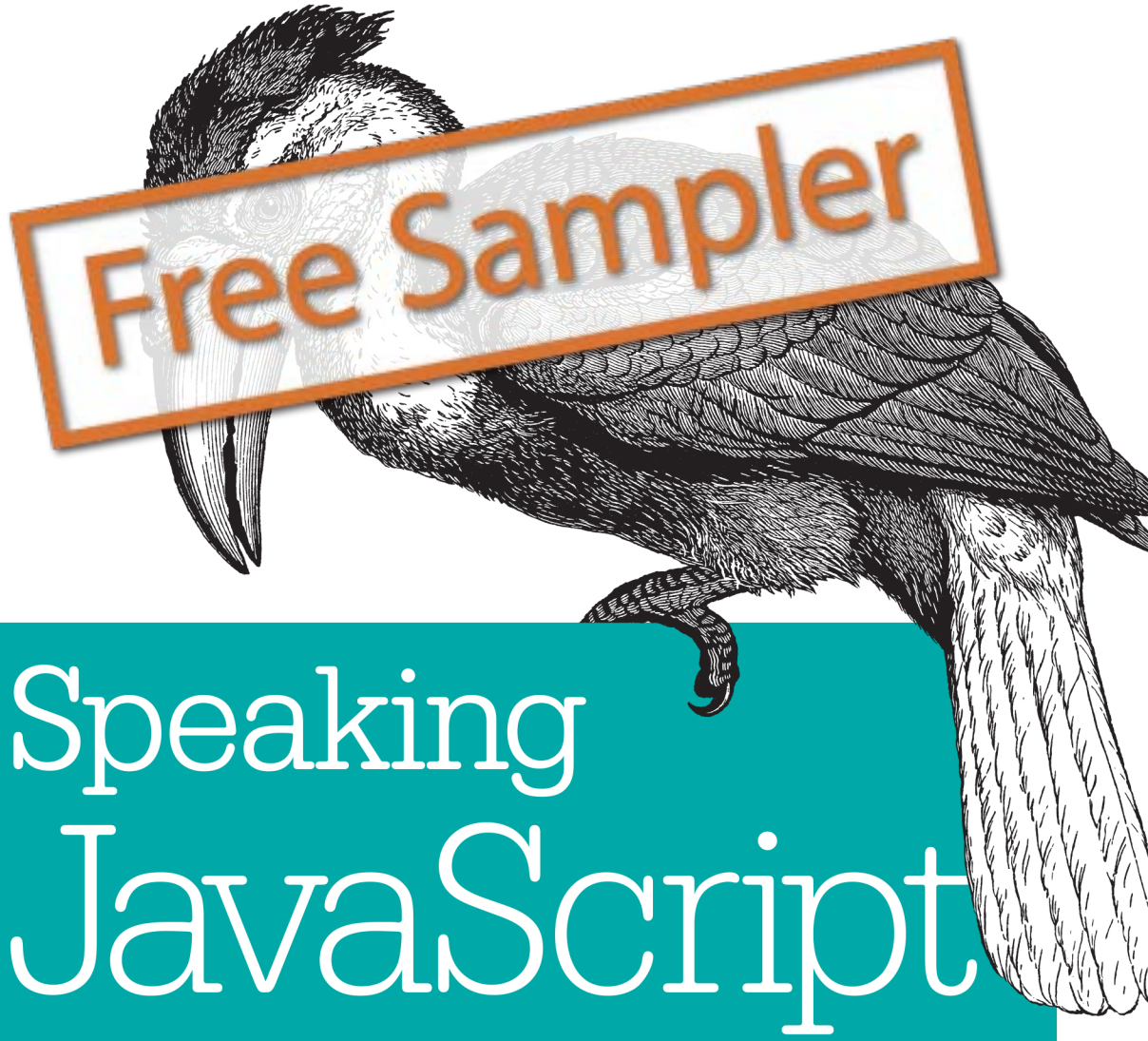
O'REILLY®



Speaking JavaScript

AN IN-DEPTH GUIDE FOR PROGRAMMERS

Dr. Axel Rauschmayer

O'REILLY®

Speaking JavaScript

If you're ready to dive into JavaScript, either as a programmer new to the language or as a JavaScript developer looking to increase your skills, this guide is the one source you need.

Written by a programmer for programmers, *Speaking JavaScript* provides a quick start guide to help you learn JavaScript quickly and properly. It also contains a complete and easy-to-read reference that covers language features in depth. Altogether, four standalone parts help you approach JavaScript in different ways.

- JavaScript quick start: If you're familiar with object-oriented programming, this part teaches you just enough of the language to help you be productive.
- JavaScript in depth: Learn the details of ECMAScript 5 from syntax, variables, functions, and object-oriented programming to regular expressions and JSON with lots of examples. Pick a topic and jump in.
- Background: This part places JavaScript in historical and technical context, including how it's related to other programming languages.
- Tips, tools, and libraries: Survey existing style guides, best practices, advanced techniques, module systems, package managers, build tools, and other learning resources.

"The most concise and yet complete JavaScript book written to date for the initiated programmer."

—**Cody Lindley**
Front end engineer and author of *JavaScript Enlightenment*

"*Speaking JavaScript* is very well written and extremely easy to understand. It is indispensable for JavaScript novices and an excellent reference for experienced web developers."

—**Ariya Hidayat**
Director of Engineering at Shape Security and author of the PhantomJS open source project

Dr. Axel Rauschmayer specializes in JavaScript and web development. He blogs at 2ality.com, is a trainer for Ecmanauten, edits *JavaScript Weekly*, and organizes the MunichJS user group.

PROGRAMMING / JAVASCRIPT

US $39.99 CAN $41.99

ISBN: 978-1-449-36503-5

Twitter: @oreillymedia
facebook.com/oreilly

Praise for *Speaking JavaScript*

"A lot of people think JavaScript is simple and in many cases it is. But in its elegant simplicity lies a deeper functionality that if leveraged properly, can produce amazing results. Axel's ability to distill this into an approachable reference will certainly help both aspiring and experienced developers achieve a better understanding of the language."

—Rey Bango
Advocate for cross-browser development, proponent of the open web, and lover of the JavaScript programming language

"Axel's writing style is succinct, to the point, yet at the same time extremely detailed. The many code examples make even the most complex topics in the book easy to understand."

—Mathias Bynens
Belgian web standards enthusiast who likes HTML, CSS, JavaScript, Unicode, performance, and security

"*Speaking JavaScript* is a modern, up to date book perfectly aimed at the existing experienced programmer ready to take a deep dive into JavaScript. Without wasting time on laborious explanations, Dr. Rauschmayer quickly cuts to the core of JavaScript and its various concepts and gets developers up to speed quickly with a language that seems intent on taking over the developer world."

—Peter Cooper
Publisher, entrepreneur, and co-organizer of Fluent Conference

"If you have enjoyed Axel's blog, then you'll love this book. His book is filled with tons of bite-sized code snippets to aid in the learning process. If you want to dive deep and understand the ins and outs of JavaScript, then I highly recommend this book."

—Elijah Manor
Christian, family man, and front end web developer for Dave Ramsey; enjoys speaking, blogging, and tweeting

"This book opens the door into the modern JavaScript community with just enough background and plenty of in-depth introduction to make it seem like you've been with the community from the start."

—Mitch Pronschinske
DZone Editor

"After following Dr. Axel Rauschmayer's work for a few years, I was delighted to learn that he was writing a book to share his deep expertise of JavaScript with those getting started with the language. I've read many JavaScript books, but none that show the attention to detail and comprehensiveness of Speaking JS, without being boring or overwhelming. I'll be recommending this book for years to come."

—Guillermo Rauch
Speaker, creator of socket.io, mongoose, early Node.js contributor, author of "Smashing Node.js", founder of LearnBoost/Cloudup (acq. by Wordpress in 2013), and Open Academy mentor

Speaking JavaScript

Axel Rauschmayer

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Speaking JavaScript

by Axel Rauschmayer

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Simon St. Laurent and Amy Jollymore
Production Editor: Kara Ebrahim
Copyeditor: Rachel Monaghan
Proofreader: Charles Roumeliotis
Indexer: Ellen Troutman
Cover Designer: Randy Comer
Interior Designer: David Futato
Illustrator: Rebecca Demarest

March 2014: First Edition

Revision History for the First Edition:

2014-02-20: First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449365035* for release details.

ISBN: 978-1-449-36503-5

[LSI]

Table of Contents

CHAPTER 1

Basic JavaScript

This chapter is about "Basic JavaScript," a name I chose for a subset of JavaScript that is as concise as possible while still enabling you to be productive. When you are starting to learn JavaScript, I recommend that you program in it for a while before moving on to the rest of the language. That way, you don't have to learn everything at once, which can be confusing.

Background

This section gives a little background on JavaScript to help you understand why it is the way it is.

JavaScript Versus ECMAScript

ECMAScript is the official name for JavaScript. A new name became necessary because there is a trademark on *Java* (held originally by Sun, now by Oracle). At the moment, Mozilla is one of the few companies allowed to officially use the name *JavaScript* because it received a license long ago. For common usage, the following rules apply:

- *JavaScript* means the programming language.
- *ECMAScript* is the name used by the language specification. Therefore, whenever referring to versions of the language, people say *ECMAScript*. The current version of JavaScript is ECMAScript 5; ECMAScript 6 is currently being developed.

Influences and Nature of the Language

JavaScript's creator, Brendan Eich, had no choice but to create the language very quickly (or other, worse technologies would have been adopted by Netscape). He borrowed from several programming languages: Java (syntax, primitive values versus objects), Scheme

and AWK (first-class functions), Self (prototypal inheritance), and Perl and Python (strings, arrays, and regular expressions).

JavaScript did not have exception handling until ECMAScript 3, which explains why the language so often automatically converts values and so often fails silently: it initially couldn't throw exceptions.

On one hand, JavaScript has quirks and is missing quite a bit of functionality (block-scoped variables, modules, support for subclassing, etc.). On the other hand, it has several powerful features that allow you to work around these problems. In other languages, you learn language features. In JavaScript, you often learn patterns instead.

Given its influences, it is no surprise that JavaScript enables a programming style that is a mixture of functional programming (higher-order functions; built-in `map`, `reduce`, etc.) and object-oriented programming (objects, inheritance).

Syntax

This section explains basic syntactic principles of JavaScript.

An Overview of the Syntax

A few examples of syntax:

```
// Two slashes start single-line comments

var x;  // declaring a variable

x = 3 + y;  // assigning a value to the variable `x`

foo(x, y);  // calling function `foo` with parameters `x` and `y`
obj.bar(3);  // calling method `bar` of object `obj`

// A conditional statement
if (x === 0) {  // Is `x` equal to zero?
    x = 123;
}

// Defining function `baz` with parameters `a` and `b`
function baz(a, b) {
    return a + b;
}
```

Note the two different uses of the equals sign:

- A single equals sign (=) is used to assign a value to a variable.
- A triple equals sign (===) is used to compare two values (see "Equality Operators" on page 14).

Statements Versus Expressions

To understand JavaScript's syntax, you should know that it has two major syntactic categories: statements and expressions:

- Statements "do things." A program is a sequence of statements. Here is an example of a statement, which declares (creates) a variable `foo`:

  ```
  var foo;
  ```

- Expressions produce values. They are function arguments, the right side of an assignment, etc. Here's an example of an expression:

  ```
  3 * 7
  ```

The distinction between statements and expressions is best illustrated by the fact that JavaScript has two different ways to do `if-then-else`—either as a statement:

```
var x;
if (y >= 0) {
    x = y;
} else {
    x = -y;
}
```

or as an expression:

```
var x = y >= 0 ? y : -y;
```

You can use the latter as a function argument (but not the former):

```
myFunction(y >= 0 ? y : -y)
```

Finally, wherever JavaScript expects a statement, you can also use an expression; for example:

```
foo(7, 1);
```

The whole line is a statement (a so-called *expression statement*), but the function call `foo(7, 1)` is an expression.

Semicolons

Semicolons are optional in JavaScript. However, I recommend always including them, because otherwise JavaScript can guess wrong about the end of a statement. The details are explained in "Automatic Semicolon Insertion" on page 59.

Semicolons terminate statements, but not blocks. There is one case where you will see a semicolon after a block: a function expression is an expression that ends with a block. If such an expression comes last in a statement, it is followed by a semicolon:

```
// Pattern: var _ = ___;
var x = 3 * 7;
var f = function () { };  // function expr. inside var decl.
```

Comments

JavaScript has two kinds of comments: single-line comments and multiline comments. Single-line comments start with `//` and are terminated by the end of the line:

```
x++; // single-line comment
```

Multiline comments are delimited by `/*` and `*/`:

```
/* This is
   a multiline
   comment.
 */
```

Variables and Assignment

Variables in JavaScript are declared before they are used:

```
var foo;  // declare variable `foo`
```

Assignment

You can declare a variable and assign a value at the same time:

```
var foo = 6;
```

You can also assign a value to an existing variable:

```
foo = 4;  // change variable `foo`
```

Compound Assignment Operators

There are compound assignment operators such as `+=`. The following two assignments are equivalent:

```
x += 1;
x = x + 1;
```

Identifiers and Variable Names

Identifiers are names that play various syntactic roles in JavaScript. For example, the name of a variable is an identifier. Identifiers are case sensitive.

Roughly, the first character of an identifier can be any Unicode letter, a dollar sign (`$`), or an underscore (`_`). Subsequent characters can additionally be any Unicode digit. Thus, the following are all legal identifiers:

```
arg0
_tmp
$elem
π
```

The following identifiers are *reserved words*—they are part of the syntax and can't be used as variable names (including function names and parameter names):

```
arguments   break    case       catch
class       const    continue   debugger
default     delete   do         else
enum        export   extends    false
finally     for      function   if
implements  import   in         instanceof
interface   let      new        null
package     private  protected  public
return      static   super      switch
this        throw    true       try
typeof      var      void       while
```

The following three identifiers are not reserved words, but you should treat them as if they were:

```
Infinity
NaN
undefined
```

Lastly, you should also stay away from the names of standard global variables (see Chapter 23). You can use them for local variables without breaking anything, but your code still becomes confusing.

Values

JavaScript has many values that we have come to expect from programming languages: booleans, numbers, strings, arrays, and so on. All values in JavaScript have *properties*. Each property has a *key* (or *name*) and a *value*. You can think of properties like fields of a record. You use the dot (`.`) operator to read a property:

```
value.propKey
```

For example, the string `'abc'` has the property `length`:

```
> var str = 'abc';
> str.length
3
```

The preceding can also be written as:

```
> 'abc'.length
3
```

The dot operator is also used to assign a value to a property:

```
> var obj = {};  // empty object
> obj.foo = 123; // create property `foo`, set it to 123
123
> obj.foo
123
```

And you can use it to invoke methods:

```
> 'hello'.toUpperCase()
'HELLO'
```

In the preceding example, we have invoked the method `toUpperCase()` on the value `'hello'`.

Primitive Values Versus Objects

JavaScript makes a somewhat arbitrary distinction between values:

- The *primitive values* are booleans, numbers, strings, `null`, and `undefined`.
- All other values are *objects*.

A major difference between the two is how they are compared; each object has a unique identity and is only (strictly) equal to itself:

```
> var obj1 = {};  // an empty object
> var obj2 = {};  // another empty object
> obj1 === obj2
false
> obj1 === obj1
true
```

In contrast, all primitive values encoding the same value are considered the same:

```
> var prim1 = 123;
> var prim2 = 123;
> prim1 === prim2
true
```

The next two sections explain primitive values and objects in more detail.

Primitive Values

The following are all of the primitive values (or *primitives* for short):

- Booleans: `true`, `false` (see "Booleans" on page 12)
- Numbers: `1736`, `1.351` (see "Numbers" on page 14)
- Strings: `'abc'`, `"abc"` (see "Strings" on page 15)
- Two "nonvalues": `undefined`, `null` (see "undefined and null" on page 10)

Primitives have the following characteristics:

Compared by value
: The "content" is compared:

```
> 3 === 3
true
> 'abc' === 'abc'
true
```

Always immutable
: Properties can't be changed, added, or removed:

```
> var str = 'abc';

> str.length = 1; // try to change property `length`
> str.length      // ⇒ no effect
3

> str.foo = 3; // try to create property `foo`
> str.foo      // ⇒ no effect, unknown property
undefined
```

(Reading an unknown property always returns `undefined`.)

Objects

All nonprimitive values are *objects*. The most common kinds of objects are:

- *Plain objects*, which can be created by *object literals* (see "Single Objects" on page 25):

```
{
    firstName: 'Jane',
    lastName: 'Doe'
}
```

  The preceding object has two properties: the value of property `firstName` is `'Jane'` and the value of property `lastName` is `'Doe'`.

- *Arrays*, which can be created by *array literals* (see "Arrays" on page 28):

```
[ 'apple', 'banana', 'cherry' ]
```

The preceding array has three elements that can be accessed via numeric indices. For example, the index of `'apple'` is 0.

- *Regular expressions*, which can be created by *regular expression literals* (see "Regular Expressions" on page 31):

```
/^a+b+$/
```

Objects have the following characteristics:

Compared by reference
: Identities are compared; every value has its own identity:

```
> {} === {}  // two different empty objects
false

> var obj1 = {};
> var obj2 = obj1;
> obj1 === obj2
true
```

Mutable by default
: You can normally freely change, add, and remove properties (see "Single Objects" on page 25):

```
> var obj = {};
> obj.foo = 123; // add property `foo`
> obj.foo
123
```

undefined and null

Most programming languages have values denoting missing information. JavaScript has two such "nonvalues," `undefined` and `null`:

- `undefined` means "no value." Uninitialized variables are `undefined`:

```
> var foo;
> foo
undefined
```

Missing parameters are `undefined`:

```
> function f(x) { return x }
> f()
undefined
```

If you read a nonexistent property, you get `undefined`:

```
> var obj = {}; // empty object
> obj.foo
undefined
```

- `null` means "no object." It is used as a nonvalue whenever an object is expected (parameters, last in a chain of objects, etc.).

`undefined` and `null` have no properties, not even standard methods such as `toString()`.

Checking for undefined or null

Functions normally allow you to indicate a missing value via either `undefined` or `null`. You can do the same via an explicit check:

```
if (x === undefined || x === null) {
    ···
}
```

You can also exploit the fact that both `undefined` and `null` are considered `false`:

```
if (!x) {
    ···
}
```

`false`, `0`, `NaN`, and `''` are also considered `false` (see "Truthy and Falsy" on page 13).

Categorizing Values Using typeof and instanceof

There are two operators for categorizing values: `typeof` is mainly used for primitive values, while `instanceof` is used for objects.

`typeof` looks like this:

```
typeof value
```

It returns a string describing the "type" of `value`. Here are some examples:

```
> typeof true
'boolean'
> typeof 'abc'
'string'
> typeof {} // empty object literal
'object'
> typeof [] // empty array literal
'object'
```

The following table lists all results of `typeof`:

Operand	Result
`undefined`	`'undefined'`
`null`	`'object'`
Boolean value	`'boolean'`
Number value	`'number'`
String value	`'string'`
Function	`'function'`
All other normal values	`'object'`
(Engine-created value)	JavaScript engines are allowed to create values for which `typeof` returns arbitrary strings (different from all results listed in this table).

`typeof null` returning `'object'` is a bug that can't be fixed, because it would break existing code. It does not mean that `null` is an object.

`instanceof` looks like this:

```
value instanceof Constr
```

It returns `true` if `value` is an object that has been created by the constructor `Constr` (see "Constructors: Factories for Objects" on page 28). Here are some examples:

```
> var b = new Bar();  // object created by constructor Bar
> b instanceof Bar
true

> {} instanceof Object
true
> [] instanceof Array
true
> [] instanceof Object  // Array is a subconstructor of Object
true

> undefined instanceof Object
false
> null instanceof Object
false
```

Booleans

The primitive boolean type comprises the values `true` and `false`. The following operators produce booleans:

- Binary logical operators: `&&` (And), `||` (Or)
- Prefix logical operator: `!` (Not)

- Comparison operators:
  - — Equality operators: `===`, `!==`, `==`, `!=`
  - — Ordering operators (for strings and numbers): `>`, `>=`, `<`, `<=`

Truthy and Falsy

Whenever JavaScript expects a boolean value (e.g., for the condition of an `if` statement), any value can be used. It will be interpreted as either `true` or `false`. The following values are interpreted as `false`:

- `undefined`, `null`
- Boolean: `false`
- Number: `-0`, `NaN`
- String: `''`

All other values (including all objects!) are considered `true`. Values interpreted as `false` are called *falsy*, and values interpreted as `true` are called *truthy*. `Boolean()`, called as a function, converts its parameter to a boolean. You can use it to test how a value is interpreted:

```
> Boolean(undefined)
false
> Boolean(0)
false
> Boolean(3)
true
> Boolean({}) // empty object
true
> Boolean([]) // empty array
true
```

Binary Logical Operators

Binary logical operators in JavaScript are *short-circuiting*. That is, if the first operand suffices for determining the result, the second operand is not evaluated. For example, in the following expressions, the function `foo()` is never called:

```
false && foo()
true  || foo()
```

Furthermore, binary logical operators return either one of their operands—which may or may not be a boolean. A check for truthiness is used to determine which one:

And (`&&`)
: If the first operand is falsy, return it. Otherwise, return the second operand:

```
> NaN && 'abc'
NaN
> 123 && 'abc'
'abc'
```

Or (||)
: If the first operand is truthy, return it. Otherwise, return the second operand:

```
> 'abc' || 123
'abc'
> '' || 123
123
```

Equality Operators

JavaScript has two kinds of equality:

- Normal, or "lenient," (in)equality: `==` and `!=`
- Strict (in)equality: `===` and `!==`

Normal equality considers (too) many values to be equal (the details are explained in "Normal (Lenient) Equality (==, !=)" on page 84), which can hide bugs. Therefore, always using strict equality is recommended.

Numbers

All numbers in JavaScript are floating-point:

```
> 1 === 1.0
true
```

Special numbers include the following:

`NaN` *("not a number")*
: An error value:

```
> Number('xyz')  // 'xyz' can't be converted to a number
NaN
```

`Infinity`
: Also mostly an error value:

```
> 3 / 0
Infinity
> Math.pow(2, 1024)  // number too large
Infinity
```

`Infinity` is larger than any other number (except `NaN`). Similarly, `-Infinity` is smaller than any other number (except `NaN`). That makes these numbers useful as default values (e.g., when you are looking for a minimum or a maximum).

Operators

JavaScript has the following arithmetic operators (see "Arithmetic Operators" on page 122):

- Addition: `number1 + number2`
- Subtraction: `number1 - number2`
- Multiplication: `number1 * number2`
- Division: `number1 / number2`
- Remainder: `number1 % number2`
- Increment: `++variable`, `variable++`
- Decrement: `--variable`, `variable--`
- Negate: `-value`
- Convert to number: `+value`

The global object `Math` (see "Math" on page 31) provides more arithmetic operations, via functions.

JavaScript also has operators for bitwise operations (e.g., bitwise And; see "Bitwise Operators" on page 124).

Strings

Strings can be created directly via string literals. Those literals are delimited by single or double quotes. The backslash (\) escapes characters and produces a few control characters. Here are some examples:

```
'abc'
"abc"

'Did she say "Hello"?'
"Did she say \"Hello\"?"

'That\'s nice!'
"That's nice!"

'Line 1\nLine 2'  // newline
'Backlash: \\'
```

Single characters are accessed via square brackets:

```
> var str = 'abc';
> str[1]
'b'
```

The property `length` counts the number of characters in the string:

```
> 'abc'.length
3
```

Like all primitives, strings are immutable; you need to create a new string if you want to change an existing one.

String Operators

Strings are concatenated via the plus (+) operator, which converts the other operand to a string if one of the operands is a string:

```
> var messageCount = 3;
> 'You have ' + messageCount + ' messages'
'You have 3 messages'
```

To concatenate strings in multiple steps, use the += operator:

```
> var str = '';
> str += 'Multiple ';
> str += 'pieces ';
> str += 'are concatenated.';
> str
'Multiple pieces are concatenated.'
```

String Methods

Strings have many useful methods (see "String Prototype Methods" on page 139). Here are some examples:

```
> 'abc'.slice(1)  // copy a substring
'bc'
> 'abc'.slice(1, 2)
'b'

> '\t xyz  '.trim()  // trim whitespace
'xyz'

> 'mjölnir'.toUpperCase()
'MJÖLNIR'

> 'abc'.indexOf('b')  // find a string
1
> 'abc'.indexOf('x')
-1
```

Statements

Conditionals and loops in JavaScript are introduced in the following sections.

Conditionals

The `if` statement has a `then` clause and an optional `else` clause that are executed depending on a boolean condition:

```
if (myvar === 0) {
    // then
}

if (myvar === 0) {
    // then
} else {
    // else
}

if (myvar === 0) {
    // then
} else if (myvar === 1) {
    // else-if
} else if (myvar === 2) {
    // else-if
} else {
    // else
}
```

I recommend always using braces (they denote blocks of zero or more statements). But you don't have to do so if a clause is only a single statement (the same holds for the control flow statements `for` and `while`):

```
if (x < 0) return -x;
```

The following is a `switch` statement. The value of `fruit` decides which `case` is executed:

```
switch (fruit) {
    case 'banana':
        // ...
        break;
    case 'apple':
        // ...
        break;
    default:  // all other cases
        // ...
}
```

The "operand" after `case` can be any expression; it is compared via `===` with the parameter of `switch`.

Loops

The `for` loop has the following format:

```
for ([«init»]; [«condition»]; [«post_iteration»])
    «statement»
```

`init` is executed at the beginning of the loop. `condition` is checked before each loop iteration; if it becomes `false`, then the loop is terminated. `post_iteration` is executed after each loop iteration.

This example prints all elements of the array `arr` on the console:

```
for (var i=0; i < arr.length; i++) {
    console.log(arr[i]);
}
```

The `while` loop continues looping over its body while its condition holds:

```
// Same as for loop above:
var i = 0;
while (i < arr.length) {
    console.log(arr[i]);
    i++;
}
```

The `do-while` loop continues looping over its body while its condition holds. As the condition follows the body, the body is always executed at least once:

```
do {
    // ...
} while (condition);
```

In all loops:

- `break` leaves the loop.
- `continue` starts a new loop iteration.

Functions

One way of defining a function is via a *function declaration*:

```
function add(param1, param2) {
    return param1 + param2;
}
```

The preceding code defines a function, `add`, that has two parameters, `param1` and `param2`, and returns the sum of both parameters. This is how you call that function:

```
> add(6, 1)
7
> add('a', 'b')
'ab'
```

Another way of defining `add()` is by assigning a *function expression* to a variable `add`:

```
var add = function (param1, param2) {
    return param1 + param2;
};
```

A function expression produces a value and can thus be used to directly pass functions as arguments to other functions:

```
someOtherFunction(function (p1, p2) { ... });
```

Function Declarations Are Hoisted

Function declarations are *hoisted*—moved in their entirety to the beginning of the current scope. That allows you to refer to functions that are declared later:

```
function foo() {
    bar();  // OK, bar is hoisted
    function bar() {
        ...
    }
}
```

Note that while `var` declarations are also hoisted (see "Variables Are Hoisted" on page 23), assignments performed by them are not:

```
function foo() {
    bar();  // Not OK, bar is still undefined
    var bar = function () {
        // ...
    };
}
```

The Special Variable arguments

You can call any function in JavaScript with an arbitrary amount of arguments; the language will never complain. It will, however, make all parameters available via the special variable `arguments`. `arguments` looks like an array, but has none of the array methods:

```
> function f() { return arguments }
> var args = f('a', 'b', 'c');
> args.length
3
> args[0]  // read element at index 0
'a'
```

Too Many or Too Few Arguments

Let's use the following function to explore how too many or too few parameters are handled in JavaScript (the function `toArray()` is shown in "Converting arguments to an Array" on page 21):

```
function f(x, y) {
    console.log(x, y);
    return toArray(arguments);
}
```

Additional parameters will be ignored (except by `arguments`):

```
> f('a', 'b', 'c')
a b
[ 'a', 'b', 'c' ]
```

Missing parameters will get the value `undefined`:

```
> f('a')
a undefined
[ 'a' ]
> f()
undefined undefined
[]
```

Optional Parameters

The following is a common pattern for assigning default values to parameters:

```
function pair(x, y) {
    x = x || 0;  // (1)
    y = y || 0;
    return [ x, y ];
}
```

In line (1), the `||` operator returns `x` if it is truthy (not `null`, `undefined`, etc.). Otherwise, it returns the second operand:

```
> pair()
[ 0, 0 ]
> pair(3)
[ 3, 0 ]
> pair(3, 5)
[ 3, 5 ]
```

Enforcing an Arity

If you want to enforce an *arity* (a specific number of parameters), you can check `arguments.length`:

```
function pair(x, y) {
    if (arguments.length !== 2) {
        throw new Error('Need exactly 2 arguments');
    }
    ...
}
```

Converting arguments to an Array

`arguments` is not an array, it is only *array-like* (see "Array-Like Objects and Generic Methods" on page 262). It has a property `length`, and you can access its elements via indices in square brackets. You cannot, however, remove elements or invoke any of the array methods on it. Thus, you sometimes need to convert `arguments` to an array, which is what the following function does (it is explained in "Array-Like Objects and Generic Methods" on page 262):

```
function toArray(arrayLikeObject) {
    return Array.prototype.slice.call(arrayLikeObject);
}
```

Exception Handling

The most common way to handle exceptions (see Chapter 14) is as follows:

```
function getPerson(id) {
    if (id < 0) {
        throw new Error('ID must not be negative: '+id);
    }
    return { id: id }; // normally: retrieved from database
}

function getPersons(ids) {
    var result = [];
    ids.forEach(function (id) {
        try {
            var person = getPerson(id);
            result.push(person);
        } catch (exception) {
            console.log(exception);
        }
    });
    return result;
}
```

The `try` clause surrounds critical code, and the `catch` clause is executed if an exception is thrown inside the `try` clause. Using the preceding code:

```
> getPersons([2, -5, 137])
[Error: ID must not be negative: -5]
[ { id: 2 }, { id: 137 } ]
```

Strict Mode

Strict mode (see "Strict Mode" on page 62) enables more warnings and makes JavaScript a cleaner language (nonstrict mode is sometimes called "sloppy mode"). To switch it on, type the following line first in a JavaScript file or a `<script>` tag:

```
'use strict';
```

You can also enable strict mode per function:

```
function functionInStrictMode() {
    'use strict';
}
```

Variable Scoping and Closures

In JavaScript, you declare variables via `var` before using them:

```
> var x;
> x = 3;
> y = 4;
ReferenceError: y is not defined
```

You can declare and initialize several variables with a single `var` statement:

```
var x = 1, y = 2, z = 3;
```

But I recommend using one statement per variable (the reason is explained in "Syntax" on page 382). Thus, I would rewrite the previous statement to:

```
var x = 1;
var y = 2;
var z = 3;
```

Because of hoisting (see "Variables Are Hoisted" on page 23), it is usually best to declare variables at the beginning of a function.

Variables Are Function-Scoped

The scope of a variable is always the complete function (as opposed to the current block). For example:

```
function foo() {
    var x = -512;
    if (x < 0) {  // (1)
        var tmp = -x;
        ···
    }
    console.log(tmp);  // 512
}
```

We can see that the variable `tmp` is not restricted to the block starting in line (1); it exists until the end of the function.

Variables Are Hoisted

Each variable declaration is *hoisted*: the declaration is moved to the beginning of the function, but assignments that it makes stay put. As an example, consider the variable declaration in line (1) in the following function:

```
function foo() {
    console.log(tmp); // undefined
    if (false) {
        var tmp = 3;  // (1)
    }
}
```

Internally, the preceding function is executed like this:

```
function foo() {
    var tmp;  // hoisted declaration
    console.log(tmp);
    if (false) {
        tmp = 3;  // assignment stays put
    }
}
```

Closures

Each function stays connected to the variables of the functions that surround it, even after it leaves the scope in which it was created. For example:

```
function createIncrementor(start) {
    return function () {  // (1)
        start++;
        return start;
    }
}
```

The function starting in line (1) leaves the context in which it was created, but stays connected to a live version of `start`:

```
> var inc = createIncrementor(5);
> inc()
6
> inc()
7
> inc()
8
```

A *closure* is a function plus the connection to the variables of its surrounding scopes. Thus, what `createIncrementor()` returns is a closure.

The IIFE Pattern: Introducing a New Scope

Sometimes you want to introduce a new variable scope—for example, to prevent a variable from becoming global. In JavaScript, you can't use a block to do so; you must use a function. But there is a pattern for using a function in a block-like manner. It is called *IIFE* (immediately invoked function expression, pronounced "iffy"):

```
(function () {  // open IIFE
    var tmp = ...;  // not a global variable
}());  // close IIFE
```

Be sure to type the preceding example exactly as shown (apart from the comments). An IIFE is a function expression that is called immediately after you define it. Inside the function, a new scope exists, preventing `tmp` from becoming global. Consult "Introducing a New Scope via an IIFE" on page 183 for details on IIFEs.

IIFE use case: inadvertent sharing via closures

Closures keep their connections to outer variables, which is sometimes not what you want:

```
var result = [];
for (var i=0; i < 5; i++) {
    result.push(function () { return i });  // (1)
}
console.log(result[1]()); // 5 (not 1)
console.log(result[3]()); // 5 (not 3)
```

The value returned in line (1) is always the current value of `i`, not the value it had when the function was created. After the loop is finished, `i` has the value 5, which is why all functions in the array return that value. If you want the function in line (1) to receive a snapshot of the current value of `i`, you can use an IIFE:

```
for (var i=0; i < 5; i++) {
    (function () {
        var i2 = i; // copy current i
        result.push(function () { return i2 });
    }());
}
```

Objects and Constructors

This section covers two basic object-oriented mechanisms of JavaScript: single objects and *constructors* (which are factories for objects, similar to classes in other languages).

Single Objects

Like all values, objects have properties. You could, in fact, consider an object to be a set of properties, where each property is a (key, value) pair. The key is a string, and the value is any JavaScript value.

In JavaScript, you can directly create plain objects, via *object literals*:

```
'use strict';
var jane = {
    name: 'Jane',

    describe: function () {
        return 'Person named '+this.name;
    }
};
```

The preceding object has the properties `name` and `describe`. You can read ("get") and write ("set") properties:

```
> jane.name  // get
'Jane'
> jane.name = 'John';  // set
> jane.newProperty = 'abc';  // property created automatically
```

Function-valued properties such as `describe` are called *methods*. They use `this` to refer to the object that was used to call them:

```
> jane.describe()  // call method
'Person named John'
> jane.name = 'Jane';
> jane.describe()
'Person named Jane'
```

The `in` operator checks whether a property exists:

```
> 'newProperty' in jane
true
> 'foo' in jane
false
```

If you read a property that does not exist, you get the value `undefined`. Hence, the previous two checks could also be performed like this:

```
> jane.newProperty !== undefined
true
> jane.foo !== undefined
false
```

The `delete` operator removes a property:

```
> delete jane.newProperty
true
```

```
> 'newProperty' in jane
false
```

Arbitrary Property Keys

A property key can be any string. So far, we have seen property keys in object literals and after the dot operator. However, you can use them that way only if they are identifiers (see "Identifiers and Variable Names" on page 6). If you want to use other strings as keys, you have to quote them in an object literal and use square brackets to get and set the property:

```
> var obj = { 'not an identifier': 123 };
> obj['not an identifier']
123
> obj['not an identifier'] = 456;
```

Square brackets also allow you to compute the key of a property:

```
> var obj = { hello: 'world' };
> var x = 'hello';

> obj[x]
'world'
> obj['hel'+'lo']
'world'
```

Extracting Methods

If you extract a method, it loses its connection with the object. On its own, the function is not a method anymore, and `this` has the value `undefined` (in strict mode).

As an example, let's go back to the earlier object `jane`:

```
'use strict';
var jane = {
    name: 'Jane',

    describe: function () {
        return 'Person named '+this.name;
    }
};
```

We want to extract the method `describe` from `jane`, put it into a variable `func`, and call it. However, that doesn't work:

```
> var func = jane.describe;
> func()
TypeError: Cannot read property 'name' of undefined
```

The solution is to use the method `bind()` that all functions have. It creates a new function whose `this` always has the given value:

```
> var func2 = jane.describe.bind(jane);
> func2()
'Person named Jane'
```

Functions Inside a Method

Every function has its own special variable `this`. This is inconvenient if you nest a function inside a method, because you can't access the method's `this` from the function. The following is an example where we call `forEach` with a function to iterate over an array:

```
var jane = {
    name: 'Jane',
    friends: [ 'Tarzan', 'Cheeta' ],
    logHiToFriends: function () {
        'use strict';
        this.friends.forEach(function (friend) {
            // `this` is undefined here
            console.log(this.name+' says hi to '+friend);
        });
    }
}
```

Calling `logHiToFriends` produces an error:

```
> jane.logHiToFriends()
TypeError: Cannot read property 'name' of undefined
```

Let's look at two ways of fixing this. First, we could store `this` in a different variable:

```
logHiToFriends: function () {
    'use strict';
    var that = this;
    this.friends.forEach(function (friend) {
        console.log(that.name+' says hi to '+friend);
    });
}
```

Or, `forEach` has a second parameter that allows you to provide a value for `this`:

```
logHiToFriends: function () {
    'use strict';
    this.friends.forEach(function (friend) {
        console.log(this.name+' says hi to '+friend);
    }, this);
}
```

Function expressions are often used as arguments in function calls in JavaScript. Always be careful when you refer to `this` from one of those function expressions.

Constructors: Factories for Objects

Until now, you may think that JavaScript objects are *only* maps from strings to values, a notion suggested by JavaScript's object literals, which look like the map/dictionary literals of other languages. However, JavaScript objects also support a feature that is truly object-oriented: inheritance. This section does not fully explain how JavaScript inheritance works, but it shows you a simple pattern to get you started. Consult Chapter 17 if you want to know more.

In addition to being "real" functions and methods, functions play another role in JavaScript: they become *constructors*—factories for objects—if invoked via the `new` operator. Constructors are thus a rough analog to classes in other languages. By convention, the names of constructors start with capital letters. For example:

```
// Set up instance data
function Point(x, y) {
    this.x = x;
    this.y = y;
}
// Methods
Point.prototype.dist = function () {
    return Math.sqrt(this.x*this.x + this.y*this.y);
};
```

We can see that a constructor has two parts. First, the function `Point` sets up the instance data. Second, the property `Point.prototype` contains an object with the methods. The former data is specific to each instance, while the latter data is shared among all instances.

To use `Point`, we invoke it via the `new` operator:

```
> var p = new Point(3, 5);
> p.x
3
> p.dist()
5.830951894845301
```

`p` is an instance of `Point`:

```
> p instanceof Point
true
```

Arrays

Arrays are sequences of elements that can be accessed via integer indices starting at zero.

Array Literals

Array literals are handy for creating arrays:

```
> var arr = [ 'a', 'b', 'c' ];
```

The preceding array has three elements: the strings `'a'`, `'b'`, and `'c'`. You can access them via integer indices:

```
> arr[0]
'a'
> arr[0] = 'x';
> arr
[ 'x', 'b', 'c' ]
```

The `length` property indicates how many elements an array has. You can use it to append elements and to remove elements:

```
> var arr = ['a', 'b'];
> arr.length
2

> arr[arr.length] = 'c';
> arr
[ 'a', 'b', 'c' ]
> arr.length
3

> arr.length = 1;
> arr
[ 'a' ]
```

The `in` operator works for arrays, too:

```
> var arr = [ 'a', 'b', 'c' ];
> 1 in arr // is there an element at index 1?
true
> 5 in arr // is there an element at index 5?
false
```

Note that arrays are objects and can thus have object properties:

```
> var arr = [];
> arr.foo = 123;
> arr.foo
123
```

Array Methods

Arrays have many methods (see "Array Prototype Methods" on page 286). Here are a few examples:

```
> var arr = [ 'a', 'b', 'c' ];

> arr.slice(1, 2)  // copy elements
[ 'b' ]
> arr.slice(1)
[ 'b', 'c' ]
```

```
> arr.push('x')  // append an element
4
> arr
[ 'a', 'b', 'c', 'x' ]

> arr.pop()  // remove last element
'x'
> arr
[ 'a', 'b', 'c' ]

> arr.shift()  // remove first element
'a'
> arr
[ 'b', 'c' ]

> arr.unshift('x')  // prepend an element
3
> arr
[ 'x', 'b', 'c' ]

> arr.indexOf('b')  // find the index of an element
1
> arr.indexOf('y')
-1

> arr.join('-')  // all elements in a single string
'x-b-c'
> arr.join('')
'xbc'
> arr.join()
'x,b,c'
```

Iterating over Arrays

There are several array methods for iterating over elements (see "Iteration (Nondestructive)" on page 291). The two most important ones are `forEach` and `map`.

`forEach` iterates over an array and hands the current element and its index to a function:

```
[ 'a', 'b', 'c' ].forEach(
    function (elem, index) {  // (1)
        console.log(index + '. ' + elem);
    });
```

The preceding code produces the following output:

```
0. a
1. b
2. c
```

Note that the function in line (1) is free to ignore arguments. It could, for example, only have the parameter `elem`.

`map` creates a new array by applying a function to each element of an existing array:

```
> [1,2,3].map(function (x) { return x*x })
[ 1, 4, 9 ]
```

Regular Expressions

JavaScript has built-in support for regular expressions (Chapter 19 refers to tutorials and explains in more detail how they work). They are delimited by slashes:

```
/^abc$/
/[A-Za-z0-9]+/
```

Method test(): Is There a Match?

```
> /^a+b+$/.test('aaab')
true
> /^a+b+$/.test('aaa')
false
```

Method exec(): Match and Capture Groups

```
> /a(b+)a/.exec('_abbba_aba_')
[ 'abbba', 'bbb' ]
```

The returned array contains the complete match at index 0, the capture of the first group at index 1, and so on. There is a way (discussed in "RegExp.prototype.exec: Capture Groups" on page 305) to invoke this method repeatedly to get all matches.

Method replace(): Search and Replace

```
> '<a> <bbb>'.replace(/<(.*?)>/g, '[$1]')
'[a] [bbb]'
```

The first parameter of `replace` must be a regular expression with a `/g` flag; otherwise, only the first occurrence is replaced. There is also a way (as discussed in "String.prototype.replace: Search and Replace" on page 307) to use a function to compute the replacement.

Math

`Math` (see Chapter 21) is an object with arithmetic functions. Here are some examples:

```
> Math.abs(-2)
2

> Math.pow(3, 2)  // 3 to the power of 2
9
```

```
> Math.max(2, -1, 5)
5

> Math.round(1.9)
2

> Math.PI  // pre-defined constant for π
3.141592653589793

> Math.cos(Math.PI)  // compute the cosine for 180°
-1
```

Other Functionality of the Standard Library

JavaScript's standard library is relatively spartan, but there are more things you can use:

`Date` *(Chapter 20)*
: A constructor for dates whose main functionality is parsing and creating date strings and accessing the components of a date (year, hour, etc.).

`JSON` *(Chapter 22)*
: An object with functions for parsing and generating JSON data.

`console.*` *methods (see "The Console API" on page 351)*
: These browser-specific methods are not part of the language proper, but some of them also work on Node.js.