `true.toString()`). When these primitive values are treated like objects by attempting to access their properties, JavaScript will create a wrapper object from the primitive's associated constructor, so that the properties and methods of the wrapper object can be accessed. Once the properties have been accessed, the wrapper object is discarded. This conversion allows us to write code that would make it appear as if a primitive value was, in fact, an object. Truth be told, when it is treated like an object in code, JavaScript will convert it to an object so property access will work, and then convert it back to a primitive value once a value is returned. The key thing to notice here is what is occurring, and that JavaScript is doing this for you behind the scenes.

String
Sample: sample56.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // String object treated like an object.
    var stringObject = new String('foo');
    console.log(stringObject.length); // Logs 3.
    console.log(stringObject['length']); // Logs 3.

    // String literal/primitive converted to an object when treated as an
object.
    var stringLiteral = 'foo';
    console.log(stringLiteral.length); // Logs 3.
    console.log(stringLiteral['length']); // Logs 3.
    console.log('bar'.length); // Logs 3.
    console.log('bar'['length']); // Logs 3.

</script></body></html>
```

Number
Sample: sample57.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // Number object treated like an object.
```

```
    var numberObject = new Number(1.10023);
    console.log(numberObject.toFixed()); // Logs 1.
    console.log(numberObject['toFixed']()); // Logs 1.

    // Number literal/primitive converted to an object when treated as an
object.
    var numberLiteral = 1.10023;
    console.log(numberLiteral.toFixed()); // Logs 1.
    console.log(numberLiteral['toFixed']()); // Logs 1.
    console.log((1234).toString()); // Logs '1234'.
    console.log(1234['toString']()); // Logs '1234'.

</script></body></html>
```

Boolean
Sample: sample58.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // Boolean object treated like an object.
    var booleanObject = new Boolean(0);
    console.log(booleanObject.toString()); // Logs 'false'.
    console.log(booleanObject['toString']()); // Logs 'false'.

    // Boolean literal/primitive converted to an object when treated as an
object.
    var booleanLiteral = false;
    console.log(booleanLiteral.toString()); // Logs 'false'.
    console.log(booleanLiteral['toString']()); // Logs 'false'.
    console.log((true).toString()); // Logs 'true'.
    console.log(true['toString']()); // Logs 'true'.

</script></body></html>
```

Notes
When accessing a property on a primitive number directly (not stored in a variable), you have to first evaluate the number before the value is treated as an object (e.g., `(1).toString();` or `1..toString();`). Why two dots? The first dot is considered a numeric decimal, not an operator for accessing object properties.

You should typically use primitive string, number, and Boolean values

The literal/primitive values that represent a string, number, or Boolean are faster to write and are more concise in the literal form.

You should use the literal value because of this. Additionally, the accuracy of the **typeof** operator depends on how you create the value (literal versus constructor invocation). If you create a string, number, or Boolean object, the **typeof** operator reports the type as an object. If you use literals, the **typeof** operator returns a string name of the actual value type (e.g., **typeof 'foo' // returns 'string'**).

I demonstrate this fact in the following code.

Sample: sample59.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // String, number, and Boolean objects.
    console.log(typeof new String('foo')); // Logs 'object'.
    console.log(typeof new Number(1)); // Logs 'object'.
    console.log(typeof new Boolean(true)); // Logs 'object'.

    // String, number, and Boolean literals/primitives.
    console.log(typeof 'foo'); // Logs 'string'.
    console.log(typeof 1); // Logs 'number'.
    console.log(typeof true); // Logs 'boolean'.

</script></body></html>
```

If your program depends upon the **typeof** operator to identify string, number, or Boolean values in terms of those primitive types, you should avoid the **String**, **Number**, and **Boolean** constructors.

Chapter 7 Null

Conceptual overview of using the `null` value

You can use `null` to explicitly indicate that an object property does not contain a value. Typically, if a property is set up to contain a value, but the value is not available for some reason, the value `null` should be used to indicate that the reference property has an empty value.

Sample: sample60.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // The property foo is waiting for a value, so we set its initial value
to null.
    var myObjectObject = { foo: null };

    console.log(myObjectObject.foo); // Logs 'null'.

</script></body></html>
```

Notes

Don't confuse `null` with `undefined`. `undefined` is used by JavaScript to tell you that something is missing. `null` is provided so you can determine when a value is expected but not available yet.

`typeof` returns `null` values as "object"

For a variable that has a value of `null`, the `typeof` operator returns "object". If you need to verify a `null` value, the ideal solution would be to see if the value you are after is equal to `null`. In the following sample, we use the `===` operator to specifically verify that we are dealing with a `null` value.

Sample: sample61.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myObject = null;

    console.log(typeof myObject); // Logs 'object', not exactly helpful.
    console.log(myObject === null); // Logs true, only for a real null value.

</script></body></html>
```

Notes

When verifying a `null` value, always use `===` because `==` does not distinguish between `null` and `undefined`.

Chapter 8 Undefined

Conceptual overview of the `undefined` value

The **`undefined`** value is used by JavaScript in two slightly different ways.

The first way it's used is to indicate that a declared variable (e.g., **`var foo`**) has no *assigned value*. The second way it's used is to indicate that an object property you're trying to access is not *defined* (i.e. it has not even been named), and is not found in the prototype chain.

In the following sample, I examine both usages of **`undefined`** by JavaScript.

Sample: sample62.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var initializedVariable; // Declare variable.

    console.log(initializedVariable); // Logs undefined.
    console.log(typeof initializedVariable); // Confirm that JavaScript
returns undefined.

    var foo = {};

    console.log(foo.bar); // Logs undefined, no bar property in foo object.
    console.log(typeof foo.bar); // Confirm that JavaScript returns
undefined.

</script></body></html>
```

Notes
It is considered good practice to allow JavaScript alone to use **`undefined`**. You should never find yourself setting a value to **`undefined`**, as in ***`foo`*** **`= undefined`**. Instead, **`null`** should be used if you are specifying that a property or variable value is not available.

JavaScript ECMA-262 Edition 3 (and later) declares the `undefined` variable in the global scope

Unlike previous versions, JavaScript ECMA-262 Edition 3 (and later) has a global variable called **`undefined`** declared in the global scope. Because the variable is declared and not assigned a value, the undefined variable is set to **`undefined`**.

Sample: sample63.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // Confirm that undefined is a property of the global scope.
    console.log(undefined in this); // Logs true.

</script></body></html>
```

Chapter 9 The Head/Global Object

Conceptual overview of the head object

JavaScript code itself must be contained within an object. For example, when crafting JavaScript code for a web browser environment, JavaScript is contained and executed within the **window** object. This **window** object is considered to be the "head object," or sometimes confusingly referred to as "the global object." All implementations of JavaScript require the use of a single head object.

The head object is set up by JavaScript behind the scenes to encapsulate user-defined code and to house the native code with which JavaScript comes prepackaged. User-defined code is placed by JavaScript inside the head object for execution. Let's verify this as it pertains to a web browser.

In the following sample, I am creating some JavaScript values and verifying the values are placed in the head **window** object.

Sample: sample64.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myStringVar = 'myString';
    var myFunctionVar = function () { };
    myString = 'myString';
    myFunction = function () { };

    console.log('myStringVar' in window); // Returns true.
    console.log('myFunctionVar' in window); // Returns true.
    console.log('myString' in window); // Returns true.
    console.log('myFunction' in window); // Return true.

</script></body></html>
```

You should always be aware that when you write JavaScript, it will be written in the context of the head object. The remaining material in this chapter assumes you are aware that the term "head object" is synonymous with "global object."

Notes
The head object is the highest scope/context available in a JavaScript environment.

Global functions contained within the head object

JavaScript ships with some predefined functions. The following native functions are considered methods of the head object (e.g., in a web browser, `window.parseInt('500')`). You can think of these as ready-to-use functions and methods (of the head object) provided by JavaScript.

- decodeURI()
- decodeURIComponent()
- encodeURI()
- encodeURIComponent()
- eval()
- isFinite()
- isNaN()
- parseFloat()
- parseInt()

The head object vs. global properties and global variables

Do not confuse the head object with global properties or global variables contained within the global scope. The head object is an object that contains all objects. The term "global properties" or "global variables" is used to refer to values directly contained inside the head object and are not specifically scoped to other objects. These values are considered global because no matter where code is currently executing, in terms of scope, all code has access (via the scope chain) to these global properties and variables.

In the following sample, I place a ***foo*** property in the global scope, then access this property from a different scope.

Sample: sample65.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var foo = 'bar'; // foo is a global object and a property of the
head/window object.

    var myApp = function () { // Remember functions create scope.
        var run = function () {
```

```
            // Logs bar, foo's value is found via the scope chain in the head
object.
            console.log(foo);
        } ();
    }

    myApp();

</script></body></html>
```

Had I placed the ***foo*** property outside of the global scope, the ***console.log*** function would return **undefined**. This is demonstrated in the next code example.

Sample: sample66.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myFunction = function () { var foo = 'bar' }; // foo is now in the
scope of myFunction()

    var myApp = function () {
        var run = function () {
            console.log(foo); // foo is undefined, no longer in the global
scope, an error occurs.
        } ();
    }

    myApp();

</script></body></html>
```

In the browser environment, this is why global property methods (e.g., **window.alert()**) can be invoked from any scope. What you need to take away from this is that anything in the global scope is available to any scope, and thus gets the title of "global variable" or "global property."

Notes
There is a slight difference between using **var** and not using **var** in the global scope (global properties vs. global variables). Have a look at this Stack Overflow exchange for the details: Difference between using var and not using var in JavaScript.

Referring to the head object

There are typically two ways to reference the head object. The first way is to simply reference the name given to the head object (e.g., in a web browser this would be

window). The second way is to use the **this** keyword in the global scope. Each of these is detailed in the following sample.

Sample: sample67.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var foo = 'bar';

    windowRef1 = window;
    windowRef2 = this;

    console.log(windowRef1, windowRef2); // Logs reference to window object.

    console.log(windowRef1.foo, windowRef2.foo); // Logs 'bar', 'bar'.

</script></body></html>
```

In this example, we explicitly store a reference to the head object in two variables that are then used to gain access to the global ***foo*** variable.

The head object is implied and typically not referenced explicitly

Typically a reference to the head object is not used because it is implied. For example, in the browser environment **window.alert** and **alert()** are essentially the same statement. JavaScript fills in the blanks here. Because the **window** object (i.e. the head object) is the last object checked in the scope chain for a value, the **window** object is essentially always implied. In the next example, we leverage the **alert()** function which is contained in the global scope.

Sample: sample68.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var foo = { // window is implied here, window.foo
        fooMethod: function () {
            alert('foo' + 'bar'); // window is implied here, window.alert
            window.alert('foo' + 'bar'); // window is explicitly used, with
the same effect.
        }
    }

    foo.fooMethod(); // window is implied here, window.foo.fooMethod()

</script></body></html>
```

Make sure you understand that the head object is implied even when you don't explicitly include it, because the head object is the last stop in the scope chain.

Notes

Being explicit (e.g., `window.alert()` vs. `alert()`) costs a little bit more with regard to performance (how fast the code runs). It's faster if you rely on the scope chain alone and avoid explicitly referencing the head object even if you know the property you want is contained in the global scope.

Chapter 10 `Object()`

Conceptual overview of using `Object()` objects

Using the built-in **`Object()`** constructor function, we can create generic empty objects on the fly. In fact, if you remember back to the beginning of Chapter 1, this is exactly what we did by creating the ***`cody`*** object. Let's recreate the ***`cody`*** object.

Sample: sample69.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var cody = new Object(); // Create an empty object with no properties.

    for (key in cody) { // Confirm that cody is an empty generic object.
        if (cody.hasOwnProperty(key)) {
            console.log(key); // Should not see any logs, because cody itself
has no properties.
        }
    }

</script></body></html>
```

Here, all we are doing is using the **`Object()`** constructor function to create a generic object called ***`cody`***. You can think of the **`Object()`** constructor as a cookie cutter for creating empty objects that have no predefined properties or methods (except, of course, those inherited from the prototype chain).

Notes

If it's not obvious, the **`Object()`** constructor is an object itself. That is, the constructor function is based on an object created from the **`Function`** constructor. This can be confusing. Just remember that like the **`Array`** constructor, the **`Object`** constructor simply spits out blank objects. And yes, you can create all the empty objects you like. However, creating an empty object like ***`cody`*** is very different than creating your own constructor function with predefined properties. Make sure you understand that ***`cody`*** is just an empty object based on the **`Object()`** constructor. To really harness the power of JavaScript, you will need to learn not only how to create empty object containers from **`Object()`**, but also how to build your own "class" of objects (e.g., ***`Person()`***) like the **`Object()`** constructor function itself.

`Object()` parameters

The `Object()` constructor function takes one optional parameter. That parameter is the value you would like to create. If you provide no parameter, then a `null` or `undefined` value will be assumed.

Sample: sample70.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // Create an empty object with no properties.
    var cody1 = new Object();
    var cody2 = new Object(undefined);
    var cody3 = new Object(null);

    console.log(typeof cody1, typeof cody2, typeof cody3); // Logs 'object
object object'.

</script></body></html>
```

If a value besides `null` or `undefined` is passed to the `Object` constructor, the value passed will be created as an object. So theoretically, we can use the `Object()` constructor to create any of the other native objects that have a constructor. In the next example, I do just that.

Sample: sample71.html

```
<!DOCTYPE html><html lang="en"><body><script>

    /* Use the Object() constructor to create string, number, array,
function, Boolean, and regex objects. */

    // The following logs confirm object creation.
    console.log(new Object('foo'));
    console.log(new Object(1));
    console.log(new Object([]));
    console.log(new Object(function () { }));
    console.log(new Object(true));
    console.log(new Object(/\bt[a-z]+\b/));

    /* Creating string, number, array, function, Boolean, and regex object
instances via the Object() constructor is really never done. I am just
demonstrating that it can be done. */

</script></body></html>
```

`Object()` properties and methods

The `Object()` object has the following properties (not including inherited properties and methods):

Properties (e.g., `Object.prototype;`):

- `prototype`

`Object()` object instance properties and methods

`Object()` object instances have the following properties and methods (does not include inherited properties and methods):

Instance Properties (e.g., `var myObject = {}; myObject.constructor;`):

- `constructor`

Instance Methods (e.g., `var myObject = {}; myObject.toString();`):

- `hasOwnProperty()`
- `isPrototypeOf()`
- `propertyIsEnumerable()`
- `toLocaleString()`
- `toString()`
- `valueOf()`

Notes
The prototype chain ends with `Object.prototype`, and thus all of the properties and methods of `Object()` are inherited by all JavaScript objects.

Creating `Object()` objects using "object literals"

Creating an "object literal" entails instantiating an object with or without properties using braces (e.g., `var cody = {};`). Remember at the beginning of Chapter 1 when we created the one-off ***cody*** object and then gave the ***cody*** object properties using dot notation? Let's do that again.

Sample: sample72.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var cody = new Object();
    cody.living = true;
    cody.age = 33;
    cody.gender = 'male';
    cody.getGender = function () { return cody.gender; };

    console.log(cody); // Logs cody object and properties.

</script></body></html>
```

Notice in the code that creating the ***cody*** object and its properties took five statements. Using the object literal notation we can express the same ***cody*** object in one statement.

Sample: sample73.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var cody = {
        living: true,
        age: 23,
        gender: 'male',
        getGender: function () { return cody.gender; }
    };
    // Notice the last property has no comma after it.

    console.log(cody); // Logs the cody object and its properties.

</script>
</body>
```

Using literal notation gives us the ability to create objects, including defined properties, with less code and visually encapsulate the related data. Notice the use of the `:` and `,` operators in a single statement. This is actually the preferred syntax for creating objects in JavaScript because of its terseness and readability.

You should be aware that property names can also be specified as strings:

Sample: sample74.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var cody = {
        'living': true,
```

```
        'age': 23,
        'gender': 'male',
        'getGender': function () { return cody.gender; }
    };

    console.log(cody); // Logs the cody object and its properties.

</script>
</body>
```

It's not necessary to specify properties as strings unless the property name:

- Is one of the reserved keywords (e.g., **`class`**).
- Contains spaces or special characters (anything other than numbers, letters, the dollar sign ($), or the underscore (_) character).
- Starts with a number.

Notes
Careful! The last property of an object should not have a trailing comma. This will cause an error in some JavaScript environments.

All objects inherit from `Object.prototype`

The **`Object()`** constructor function in JavaScript is special, as its **`prototype`** property is the last stop in the prototype chain.

In the following sample, I augment the **`Object.prototype`** with a ***`foo`*** property and then create a string and attempt to access the ***`foo`*** property as if it were a property of the string instance. Since the ***`myString`*** instance does not have a ***`foo`*** property, the prototype chain kicks in and the value is looked for at **`String.prototype`**. It is not there, so the next place to look is **`Object.prototype`**, which is the final location JavaScript will look for an object value. The ***`foo`*** value is found because I added it, thus it returns the value of ***`foo`***.

Sample: sample75.html

```
<!DOCTYPE html><html lang="en"><body><script>

    Object.prototype.foo = 'foo';

    var myString = 'bar';
```

```
    // Logs 'foo', being found at Object.prototype.foo via the prototype
chain.
    console.log(myString.foo);

</script>
</body>
```

Notes

Careful! Anything added to **`Object.prototype`** will show up in a **`for in`** loop and the prototype chain. Because of this, it's been said that changing **`Object.prototype`** is forbidden.

Chapter 11 `Function()`

Conceptual overview of using `Function()` objects

A function is a container of code statements that can be invoked using the parentheses `()` operator. Parameters can be passed inside of the parentheses during invocation so that the statements in the function can access certain values when the function is invoked.

In the following code, we create two versions of an ***`addNumbers`*** function object—one using the **`new`** operator and another using the more common literal pattern. Both are expecting two parameters. In each case, we invoke the function, passing parameters in the parentheses `()` operator.

Sample: sample76.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var addNumbersA = new Function('num1', 'num2', 'return num1 + num2');

    console.log(addNumbersA(2, 2)); // Logs 4.

    // Could also be written the literal way, which is much more common.
    var addNumbersB = function (num1, num2) { return num1 + num2; };

    console.log(addNumbersB(2, 2)); // Logs 4.

</script></body></html>
```

A function can be used to return a value, construct an object, or as a mechanism to simply run code. JavaScript has several uses for functions, but in its most basic form a function is simply a unique scope of executable statements.

`Function()` parameters

The **`Function()`** constructor takes an indefinite number of parameters, but the last parameter expected by the **`Function()`** constructor is a string containing statements that comprise the body of the function. Any parameters passed to the constructor before the last will be available to the function being created. It's also possible to send multiple parameters as a comma-separated string.

In the following code, I contrast the usage of the **`Function()`** constructor with the more common patterns of instantiating a function object.

Sample: sample77.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var addFunction = new Function('num1', 'num2', 'return num1 + num2');

    /* Alternately, a single comma-separated string with arguments can be
    the first parameter of the constructor, with the function body following.
*/
    var timesFunction = new Function('num1,num2', 'return num1 * num2');

    console.log(addFunction(2, 2), timesFunction(2, 2)); // Logs '4 4'

    // Versus the more common patterns for instantiating a function:
    var addFunction = function (num1, num2) { return num1 + num2; }; //
Expression form.
    function addFunction(num1, num2) { return num1 + num2; } // Statement
form.

</script></body></html>
```

Notes

Directly leveraging the `Function()` constructor is not recommended or typically ever done because JavaScript will use `eval()` to parse the string containing the function's logic. Many consider `eval()` to be unnecessary overhead. If it's in use, a flaw in the design of the code is highly possible.

Using the `Function()` constructor without the `new` keyword has the same effect as using only the constructor to create function objects (e.g., `new Function('x','return x')` vs. `function('x','return x')`).

No closure is created (see Chapter 7) when invoking the `Function()` constructor directly.

`Function()` properties and methods

The function object has the following properties (not including inherited properties and methods):

Properties (e.g., `Function.prototype;`):

- `prototype`

Function object instance properties and methods

Function object instances have the following properties and methods (not including inherited properties and methods):

Instance Properties (e.g., `var myFunction = function(x, y, z) {}; myFunction.length;`):

- `arguments`
- `constructor`
- `length`

Instance Methods (e.g., `var myFunction = function(x, y, z) {}; myFunction.toString();`):

- `apply()`
- `call()`
- `toString()`

Functions always return a value

While it's possible to create a function simply to execute code statements, it's also very common for a function to return a value. In the following sample, we are returning a string from the *`sayHi`* function.

Sample: sample78.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var sayHi = function () {
        return 'Hi';
    };

    console.log(sayHi()); // Logs "Hi".

</script></body></html>
```

If a function does not specify a return value, **`undefined`** is returned. In the following sample, we call the *`yelp`* function which logs the string *`'yelp'`* to the console without explicitly returning a value.

Sample: sample79.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var yelp = function () {
        console.log('I am yelping!');
        // Functions return undefined even if we don't.
    }

    /* Logs true because a value is always returned, even if we don't
specifically return one. */
    console.log(yelp() === undefined);

</script></body></html>
```

The concept to take away here is that all functions return a value, even if you do not explicitly provide a value to return. If you do not specify a value to return, the value returned is **`undefined`**.

Functions are first-class citizens (not just syntax, but values)

In JavaScript, functions are objects. This means that a function can be stored in a variable, array, or object. Also, a function can be passed to and returned from a function. A function has properties because it is an object. All of these factors make functions first-class citizens in JavaScript.

Sample: sample80.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // Functions can be stored in variables (funcA), arrays (funcB), and
objects (funcC).
    var funcA = function () { }; // Called like so: funcA()
    var funcB = [function () { } ]; // Called like so: funcB[0]()
    var funcC = { method: function () { } }; // too.method() or
funcC['method']()

    // Functions can be sent to and sent back from functions.
    var funcD = function (func) {
        return func
    };

    var runFuncPassedToFuncD = funcD(function () { console.log('Hi'); });

    runFuncPassedToFuncD();

    // Functions are objects, which means they can have properties.
    var funcE = function () { };
```

```
    funcE.answer = 'yup'; // Instance property.
    console.log(funcE.answer); // Logs 'yup'.

</script></body></html>
```

It is crucial that you realize a function is an object, and thus a value. It can be passed around or augmented like any other expression in JavaScript.

Passing parameters to a function

Parameters are vehicles for passing values into the scope of a function when it is invoked. In the following sample we invoke **addFunction()**. Since we have predefined it to take two parameters, two added values become available within its scope.

Sample: sample81.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var addFunction = function (number1, number2) {
        var sum = number1 + number2;
        return sum;
    }

    console.log(addFunction(3, 3)); // Logs 6.

</script></body></html>
```

Notes
In contrast to some other programming languages, it is perfectly legal in JavaScript to omit parameters even if the function has been defined to accept these arguments. The missing parameters are simply given the value **undefined**. Of course, by leaving out values for the parameters, the function might not work properly.

If you pass a function unexpected parameters (those not defined when the function was created), no error will occur. And it's possible to access these parameters from the **arguments** object, which is available to all functions.

this and arguments values are available to all functions

Inside the scope and body of all functions, the **this** and **arguments** values are available.

The **arguments** object is an array-like object containing all of the parameters being passed to the function. In the following code, even though we forgo specifying

parameters when defining the function, we can rely on the **arguments** array passed to the function to access parameters if they are sent upon invocation.

Sample: sample82.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var add = function () {
        return arguments[0] + arguments[1];
    };

    console.log(add(4, 4)); // Returns 8.

</script></body></html>
```

The **this** keyword, passed to all functions, is a reference to the object that contains the function. As you might expect, functions contained within objects as properties (i.e. methods) can use **this** to gain a reference to the parent object. When a function is defined in the global scope, the value of **this** is the global object. Review the following code and make sure you understand what **this** is returning.

Sample: sample83.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myObject1 = {
        name: 'myObject1',
        myMethod: function () { console.log(this); }
    };

    myObject1.myMethod(); // Logs 'myObject1'.

    var myObject2 = function () { console.log(this); };

    myObject2(); // Logs window.

</script></body></html>
```

The arguments.callee property

The **arguments** object has a property called **callee**, which is a reference to the function currently executing. This property can be used to reference the function from within the scope of the function (e.g., **arguments.callee**)—a self-reference. In the following code, we use this property to gain a reference to the calling function.

Sample: sample84.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var foo = function foo() {
        console.log(arguments.callee); // Logs foo()
        // callee could be used to invoke recursively the foo function (e.g.,
arguments.callee())
    } ();

</script></body></html>
```

This can be useful when a function needs to be called recursively.

The function instance `length` property and `arguments.length`

The **`arguments`** object has a unique **`length`** property. While you might think this length property will give you the number of defined arguments, it actually gives the number of parameters sent to the function during invocation.

Sample: sample85.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myFunction = function (z, s, d) {
        return arguments.length;
    };

    console.log(myFunction()); // Logs 0 because no parameters were passed to
the function.

</script></body></html>
```

Using the **`length`** property of all **`Function()`** instances, we can actually grab the total number of parameters the function is expecting.

Sample: sample86.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myFunction = function (z, s, d, e, r, m, q) {
        return myFunction.length;
    };

    console.log(myFunction()); // Logs 7.

</script></body></html>
```

Notes
The **arguments.length** property was deprecated in JavaScript 1.4, but the number of arguments sent to a function can be accessed from the **length** property of the function object. Moving forward, you can get the length value by leveraging the **callee** property to first gain reference to the function being invoked (i.e. **arguments.callee.length**).

Redefining function parameters

A function's parameters can be redefined inside the function either directly, or by using the **arguments** array. Take a look at this code:

Sample: sample87.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var foo = false;
    var bar = false;

    var myFunction = function (foo, bar) {
        arguments[0] = true;
        bar = true;
        console.log(arguments[0], bar); // Logs true true.
    }

    myFunction();

</script></body></html>
```

Notice that I can redefine the value of the ***bar*** parameter using the **arguments** index or by directly reassigning a new value to the parameter.

Return a function before it is done (i.e. cancel function execution)

Functions can be cancelled at any time during invocation by using the **return** keyword with or without a value. In the following sample, we are canceling the ***add*** function if the parameters are undefined or not a number.

Sample: sample88.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var add = function (x, y) {
        // If the parameters are not numbers, return error.
        if (typeof x !== 'number' || typeof y !== 'number') { return 'pass in
numbers'; }
        return x + y;
```

```
    }
    console.log(add(3, 3)); // Logs 6.
    console.log(add('2', '2')); // Logs 'pass in numbers'.

</script></body></html>
```

The concept to take away here is that you can cancel a function's execution by using the **`return`** keyword at any point in the execution of the function.

Defining a function (statement, expression, or constructor)

A function can be defined in three different ways: a function constructor, a function statement, or a function expression. In the following example, I demonstrate each variation.

Sample: sample89.html

```
<!DOCTYPE html><html lang="en"><body><script>

    /* Function constructor: The last parameter is the function logic,
    everything before it is a parameter. */
    var addConstructor = new Function('x', 'y', 'return x + y');

    // Function statement.
    function addStatement(x, y) {
        return x + y;
    }

    // Function expression.
    var addExpression = function (x, y) {
        return x + y;
    };

    console.log(addConstructor(2, 2), addStatement(2, 2), addExpression(2,
2)); // Logs '4 4 4'.

</script></body></html>
```

Notes

Some have said that there is a fourth type of definition for functions, called the "named function expression." A named function expression is simply a function expression that also contains a name (e.g., **`var add = function add(x, y) {return x+y}`**).

Invoking a function (function, method, constructor, or `call()` and `apply()`)

Functions are invoked using four different scenarios or patterns.

- As a function
- As a method
- As a constructor
- Using `apply()` or `call()`

In the following sample, we examine each of these invocation patterns.

Sample: sample90.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // Function pattern.
    var myFunction = function () { return 'foo' };
    console.log(myFunction()); // Logs 'foo'.

    // Method pattern.
    var myObject = { myFunction: function () { return 'bar'; } }
    console.log(myObject.myFunction()); // Logs 'bar'.

    // Constructor pattern.
    var Cody = function () {
        this.living = true;
        this.age = 33;
        this.gender = 'male';
        this.getGender = function () { return this.gender; };
    }
    var cody = new Cody(); // Invoke via the Cody constructor.
    console.log(cody); // Logs the cody object and properties.

    // apply() and call() pattern.
    var greet = {
        runGreet: function () {
            console.log(this.name, arguments[0], arguments[1]);
        }
    }

    var cody = { name: 'cody' };
    var lisa = { name: 'lisa' };

    // Invoke the runGreet function as if it were inside of the cody object.
    greet.runGreet.call(cody, 'foo', 'bar'); // Logs 'cody foo bar'.
```

```
    // Invoke the runGreet function as if it were inside of the lisa object.
    greet.runGreet.apply(lisa, ['foo', 'bar']); // Logs 'lisa foo bar'.

    /* Notice the difference between call() and apply() in how parameters are
sent to the function being invoked. */

</script></body></html>
```

Make sure you are aware of all four of the invocation patterns, as code you will encounter may contain any of them.

Anonymous functions

An anonymous function is a function that is not given an identifier. Anonymous functions are mostly used for passing functions as a parameter to another function.

Sample: sample91.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // function(){console.log('hi');}; // Anonymous function, but no way to
invoke it.

    // Create a function that can invoke our anonymous function.
    var sayHi = function (f) {
        f(); // Invoke the anonymous function.
    }

    // Pass an anonymous function as a parameter.
    sayHi(function () { console.log('hi'); }); // Logs 'hi'.

</script></body></html>
```

Self-invoking function expression

A function expression (really any function except one created from the **`Function()`** constructor) can be immediately invoked after definition by using the parentheses operator. In the following sample, we create a ***`sayWord()`*** function expression and then immediately invoke the function. This is considered to be a self-invoking function.

Sample: sample92.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var sayWord = function () { console.log('Word 2 yo mo!'); } (); // Logs
'Word 2 yo mo!'
```

```
</script></body></html>
```

Self-invoking anonymous function statements

It's possible to create an anonymous function statement that is self-invoked. This is called a self-invoking anonymous function. In the following sample, we create several anonymous functions that are immediately invoked.

Sample: sample93.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // Most commonly used/seen in the wild.
    (function (msg) {
        console.log(msg);
    })('Hi');

    // Slightly different, but achieving the same thing:
    (function (msg) {
        console.log(msg)
    } ('Hi'));

    // The shortest possible solution.
    !function sayHi(msg) { console.log(msg); } ('Hi');

    // FYI, this does NOT work!
    // function sayHi() {console.log('hi');}();

</script></body></html>
```

Notes

According to the ECMAScript standard, the parentheses around the function (or anything that transforms the function into an expression) are required if the function is to be invoked immediately.

Functions can be nested

Functions can be nested inside of other functions indefinitely. In the following code sample, we encapsulate the ***goo*** function inside of the ***bar*** function, which is inside of the ***foo*** function.

Sample: sample94.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var foo = function () {
```

```
        var bar = function () {
            var goo = function () {
                console.log(this); // Logs reference to head window object.
            } ();
        } ();
    } ();

</script></body></html>
```

The simple concept here is that functions can be nested and there is no limit to how deep the nesting can go.

Notes
Remember, the value of **`this`** for nested functions will be the head object (e.g., the **`window`** object in a web browser) in JavaScript 1.5, ECMA-262, Edition 3.

Passing functions to functions and returning functions from functions

As previously mentioned, functions are first-class citizens in JavaScript. And since a function is a value, and a function can be passed any sort of value, a function can be passed to a function. Functions that take and/or return other functions are sometimes called "higher-order functions."

In the following code, we are passing an anonymous function to the ***`foo`*** function which we then immediately return from the ***`foo`*** function. It is this anonymous function that the variable ***`bar`*** points to, since ***`foo`*** accepts and then returns the anonymous function.

Sample: sample95.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // Functions can be sent to, and sent back from, functions.
    var foo = function (f) {
        return f;
    }

    var bar = foo(function () { console.log('Hi'); });

    bar(); // Logs 'Hi'.

</script></body></html>
```

So when ***`bar`*** is invoked, it invokes the anonymous function that was passed to the ***`foo()`*** function, which is then passed back from the ***`foo()`*** function and referenced from

the *bar* variable. All this is to showcase the fact that functions can be passed around just like any other value.

Invoking function statements before they are defined (aka function hoisting)

A function statement can be invoked during execution before its actual definition. This is a bit odd, but you should be aware of it so you can leverage it, or at least know what's going on when you encounter it. In the following sample, I invoke the *sayYo()* and *sum()* function statements before they are defined.

Sample: sample96.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // Example 1
    var speak = function () {
        sayYo(); // sayYo() has not been defined yet, but it can still be
invoked, logs 'yo'.
        function sayYo() { console.log('Yo'); }
    } (); // Invoke

    // Example 2
    console.log(sum(2, 2)); // Invoke sum(), which is not defined yet, but
can still be invoked.
    function sum(x, y) { return x + y; }

</script></body></html>
```

This happens because before the code runs, function statements are interpreted and added to the execution stack/context. Make sure you are aware of this as you use function statements.

Notes
Functions defined as function expressions are not hoisted. Only function statements are hoisted.

A function can call itself (aka recursion)

It's perfectly legitimate for a function to call itself. In fact, this is often used in well-known coding patterns. In the code that follows, we kick off the *countDownFrom* function, which then calls itself via the function name *countDownFrom*. Essentially, this creates a loop that counts down from 5 to 0.

Sample: sample97.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var countDownFrom = function countDownFrom(num) {
        console.log(num);
        num--; // Change the parameter value.
        if (num < 0) { return false; } // If num < 0 return function with no
recursion.
        // Could have also done arguments.callee(num) if it was an anonymous
function.
        countDownFrom(num);
    };

    countDownFrom(5); // Kick off the function, which logs separately 5, 4,
3, 2, 1, 0.

</script></body></html>
```

You should be aware that it's natural for a function to invoke itself (aka recursion) or to do so repetitively.

Chapter 12 The `this` Keyword

Conceptual overview of `this` and how it refers to objects

When a function is created, a keyword called **`this`** is created (behind the scenes), which links to the object in which the function operates. Said another way, **`this`** is available to the scope of its function, yet is a reference to the object of which that function is a property or method.

Let's take a look at the ***cody*** object from Chapter 1 again:

Sample: sample98.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var cody = {
        living: true,
        age: 23,
        gender: 'male',
        getGender: function () { return cody.gender; }
    };

    console.log(cody.getGender()); // Logs 'male'.

</script></body></html>
```

Notice how inside of the **`getGender`** function, we are accessing the ***gender*** property using dot notation (e.g., **`cody.gender`**) on the ***cody*** object itself. This can be rewritten using **`this`** to access the ***cody*** object because **`this`** points to the ***cody*** object.

Sample: sample99.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var cody = {
        living: true,
        age: 23,
        gender: 'male',
        getGender: function () { return this.gender; }
    };

    console.log(cody.getGender()); // Logs 'male'.

</script></body></html>
```

The **this** used in **this.gender** simply refers to the ***cody*** object on which the function is operating.

The topic of **this** can be confusing, but it does not have to be. Just remember that in general, **this** is used inside of functions to refer to the object the function is contained within, as opposed to the function itself (exceptions include using the **new** keyword or **call()** and **apply()**).

Notes
The keyword **this** looks and acts like any other variable, except you can't modify it.

As opposed to **arguments** and any parameters sent to the function, **this** is a keyword (not a property) in the call/activation object.

How is the value of this determined?

The value of **this**, passed to all functions, is based on the context in which the function is called at *run time*. Pay attention here, because this is one of those quirks you just need to memorize.

The ***myObject*** object in the following code sample is given a property called ***sayFoo***, which points to the ***sayFoo*** function. When the ***sayFoo*** function is called from the global scope, **this** refers to the **window** object. When it is called as a method of ***myObject***, **this** refers to ***myObject***.

Since ***myObject*** has a property named ***foo***, that property is used.

Sample: sample100.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var foo = 'foo';
    var myObject = { foo: 'I am myObject.foo' };

    var sayFoo = function () {
        console.log(this['foo']);
    };

    // Give myObject a sayFoo property and have it point to the sayFoo
function.
    myObject.sayFoo = sayFoo;

    myObject.sayFoo(); // Logs 'I am myObject.foo'.
    sayFoo(); // Logs 'foo'.

</script></body></html>
```

Clearly, the value of **this** is based on the context in which the function is being called. Consider that both **myObject.sayFoo** and **sayFoo** point to the same function. However, depending upon where (i.e. the context) **sayFoo()** is called from, the value of **this** is different.

If it helps, here is the same code with the head object (i.e. **window**) explicitly used.

Sample: sample101.html

```
<!DOCTYPE html><html lang="en"><body><script>

    window.foo = 'foo';
    window.myObject = { foo: 'I am myObject.foo' };

    window.sayFoo = function () {
        console.log(this.foo);
    };

    window.myObject.sayFoo = window.sayFoo;

    window.myObject.sayFoo();
    window.sayFoo();

</script></body></html>
```

Make sure that as you pass around functions, or have multiple references to a function, you realize that the value of **this** will change depending upon the context in which you call the function.

Notes
All variables except **this** and **arguments** follow lexical scope.

The this keyword refers to the head object in nested functions

You might be wondering what happens to **this** when it is used inside of a function that is contained inside of another function. The bad news is in ECMA 3, **this** loses its way and refers to the head object (the **window** object in browsers), instead of the object within which the function is defined.

In the following code, **this** inside of ***func2*** and ***func3*** loses its way and refers not to ***myObject*** but instead to the head object.

Sample: sample102.html

```
<!DOCTYPE html><html lang="en"><body><script>
```

```
    var myObject = {
        func1: function () {
            console.log(this); // Logs myObject.
            var func2 = function () {
                console.log(this) // Logs window, and will do so from this
point on.
                var func3 = function () {
                    console.log(this); // Logs window, as it's the head
object.
                } ();
            } ();
        }
    }

    myObject.func1();

</script></body></html>
```

The good news is that this will be fixed in ECMAScript 5. For now, you should be aware of this predicament, especially when you start passing functions around as values to other functions.

Consider the next sample and what happens when passing an anonymous function to ***foo.func1***. When the anonymous function is called inside of ***foo.func1*** (a function inside of a function), the **this** value inside of the anonymous function will be a reference to the head object.

Sample: sample103.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var foo = {
        func1: function (bar) {
            bar(); // Logs window, not foo.
            console.log(this); // The this keyword here will be a reference
to the foo object.
        }
    }

    foo.func1(function () { console.log(this) });

</script></body></html>
```

Now you will never forget: the **this** value will always be a reference to the head object when its host function is encapsulated inside of another function or invoked within the context of another function (again, this is fixed in ECMAScript 5).

Working around the nested function issue by leveraging the scope chain

So that the **this** value does not get lost, you can simply use the scope chain to keep a reference to **this** in the parent function. The following sample demonstrates how, using a variable called ***that***, and leveraging its scope, we can keep better track of function context.

Sample: sample104.html

```
<!DOCTYPE html><html lang="en"><body><script>

var myObject = {
      myProperty: 'I can see the light',
      myMethod : function(){
            var that = this; // Store a reference to this (i.e. myObject) in
myMethod scope.
            var helperFunction = function() { // Child function.
                  // Logs 'I can see the light' via scope chain because that
= this.
                  console.log(that.myProperty); // Logs 'I can see the
light'.
                  console.log(this); // Logs window object, if we don't use
"that".
            }();
      }
}

myObject.myMethod(); // Invoke myMethod.

</script></body></html>
```

Controlling the value of **this** using **call()** or **apply()**

The value of **this** is normally determined from the context in which a function is called (except when the **new** keyword is used—more about that in a minute), but you can overwrite and control the value of **this** using **apply()** or **call()** to define what object **this** points to when invoking a function. Using these methods is like saying: "Hey, call X function but tell the function to use Z object as the value for **this**." By doing so, the default way in which JavaScript determines the value of **this** is overridden.

In the next sample, we create an object and a function. We then invoke the function via **call()** so that the value of **this** inside the function uses ***myObject*** as its context. The statements inside the ***myFunction*** function will then populate ***myObject*** with properties instead of populating the head object. We have altered the object to which **this** (inside of ***myFunction***) refers.

Sample: sample105.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myObject = {};

    var myFunction = function (param1, param2) {
        // Set via call(), 'this' points to myObject when function is
invoked.
        this.foo = param1;
        this.bar = param2;
        console.log(this) // Logs Object {foo = 'foo', bar = 'bar'}
    };

    myFunction.call(myObject, 'foo', 'bar'); // Invoke function, set this
value to myObject.

    console.log(myObject) // Logs Object {foo = 'foo', bar = 'bar'}

</script></body></html>
```

In the previous example, we used **`call()`**, but **`apply()`** could be used as well. The difference between the two is how the parameters for the function are passed. Using **`call()`**, the parameters are just comma-separated values. Using **`apply()`**, the parameter values are passed inside of an array as shown in the following sample.

Sample: sample106.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myObject = {};

    var myFunction = function (param1, param2) {
        // Set via apply(), this points to myObject when function is invoked.
        this.foo = param1;
        this.bar = param2;
        console.log(this) // Logs Object {foo = 'foo', bar = 'bar'}
    };

    myFunction.apply(myObject, ['foo', 'bar']); // Invoke function, set this
value.

    console.log(myObject) // Logs Object {foo = 'foo', bar = 'bar'}

</script></body></html>
```

What you need to learn here is that you can override the default way in which JavaScript determines the value of **this** in a function's scope.

Using the this keyword inside a user-defined constructor function

When a function is invoked with the **new** keyword, the value of **this**—as it's stated in the constructor—refers to the instance itself. Said another way: In the constructor function, we can leverage the object via **this** *before the object is actually created*. In this case, the default value of **this** changes in a way similar to using **call()** or **apply()**.

In the following sample, we set up a ***Person*** constructor function that uses **this** to reference an object being created. When an instance of ***Person*** is created, ***this.name*** will reference the newly created object and place a property called ***name*** in the new object with a value from the parameter (***name***) passed to the constructor function.

Sample: sample107.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var Person = function (name) {
        this.name = name || 'john doe'; // this will refer to the instance
created.
    }

    var cody = new Person('Cody Lindley'); // Create an instance based on the
Person constructor.

    console.log(cody.name); // Logs 'Cody Lindley'.

</script></body></html>
```

Again, **this** refers to the "object that is to be" when the constructor function is invoked using the **new** keyword. Had we not used the **new** keyword, the value of **this** would be the context in which ***Person*** is invoked—in this case the head object. Let's examine the following scenario:

Sample: sample108.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var Person = function (name) {
        this.name = name || 'john doe';
    }

    var cody = Person('Cody Lindley'); // Notice we did not use 'new'.
```

```
    console.log(cody.name); // Undefined. The value is actually set at
window.name

    console.log(window.name); // Logs 'Cody Lindley'.

</script></body></html>
```

The keyword `this` inside a prototype method refers to a constructor instance

When used in functions added to a constructor's **`prototype`** property, **`this`** refers to the instance on which the method is invoked. Say we have a custom ***`Person()`*** constructor function. As a parameter, it requires the person's full name. In case we need to access the full name of the person, we add a ***`whatIsMyFullName`*** method to the ***`Person.prototype`*** so that all ***`Person`*** instances inherit the method. When using **`this`**, the method can refer to the instance invoking it (and thus its properties).

Here I demonstrate the creation of two ***`Person`*** objects (***`cody`*** and ***`lisa`***) and the inherited ***`whatIsMyFullName`*** method that contains the **`this`** keyword to access the instance.

Sample: sample109.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var Person = function (x) {
        if (x) { this.fullName = x };
    };

    Person.prototype.whatIsMyFullName = function () {
        return this.fullName; // 'this' refers to the instance created from
Person()
    }

    var cody = new Person('cody lindley');
    var lisa = new Person('lisa lindley');

    // Call the inherited whatIsMyFullName method, which uses this to refer
to the instance.
    console.log(cody.whatIsMyFullName(), lisa.whatIsMyFullName());

    /* The prototype chain is still in effect, so if the instance does not
have a fullName property, it will look for it in the prototype chain. Next,
we add a fullName property to both the Person prototype and the Object
prototype. See the notes that follow this sample. */

    Object.prototype.fullName = 'John Doe';
```

```
    var john = new Person(); // No argument is passed so fullName is not
added to the instance.
    console.log(john.whatIsMyFullName()); // Logs 'John Doe'.

</script></body></html>
```

The concept to take away here is that the keyword **`this`** is used to refer to instances when used inside of a method contained in the **`prototype`** object. If the instance does not contain the property, the prototype lookup begins.

Notes

If the instance or the object pointed to by **`this`** does not contain the property being referenced, the same rules that apply to any property lookup are applied, and the property will be "looked up" on the prototype chain. So in our example, if the ***`fullName`*** property was not contained within our instance, ***`fullName`*** would be looked for at ***`Person.prototype.fullName`***, then ***`Object.prototype.fullName`***.

Chapter 13 Scope and Closures

Conceptual overview of JavaScript scope

In JavaScript, scope is the context in which code is executed. There are three types of scope: global scope, local scope (sometimes referred to as "function scope"), and eval scope.

Code defined using **var** inside of a function is locally scoped, and is only "visible" to other expressions in that function, which includes code inside any nested/child functions. Variables defined in the global scope can be accessed from anywhere because it is the highest level and last stop in the scope chain.

Examine the code that follows and make sure you understand that each declaration of ***foo*** is unique because of scope.

Sample: sample110.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var foo = 0; // Global scope.
    console.log(foo); // Logs 0.

    var myFunction = function () {

        var foo = 1; // Local scope.

        console.log(foo); // Logs 1.

        var myNestedFunction = function () {

            var foo = 2; // Local scope.

            console.log(foo); // Logs 2.
        } ();
    } ();

    eval('var foo = 3; console.log(foo);'); // eval() scope.

</script></body></html>
```

Make sure you understand that each ***foo*** variable contains a different value because each one is defined in a specifically delineated scope.

Notes
An unlimited number of function and eval scopes can be created, while only one global scope is used by a JavaScript environment.

The global scope is the last stop in the scope chain.

Functions that contain functions create stacked execution scopes. These stacks, which are chained together, are often referred to as the scope chain.

JavaScript does not have block scope

Since logic statements (e.g., **`if`**) and looping statements (e.g., **`for`**) do not create a scope, variables can overwrite each other. Examine the following code and make sure you understand that the value of ***`foo`*** is being redefined as the program executes the code.

Sample: sample111.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var foo = 1; // foo = 1.

    if (true) {
        foo = 2; // foo = 2.
        for (var i = 3; i <= 5; i++) {
            foo = i; // foo = 3, 4, then 5.
            console.log(foo); // Logs 3, 4, 5.
        }
    }

</script></body></html>
```

So ***`foo`*** is changing as the code executes because JavaScript has no block scope—only function, global, or eval scope.

Use `var` inside of functions to declare variables and avoid scope gotchas

JavaScript will declare any variables lacking a **`var`** declaration (even those contained in a function or encapsulated functions) to be in the global scope instead of the intended local scope. Have a look at the code that follows and notice that without the use of **`var`** to declare ***`bar`***, the variable is actually defined in the global scope and not the local scope, where it should be.

Sample: sample112.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var foo = function () {
        var boo = function () {
            bar = 2; // No var used, so bar is placed in the global scope at
window.bar
        } ();
    } ();

    console.log(bar); // Logs 2, because bar is in the global scope.

    // As opposed to...

    var foo = function () {
        var boo = function () {
            var doo = 2;
        } ();
    } ();

    // console.log(doo); logs undefined. doo is in the boo function scope, so
an error occurs

</script></body></html>
```

The concept to take away here is that you should always use **var** when defining variables inside of a function. This will prevent you from dealing with potentially confusing scope problems. The exception to this convention, of course, is when you want to create or change properties in the global scope from within a function.

The scope chain (aka lexical scoping)

There is a lookup chain that is followed when JavaScript looks for the value associated with a variable. This chain is based on the hierarchy of scope. In the code that follows, I am logging the value of ***sayHiText*** from the ***func2*** function scope.

Sample: sample113.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var sayHiText = 'howdy';

    var func1 = function () {
        var func2 = function () {
            console.log(sayHiText); // func2 scope, but it finds sayHiText in
global scope.
        } ();
```

```
    } ();

</script></body></html>
```

How is the value of ***sayHiText*** found when it is not contained inside of the scope of the ***func2*** function? JavaScript first looks in the ***func2*** function for a variable named ***sayHiText***. Not finding ***func2*** there, it looks up to ***func2***'s parent function, ***func1***. The ***sayHiText*** variable is not found in the ***func1*** scope, either, so JavaScript then continues up to the global scope where ***sayHiText*** is found, at which point the value of ***sayHiText*** is delivered. If ***sayHiText*** had not been defined in the global scope, **undefined** would have been returned by JavaScript.

This is a very important concept to understand. Let's examine another code example, one in which we grab three values from three different scopes.

Sample: sample114.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var x = 10;
    var foo = function () {
        var y = 20;
        var bar = function () {
            var z = 30;
            console.log(z + y + x); // z is local, y and z are found in the
scope chain.
        } ();
    } ()

    foo(); // Logs 60.

</script></body></html>
```

The value for ***z*** is local to the ***bar*** function and the context in which the ***console.log*** is invoked. The value for ***y*** is in the ***foo*** function, which is the parent of ***bar()***, and the value for ***x*** is in the global scope. All of these are accessible to the ***bar*** function via the scope chain. Make sure you understand that referencing variables in the ***bar*** function will check all the way up the scope chain for the variables referenced.

Notes

The scope chain, if you think about it, is not that different from the prototype chain. Both are simply a way for a value to be looked up by checking a systematic and hierarchical set of locations.

The scope chain lookup returns the first found value

In the code sample that follows, a variable called ***x*** exists in the same scope in which it is examined with ***console.log***. This "local" value of ***x*** is used, and one might say that it shadows, or masks, the identically named x variables found further up in the scope chain.

Sample: sample115.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var x = false;
    var foo = function () {
        var x = false;
        bar = function () {
            var x = true;
            console.log(x); // Local x is first in the scope so it shadows
the rest.
        } ();
    }

    foo(); // Logs true.

</script></body></html>
```

Remember that the scope lookup ends when the variable is found in the nearest available link of the chain, even if the same variable name is used further up the chain.

Scope is determined during function definition, not invocation

Since functions determine scope and functions can be passed around just like any JavaScript value, one might think that deciphering the scope chain is complicated. It is actually very simple. The scope chain is decided based on the location of a function during *definition*, not during invocation. This is also called *lexical scoping*. Think long and hard about this, as most people stumble over it often in JavaScript code.

The scope chain is created before you invoke a function. Because of this, we can create closures. For example, we can have a function return a nested function to the global scope, yet our function can still access, via the scope chain, its parent function's scope. In the following sample, we define a ***parentFunction*** that returns an anonymous function, and we call the returned function from the global scope. Because our anonymous function was defined as being contained inside of ***parentFunction***, it still has access to ***parentFunction***'s scope when it is invoked. This is called a closure.

Sample: sample116.html

```
<!DOCTYPE html><html lang="en"><body><script>
```

```
    var parentFunction = function () {
        var foo = 'foo';
        return function () { // Anonymous function being returned.
            console.log(foo); // Logs 'foo'.
        }
    }

    // nestedFunction refers to the nested function returned from
parentFunction.
    var nestedFunction = parentFunction();

    nestedFunction(); // Logs foo because the returned function accesses foo
via the scope chain.

</script></body></html>
```

The idea you should take away here is that the scope chain is determined during definition—literally in the way the code is written. Passing around functions inside of your code will not change the scope chain.

Closures are caused by the scope chain

Take what you have learned about the scope chain and scope lookup in this chapter, and a closure should not be overly complicated to understand. In the following sample, we create a function called ***countUpFromZero***. This function actually returns a reference to the child function contained within it. When this child function (nested function) is invoked, it still has access to the parent function's scope because of the scope chain.

Sample: sample117.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var countUpFromZero = function () {
        var count = 0;
        return function () { // Return nested child function when
countUpFromZero is invoked.
            return ++count; // count is defined up the scope chain, in parent
function.
        };
    } (); // Invoke immediately, return nested function.

    console.log(countUpFromZero()); // Logs 1.
    console.log(countUpFromZero()); // Logs 2.
    console.log(countUpFromZero()); // Logs 3.

</script></body></html>
```

Each time the `countUpFromZero` function is invoked, the anonymous function contained in (and returned from) the `countUpFromZero` function still has access to the parent function's scope. This technique, facilitated via the scope chain, is an example of a closure.

Notes
If you feel I have over-simplified closures, you are likely correct in this thought. But I did so purposely as I believe the important parts come from a solid understanding of functions and scope, not necessarily the complexities of execution context. If you are in need of an in-depth dive into closures, have a look at JavaScript Closures.

Chapter 14 Function Prototype Property

Conceptual overview of the `prototype` chain

The **`prototype`** property is an object created by JavaScript for every **`Function()`** instance. Specifically, it links object instances created with the **`new`** keyword back to the constructor function that created them. This is done so that instances can share, or inherit, common methods and properties. Importantly, the sharing occurs during property lookup. Remember from Chapter 1 that every time you look up or access a property on an object, the property will be searched for on the object as well as the prototype chain.

Notes

A prototype object is created for every function, regardless of whether you intend to use that function as a constructor.

In the following code, I construct an array from the **`Array()`** constructor, and then I invoke the **`join()`** method.

Sample: sample118.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myArray = new Array('foo', 'bar');

    console.log(myArray.join()); // Logs 'foo,bar'.

</script></body></html>
```

The **`join()`** method is not defined as a property of the ***`myArray`*** object instance, but somehow we have access to **`join()`** as if it were. This method is defined somewhere, but where? Well, it is defined as a property of the **`Array()`** constructor's **`prototype`** property. Since **`join()`** is not found within the array object instance, JavaScript looks up the prototype chain for a method called **`join()`**.

Okay, so why are things done this way? Really, it is about efficiency and reuse. Why should every array instance created from the array constructor function have a uniquely defined **`join()`** method when **`join()`** always functions the same way? It makes more sense for all arrays to leverage the same **`join()`** function without having to create a new instance of the function for each array instance.

This efficiency we speak of is all possible because of the **`prototype`** property, prototype linkage, and the prototype lookup chain. In this chapter, we break down these often confusing attributes of prototypal inheritance. But truth be told, you would be better off

by simply memorizing the mechanics of how the chain hierarchy actually works. Refer back to Chapter 1 if you need a refresher on how property values are resolved.

Why care about the `prototype` property?

You should care about the `prototype` property for four reasons.

Reason 1
The first reason is that the prototype property is used by the native constructor functions (e.g., `Object()`, `Array()`, `Function()`, etc.) to allow constructor instances to inherit properties and methods. It is the mechanism that JavaScript itself uses to allow object instances to inherit properties and methods from the constructor function's `prototype` property. If you want to understand JavaScript better, you need to understand how JavaScript itself leverages the `prototype` object.

Reason 2
When creating user-defined constructor functions, you can orchestrate inheritance the same way JavaScript native objects do. But first you have to learn how it works.

Reason 3
You might really dislike prototypal inheritance or prefer another pattern for object inheritance, but the reality is that someday you might have to edit or manage someone else's code who thought prototypal inheritance was the bee's knees. When this happens, you should be aware of how prototypal inheritance works, as well as how it can be replicated by developers who make use of custom constructor functions.

Reason 4
By using prototypal inheritance, you can create efficient object instances that all leverage the same methods. As already mentioned, not all array objects, which are instances of the `Array()` constructor, need their own `join()` methods. All instances can leverage the same `join()` method because the method is stored in the prototype chain.

`Prototype` is standard on all `Function()` instances

All functions are created from a `Function()` constructor, even if you do not directly invoke the `Function()` constructor (e.g., `var add = new Function('x', 'y', 'return x + z');`) and instead use the literal notation (e.g., `var add = function(x,y){return x + z};`).

When a function instance is created, it is always given a `prototype` property, which is an empty object. In the following sample, we define a function called `myFunction` and then access the `prototype` property which is simply an empty object.

Sample: sample119.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myFunction = function () { };
    console.log(myFunction.prototype); // Logs object{}
    console.log(typeof myFunction.prototype); // Logs 'object'.

</script></body></html>
```

Make sure you completely understand that the prototype property is coming from the `Function()` constructor. It is only once we intend to use our function as a user-defined constructor function that the prototype property is leveraged, but this does not change the fact that the `Function()` constructor gives each instance a prototype property.

The default `prototype` property is an `Object()` object

All this `prototype` talk can get a bit heavy. Truly, `prototype` is just an empty object property called "prototype" created behind the scenes by JavaScript and made available by invoking the `Function()` constructor. If you were to do it manually, it would look something like this:

Sample: sample120.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myFunction = function () { };

    myFunction.prototype = {}; // Add the prototype property and set it to an
empty object.

    console.log(myFunction.prototype); // Logs an empty object.

</script></body></html>
```

In fact, this sample code actually works just fine, essentially just duplicating what JavaScript already does.

Notes
The value of a prototype property can be set to any of the complex values (i.e. objects) available in JavaScript. JavaScript will ignore any prototype property set to a primitive value.

Instances created from a constructor function are linked to the constructor's `prototype` property

While it's only an object, **`prototype`** is special because the prototype chain links every instance to its constructor function's prototype property. This means that any time an object is created from a constructor function using the **`new`** keyword (or when an object wrapper is created for a primitive value), it adds a hidden link between the object instance created and the prototype property of the constructor function used to create it. This link is known inside the instance as __proto__ (though it is only *exposed/supported* via code in Firefox 2+, Safari, Chrome, and Android). JavaScript wires this together in the background when a constructor function is invoked, and it's this link that allows the prototype chain to be, well, a chain. In the following sample, we add a property to the native **`Array()`** constructor's **`prototype`**, which we can then access from an **`Array()`** instance using the **`__proto__`** property set on that instance.

Sample: sample121.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // This code only works in browsers that support __proto__ access.
    Array.prototype.foo = 'foo';
    var myArray = new Array();

    console.log(myArray.__proto__.foo); // Logs foo, because
myArray.__proto__ = Array.prototype

</script></body></html>
```

Since accessing **`__proto__`** is not part of the official ECMA standard, there is a more universal way to trace the link from an object to the prototype object it inherits, and that is by using the **`constructor`** property. This is demonstrated in the following sample.

Sample: sample122.html

```
<!DOCTYPE html><html lang="en"><body><script>

    Array.prototype.foo = 'foo'; // All instances of Array() now inherit a
foo property.
    var myArray = new Array();

    // Trace foo in a verbose way leveraging *.constructor.prototype
    console.log(myArray.constructor.prototype.foo); // Logs foo.

    // Or, of course, leverage the chain.
    console.log(myArray.foo) // Logs foo.
    // Uses prototype chain to find property at Array.prototype.foo
```

```
</script></body></html>
```

In this example, the **foo** property is found within the prototype object. You need to realize this is only possible because of the association between the instance of **Array()** and the **Array()** constructor prototype object (i.e. **Array.prototype**). Simply put, **myArray.__proto__** (or **myArray.constructor.prototype**) references **Array.prototype**.

Last stop in the prototype chain is Object.prototype

Since the prototype property is an object, the last stop in the prototype chain or lookup is at **Object.prototype**. In the code that follows, I create ***myArray***, which is an empty array. I then attempt to access a property of ***myArray*** that has not yet been defined, engaging the prototype lookup chain. The ***myArray*** object is examined for the ***foo*** property. Being absent, the property is looked for at **Array.prototype**, but it is not there either. So the final place JavaScript looks is **Object.prototype**. Because it is not defined in any of those three objects, the property is **undefined**.

Sample: sample123.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myArray = [];

    console.log(myArray.foo) // Logs undefined.

    /* foo was not found at myArray.foo or Array.prototype.foo or
Object.prototype.foo, so it is undefined. */

</script></body></html>
```

Take note that the chain stopped with **Object.prototype**. The last place we looked for ***foo*** was **Object.prototype**.

Notes
Careful! Anything added to **Object.prototype** will show up in a **for in** loop.

The prototype chain returns the first property match it finds in the chain

Like the scope chain, the **prototype** chain will use the first value it finds during the chain lookup.

Modifying the previous code example, if we added the same value to the **`Object.prototype`** and **`Array.prototype`** objects, and then attempted to access a value on an array instance, the value returned would be from the **`Array.prototype`** object.

Sample: sample124.html

```
<!DOCTYPE html><html lang="en"><body><script>

    Object.prototype.foo = 'object-foo';
    Array.prototype.foo = 'array-foo';
    var myArray = [];

    console.log(myArray.foo); // Logs 'array-foo', which was found at
Array.prototype.foo

    myArray.foo = 'bar';

    console.log(myArray.foo) // Logs 'bar', was found at Array.foo

</script></body></html>
```

In this sample, the ***`foo`*** value at ***`Array.prototype.foo`*** is shadowing, or masking, the ***`foo`*** value found at ***`Object.prototype.foo`***. Just remember that the lookup ends when the property is found in the chain, even if the same property name is also used farther up the chain.

Replacing the `prototype` property with a new object removes the default constructor property

It's possible to replace the default value of a **`prototype`** property with a new value. However, doing so will eliminate the default *constructor* property found in the "pre-made" **`prototype`** object—unless you manually specify one.

In the code that follows, we create a ***`Foo`*** constructor function, replace the **`prototype`** property with a new empty object, and verify that the constructor property is broken (it now references the less useful **`Object`** prototype).

Sample: sample125.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var Foo = function Foo() { };

    Foo.prototype = {}; // Replace prototype property with an empty object.

    var FooInstance = new Foo();
```

```
    console.log(FooInstance.constructor === Foo); // Logs false, we broke the
reference.
    console.log(FooInstance.constructor); // Logs Object(), not Foo()

    // Compare to code in which we do not replace the prototype value.

    var Bar = function Bar() { };

    var BarInstance = new Bar();

    console.log(BarInstance.constructor === Bar); // Logs true.
    console.log(BarInstance.constructor); // Logs Bar()

</script></body></html>
```

If you intend to replace the default **prototype** property (common with some JS OOP patterns) set up by JavaScript, you should wire back together a constructor property that references the constructor function. In the following sample, we alter our previous code so that the ***constructor*** property will again provide a reference to the proper constructor function.

Sample: sample126.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var Foo = function Foo() { };

    Foo.prototype = { constructor: Foo };

    var FooInstance = new Foo();

    console.log(FooInstance.constructor === Foo); // Logs true.
    console.log(FooInstance.constructor); // Logs Foo()

</script></body></html>
```

Instances that inherit properties from prototype will always get the latest values

The prototype property is dynamic in the sense that instances will always get the latest value from the prototype regardless of when it was instantiated, changed, or appended. In the code that follows, we create a ***Foo*** constructor, add the property ***x*** to the **prototype**, and then create an instance of ***Foo()*** named ***FooInstance***. Next, we log the value of ***x***. Then we update the prototype's value of ***x*** and log it again to find that our instance has access to the latest value found in the **prototype** object.

Sample: sample127.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var Foo = function Foo() { };

    Foo.prototype.x = 1;

    var FooInstance = new Foo();

    console.log(FooInstance.x); // Logs 1.

    Foo.prototype.x = 2;

    console.log(FooInstance.x); // Logs 2, the FooInstance was updated.

</script></body></html>
```

Given how the lookup chain works, this behavior should not be that surprising. If you are wondering, this works the same regardless of whether you use the default **`prototype`** object or override it with your own. In the next sample, I replace the default **`prototype`** object to demonstrate this fact.

Sample: sample128.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var Foo = function Foo() { };

    Foo.prototype = { x: 1 }; // The logs that follow still work the same.

    var FooInstance = new Foo();

    console.log(FooInstance.x); // Logs 1.

    Foo.prototype.x = 2;

    console.log(FooInstance.x); // Logs 2, the FooInstance was updated.

</script></body></html>
```

Replacing the `prototype` property with a new object does not update former instances

You might think that you can replace the **`prototype`** property entirely at any time and that all instances will be updated, but this is not correct. When you create an instance, that instance will be tied to the **`prototype`** that was minted at the time of instantiation.

Providing a new object as the prototype property does not update the connection between instances already created and the new **prototype**.

But remember, as I stated previously, you can update or add to the originally created **prototype** object and those values remain connected to the first instance(s).

Sample: sample129.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var Foo = function Foo() { };

    Foo.prototype.x = 1;

    var FooInstance = new Foo();

    console.log(FooInstance.x); // Logs 1, as you think it would.

    // Now let's replace/override the prototype object with a new Object()
object.
    Foo.prototype = { x: 2 };

    console.log(FooInstance.x); // Logs 1. WHAT? Shouldn't it log 2 because
we just updated prototype?
    /* FooInstance still references the same state of the prototype object
that was there when it was instantiated. */

    // Create a new instance of Foo()
    var NewFooInstance = new Foo();

    // The new instance is now tied to the new prototype object value (i.e.
{x:2};).
    console.log(NewFooInstance.x); // Logs 2.

</script></body></html>
```

The key idea to take away here is that an object's prototype should not be replaced with a new object once you start creating instances. Doing so will result in instances that have a link to different prototypes.

User-defined constructors can leverage the same prototype inheritance as native constructors

Hopefully at this point in the chapter, it is sinking in how JavaScript itself leverages the **prototype** property for inheritance (e.g., **Array.prototype**). This same pattern can be leveraged when creating non-native, user-defined constructor functions. In the following

sample, we take the classic ***Person*** object and mimic the pattern that JavaScript uses for inheritance.

Sample: sample130.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var Person = function () { };

    // All Person instances inherit the legs, arms, and countLimbs
properties.
    Person.prototype.legs = 2;
    Person.prototype.arms = 2;
    Person.prototype.countLimbs = function () { return this.legs + this.arms;
};

    var chuck = new Person();

    console.log(chuck.countLimbs()); // Logs 4.

</script></body></html>
```

In this code, a ***Person()*** constructor function is created. We then add properties to the **prototype** property of ***Person()***, which can be inherited by all instances. Clearly, you can leverage the prototype chain in your code the same way that JavaScript leverages it for native object inheritance.

As a good example of how you might leverage this, you can create a constructor function whose instances inherit the ***legs*** and ***arms*** properties if they are not provided as parameters. In the following sample, if the ***Person()*** constructor is sent parameters, the parameters are used as instance properties, but if one or more parameters are not provided, there is a fallback. These instance properties then shadow or mask the inherited properties, giving you the best of both worlds.

Sample: sample131.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var Person = function (legs, arms) {
        // Shadow prototype value.
        if (legs !== undefined) { this.legs = legs; }
        if (arms !== undefined) { this.arms = arms; }
    };

    Person.prototype.legs = 2;
    Person.prototype.arms = 2;
    Person.prototype.countLimbs = function () { return this.legs + this.arms;
};
```

```
    var chuck = new Person(0, 0);

    console.log(chuck.countLimbs()); // Logs 0.

</script></body></html>
```

Creating inheritance chains (the original intention)

Prototypal inheritance was conceived to allow inheritance chains that mimic the inheritance patterns found in traditional *object oriented programming* languages. In order for one object to inherit from another object in JavaScript, all you have to do is instantiate an instance of the object you want to inherit from and assign it to the **`prototype`** property of the object that is doing the inheriting.

In the code sample that follows, ***`Chef`*** objects (i.e. ***`cody`***) inherit from **`Person()`**. This means that if a property is not found in a ***`Chef`*** object, it will then be looked for on the prototype of the function that created ***`Person()`*** objects. To wire up the inheritance, all you have to do is instantiate an instance of ***`Person()`*** as the value for **`Chef.prototype`** (i.e. **`Chef.prototype = new Person();`**).

Sample: sample132.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var Person = function () { this.bar = 'bar' };
    Person.prototype.foo = 'foo';

    var Chef = function () { this.goo = 'goo' };
    Chef.prototype = new Person();
    var cody = new Chef();

    console.log(cody.foo); // Logs 'foo'.
    console.log(cody.goo); // Logs 'goo'.
    console.log(cody.bar); // Logs 'bar'.

</script></body></html>
```

All we did in this sample was leverage a system that was already in place with the native objects. Consider that ***`Person()`*** is not unlike the default **`Object()`** value for prototype properties. In other words, this is exactly what happens when a prototype property, containing its default empty **`Object()`** value, looks to the prototype of the constructor function created (i.e. **`Object.prototype`**) for inherited properties.

Chapter 15 `Array()`

Conceptual overview of using `Array()` objects

An array is an ordered list of values typically created with the intention of looping through numerically indexed values, beginning with the index zero. What you need to know is that arrays are numerically ordered sets, as opposed to objects which have property names associated with values in non-numeric order. Essentially, arrays use numbers as a lookup key, while objects have user-defined property names. JavaScript does not have true associative arrays, but objects can be used to achieve the functionality of associative arrays.

In the following sample, I store four strings in ***`myArray`*** that I can access using a numeric index. I compare and contrast ***`myArray`*** to an object literal mimicking an associative array.

Sample: sample133.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myArray = ['blue', 'green', 'orange', 'red'];

    console.log(myArray[0]); // Logs blue using the 0 index to access the
string in myArray.

    // Versus

    var myObject = { // aka an associative array/hash, known as an object in
JavaScript.
        'blue': 'blue',
        'green': 'green',
        'orange': 'orange',
        'red': 'red'
    };

    console.log(myObject['blue']); // Logs blue.

</script></body></html>
```

Notes
Arrays can hold any type of values, and these values can be updated or deleted at any time.

If you need a hash (aka associative array), an object is the closest solution.

An `Array()` is just a special type of `Object()`. That is, `Array()` instances are basically `Object()` instances with a couple of extra functions (e.g., `.length` and a built-in numeric index).

Values contained in an array are commonly referred to as elements.

`Array()` parameters

You can pass the values of an array instance to the constructor as comma-separated parameters (e.g., `new Array('foo', 'bar');`). The `Array()` constructor can take up to 4,294,967,295 parameters.

However, if only one parameter is sent to the `Array()` constructor and that value is an integer (e.g., '1', '123', or '1.0'), it will be used to set up the `length` of the array, and will not be used as a value contained within the array.

Sample: sample134.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var foo = new Array(1, 2, 3);
    var bar = new Array(100);

    console.log(foo[0], foo[2]); // Logs '1 3'.
    console.log(bar[0], bar.length); // Logs 'undefined 100'.

</script></body></html>
```

`Array()` properties and methods

The `Array()` object has the following properties (not including inherited properties and methods):

Properties (e.g., `Array.prototype`):

- prototype

Array object instance properties and methods

Array object instances have the following properties and methods (not including inherited properties and methods):

Instance Properties (e.g., `var myArray = ['foo', 'bar']; myArray.length;`):

- constructor

- length

Instance Methods (e.g., `var myArray = ['foo']; myArray.pop();`):

- pop()
- push()
- reverse()
- shift()
- sort()
- splice()
- unshift()
- concat()
- join()
- slice()

Creating arrays

Like most of the objects in JavaScript, an array object can be created using the **`new`** operator in conjunction with the **`Array()`** constructor, or by using the literal syntax.

In the following sample, I create the ***`myArray1`*** array with predefined values using the **`Array()`** constructor, and then ***`myArray2`*** using literal notation.

Sample: sample135.html

```
<!DOCTYPE html><html lang="en"><body><script>

    // Array() constructor.
    var myArray1 = new Array('blue', 'green', 'orange', 'red');

    console.log(myArray1); // Logs ["blue", "green", "orange", "red"]

    // Array literal notation.
    var myArray2 = ['blue', 'green', 'orange', 'red'];

    console.log(myArray2); // logs ["blue", "green", "orange", "red"]

</script></body></html>
```

It is more common to see an array defined using the literal syntax, but it should be noted that this shortcut is merely concealing the use of the `Array()` constructor.

Notes
In practice, the array literal is typically all you will ever need.

Regardless of how an array is defined, if you do not provide any predefined values to the array, it will still be created but will simply contain no values.

Adding and updating values in arrays

A value can be added to an array at any index, at any time. In the sample that follows, we add a value to the numeric index 50 of an empty array. What about all the indexes before 50? Well, like I said, you can add a value to an array at any index, at any time. But if you add a value to the numeric index 50 of an empty array, JavaScript will fill in all of the necessary indexes before it with `undefined` values.

Sample: sample136.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myArray = [];
    myArray[50] = 'blue';
    console.log(myArray.length); /* Logs 51 (0 is counted) because JS created
values 0 to 50 before "blue".*/

</script></body></html>
```

Additionally, considering the dynamic nature of JavaScript and the fact that JavaScript is not strongly typed, an array value can be updated at any time and the value contained in the index can be any legal value. In the following sample, I change the value at the numeric index 50 to an object.

Sample: sample137.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myArray = [];
    myArray[50] = 'blue';
    myArray[50] = { 'color': 'blue' }; // Change object type from string to
Object() object.
    console.log(myArray[50]); // Logs 'Object {color="blue"}'.
```

```
    // Using brackets to access the index in the array, then the property
blue.
    console.log(myArray[50]['color']); // Logs 'blue'.

    // Using dot notation.
    console.log(myArray[50].color); // Logs 'blue'.

</script></body></html>
```

Length vs. index

An array starts indexing values at zero. This means that the first numeric slot to hold a value in an array looks like ***myArray[0]***. This can be a bit confusing—if I create an array with a single value, the index of the value is 0 while the length of the array is 1. Make sure you understand that the length of an array represents the number of values contained within the array, while the numeric index of the array starts at zero.

In the following sample, the string value ***blue*** is contained in the ***myArray*** array at the numeric index 0, but since the array contains one value, the length of the array is 1.

Sample: sample138.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myArray = ['blue'] // The index 0 contains the string value 'blue'.
    console.log(myArray[0]); // Logs 'blue'.
    console.log(myArray.length); // Logs 1.

</script></body></html>
```

Defining arrays with a predefined `length`

As I mentioned earlier, by passing a single integer parameter to the **`Array()`** constructor, it's possible to predefine the array's length, or the number of values it will contain. In this case, the constructor makes an exception and assumes you want to set the length of the array and not pre-populate the array with values.

In the next sample, we set up the ***myArray*** array with a predefined length of 3. Again, we are configuring the *length* of the array, not passing it a value to be stored at the 0 index.

Sample: sample139.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myArray = new Array(3);
```

```
    console.log(myArray.length); // Logs 3 because we are passing one numeric
parameter.
    console.log(myArray[0]); // Logs undefined.

</script></body></html>
```

Notes
Providing a predefined **`length`** will give each numeric index, up to the length specified, an associated value of **`undefined`**.

You might be wondering if it is possible to create a predefined array containing only one numeric value. Yes, it is—by using the literal form **`var myArray = [4]`**.

Setting array length can add or remove values

The **`length`** property of an array object can be used to get or set the length of an array. As shown previously, setting the length greater than the actual number of values contained in the array will add **`undefined`** values to the array. What you might not expect is that you can actually remove values from an array by setting the length value to a number less than the number of values contained in the array.

Sample: sample140.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myArray = ['blue', 'green', 'orange', 'red'];
    console.log(myArray.length); // Logs 4.
    myArray.length = 99;
    console.log(myArray.length); // Logs 99, remember we set the length, not
an index.
    myArray.length = 1; // Removed all but one value, so index [1] is gone!
    console.log(myArray[1]); // Logs undefined.

    console.log(myArray); // Logs '["blue"]'.

</script></body></html>
```

Arrays containing other arrays (aka multidimensional arrays)

Since an array can hold any valid JavaScript value, an array can contain other arrays. When this is done, the array containing encapsulated arrays is considered a *multidimensional* array. Accessing encapsulated arrays is done by *bracket chaining*. In the following sample, we create an array literal that contains an array, inside of which we create another array literal, inside of which we create another array literal, containing a string value at the 0 index.

Sample: sample141.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myArray = [[[['4th dimension']]]];
    console.log(myArray[0][0][0][0]); // Logs '4th dimension'.

</script></body></html>
```

This code example is rather silly, but you get the idea that arrays can contain other arrays and you can access encapsulated arrays indefinitely.

Looping over an array, backwards and forwards

The simplest and arguably the fastest way to loop over an array is to use the **`while`** loop.

In the following code, we loop from the beginning of the index to the end.

Sample: sample142.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myArray = ['blue', 'green', 'orange', 'red'];

    var myArrayLength = myArray.length; // Cache array length to avoid
unnecessary lookup.
    var counter = 0; // Set up counter.

    while (counter < myArrayLength) { // Run if counter is less than array
length.
        console.log(myArray[counter]); // Logs 'blue', 'green', 'orange',
'red'.
        counter++; // Add 1 to the counter.
    }

</script></body></html>
```

And now we loop from the end of the index to the beginning.

Sample: sample143.html

```
<!DOCTYPE html><html lang="en"><body><script>

    var myArray = ['blue', 'green', 'orange', 'red'];

    var myArrayLength = myArray.length;
```

```
    while (myArrayLength--) {                // If length is not zero, loop
and subtract 1.
        console.log(myArray[myArrayLength]);  // Logs 'red', 'orange',
'green', 'blue'.
    }

</script></body></html>
```

If you are wondering why I am not showing **for** loops here, it is because **while** loops have fewer moving parts and I believe they are easier to read.

Chapter 16 `Math` Function

Conceptual overview of the built-in `Math` object

The `Math` object contains static properties and methods for mathematically dealing with numbers or providing mathematical constants (e.g., `Math.PI;`). This object is built into JavaScript, as opposed to being based on a `Math()` constructor that creates math instances.

Notes

It might seem odd that `Math` starts with a capitalized letter since you do not instantiate an instance of a `Math` object. Do not be thrown off by this. Simply be aware that JavaScript sets this object up for you.

`Math` properties and methods

The `Math` object has the following properties and methods:

Properties (e.g., `Math.PI;`):

- `E`
- `LN2`
- `LN10`
- `LOG2E`
- `LOG10E`
- `PI`
- `SQRT1_2`
- `SQRT2`

Methods (e.g., `Math.random();`):

- `abs()`
- `acos()`
- `asin()`
- `atan()`
- `atan2()`

- `ceil()`
- `cos()`
- `exp()`
- `floor()`
- `log()`
- `max()`
- `min()`
- `pow()`
- `random()`
- `round()`
- `sin()`
- `sqrt()`
- `tan()`

`Math` is not a constructor function

The `Math` object is unlike the other built-in objects that are instantiated. `Math` is a one-off object created to house static properties and methods, ready to be used when dealing with numbers. Just remember, there is no way to create an instance of `Math`, as there is no constructor.

`Math` has constants you cannot augment or mutate

Many of the `Math` properties are constants that cannot be mutated. Since this is a departure from the mutable nature of JavaScript, these properties are in all caps (e.g., `Math.PI;`). Do not confuse these property constants for constructor functions due to the capitalization of their first letter. They are simply object properties that cannot be changed.

Notes
User-defined constants are not possible in JavaScript 1.5, ECMA-262, Edition 3.

Review

The following points summarize what you should have learned by reading this book (and investigating the code examples). Read each summary, and if you don't understand what is being said, return to the topic in the book.

- An object is made up of named properties that store values.
- Most everything in JavaScript can act like an object. Complex values are objects, and primitive values can be treated like objects. This is why you may hear people say that everything in JavaScript is an object.
- Objects are created by invoking a constructor function with the `new` keyword, or by using a shorthand literal expression.
- Constructor functions are objects (`Function()` objects), thus, in JavaScript, objects create objects.
- JavaScript offers nine native constructor functions: `Object()`, `Array()`, `String()`, `Number()`, `Boolean()`, `Function()`, `Date()`, `RegExp()`, and `Error()`. The `String()`, `Number()`, and `Boolean()` constructors are dual-purposed in providing a) primitive values and b) object wrappers when needed, so that primitive values can act like objects.
- The values `null`, `undefined`, `"string"`, `10`, `true`, and `false` are all primitive values, without an object nature unless treated like an object.
- When the `Object()`, `Array()`, `String()`, `Number()`, `Boolean()`, `Function()`, `Date()`, `RegExp()`, and `Error()` constructor functions are invoked using the `new` keyword, an object is created that is known as a "complex object" or "reference object."
- `"string"`, `10`, `true`, and `false`, in their primitive forms, have no object qualities until they are used as objects; then JavaScript, behind the scenes, creates temporary wrapper objects so that such values can act like objects.
- Primitive values are stored by value, and when copied, are literally copied. Complex object values on the other hand are stored by reference, and when copied, are copied by reference.
- Primitive values are equal to other primitive values when their values are equal, whereas complex objects are equal only when they reference the same value.

That is: a complex value is equal to another complex value when both refer to the same object.

- Due to the nature of complex objects and references, JavaScript objects have dynamic properties.

- JavaScript is mutable, which means that native objects and user-defined object properties can be manipulated at any time.

- Getting/setting/updating an object's properties is done by using dot notation or bracket notation. Bracket notation is convenient when the name of the object property being manipulated is in the form of an expression (e.g., `Array['prototype']['join'].apply()`).

- When referencing object properties, a lookup chain is used to first look at the object that was referenced for the property. If the property is not there, the property is looked for on the constructor function's `prototype` property. If it's not found there, because the prototype holds an object value and the value is created from the `Object()` constructor, the property is looked for on the `Object()` constructor's `prototype` property (`Object.prototype`). If the property is not found there, then the property is determined to be `undefined`.

- The `prototype` lookup chain is how inheritance (aka prototypal inheritance) was design to be accomplished in JavaScript.

- Because of the object property lookup chain (aka prototypal inheritance), all objects inherit from `Object()` simply because the `prototype` property is, itself, an `Object()` object.

- JavaScript functions are first-class citizens: functions are objects with properties and values.

- The `this` keyword, when used inside a function, is a generic way to reference the object containing the function.

- The value of `this` is determined during run time based on the context in which the function is called.

- Used in the global scope, the `this` keyword refers to the global object.

- JavaScript uses functions as a way to create a unique scope.

- JavaScript provides the global scope, and it's in this scope that all JavaScript code exists.

- Functions (specifically, encapsulated functions) create a scope chain for resolving variable lookups.

- The scope chain is set up based on the way code is written, not necessarily by the context in which a function is invoked. This permits a function to have access to the scope in which it was originally written, even if the function is called from a different context. This result is known as a closure.

- Function expressions and variables declared inside a function without using `var` become global properties. However, function statements inside of a function scope remain defined in the scope in which they are written.

- Functions and variables declared (without `var`) in the global scope become properties of the global object.

- Functions and variables declared (with `var`) in the global scope become global variables.