Writing Sane JavaScript

A Comprehensive Introduction

Wyatt Allen

Contents

Preface

... An interesting mix ... capable of such beautiful dreams and such horrible nightmares.

— Carl Sagan, *Contact*

More good code has been written in languages denounced as bad than in languages proclaimed wonderful — much more.

— Bjarne Stroustrup, *The Design and Evolution of C++*

JavaScript is a truly remarkable language. It is ubiquitous: every computer with a browser installed comes with a full runtime. It's uncommonly powerful: its dynamic nature is more flexible than nearly any enterprise language, yet is used to do more and more amazing things. It's unstoppable: since its halfhearted genesis in 1995 it's never slowed its growing popularity.

Moreover, JavaScript is simultaneously beautiful and deeply flawed. The above quote from Carl Sagan frequently visits my thoughts as I work on code, although it was originally written about an alien being musing on the human race. There's no shortage of JavaScript code which is truly beautiful and elegant, frequently leaving the programmer with the impression of that ever-indefinable language quality: *expressiveness*.

But much more frequently, JavaScript will leave the programmer with a strong, sour impression of a desperate mess. Unreadable, unreliable, buggy and just plain bad code is rampant and difficult for the professional programmer to avoid encountering (or even to avoid writing, sometimes).

Bad JavaScript code doesn't necessarily mean bad programmers. It's a product of haphazard language design. For all of JavaScript's flexibility and power it can sometimes be maddeningly inconsistent and idiosyncratic, especially to the beginner. The aim of this book is to lend you, the reader complete knowledge of this unusual and rewarding language in order to overcome its obstacles. To borrow from Sagan — and wax just a bit starry-eyed — the aim of this book is to prevent the horrible nightmares, and promote the beautiful dreams.

Pedagogy

Learning to master JavaScript is an unusual challenge when compared to the task of learning other programming languages. No small part of this challenge is the fact that it's a very easy language to pick up and start using in a basic way without having formally learned it. Just about any programmer

with an acquaintance with the C family of languages could hack together a functioning event handler or popup in a webpage by trial and error or a handful of Google searches.

Because JavaScript is so easy to mess with, and is often used as a afterthought in webpage design, many treat it like a toy language and percieve formally learing it to be overkill. Unstoppably, however, JavaScript's role in the web has become more and more central. Scripts have grown to be complex and archetected – no longer simple, shallow event handlers but real, multi-tier applications. The casual knowledge a programmer may gain from simple script hacking becomes insufficient for constructing these new, complex, engineered systems.

This book was written to give you, the reader a proper, formal understanding of this capable language. For this reason, we do not begin with the easy aspects of the language; we will start by delving into the language's hard parts. It's not overkill. And at no point in this book will we treat JavaScript as a toy language.

Many other JavaScript books will begin with the easier, more casual aspects of the language, forcing readers to make a paradigm-shift partway through before they can really, formally learn it. Other books focus exclusively on the "good parts" of the language, advising readers to avoid using (or even considering) the poorly-designed parts which can cause trouble. Still other books are dry and meticulous tomes which, while useful reference volumes, afford little to the learner.

In this book, we will make an unflinching, complete tour through JavaScript — from start to finish. After reading, you, the reader will be able to say that you know or grok the language, warts and all. Moreover, the power is given to you the reader throughout these chapters to judge the language. I will always endeavor to advise you on the common snags, but, as a programmer, your judgement is respected and you choose what to make of it all.

Who is this Book For?

As stated above, in this book, the power is given to you the reader *as a programmer*. Consequently, you will need to be able to program. This is not an introduction to programming via JavaScript, but a comprehensive introduction to JavaScript programming. No prior JavaScript knowledge will be assumed.

As we shall see in JavaScript's history, its language design is influenced by C, Java and (remarkably) Self. If the reader is acquainted with any of these, it will be a great help. At least some understanding of Object-Oriented Programming (OOP) will be requisite, although JavaScript may force you to un-learn some of OOP's traditional forms. Acquaintance with Functional Progamming (FP) will be helpful for understanding much of JavaScript, but is not necessary.

Introduction

History

An essential part of understanding JavaScript and its significance is its unusual history.

TODO: Title, History, Significance, Beauty, State, How to read

Part I

The Language

Chapter 1

The Type System

1.1 Overview

The first hurdle in mastering JavaScript is a complete understanding of the type system. In this chapter we will have a first look at each of the fundamental datatypes in JavaScript. We examine the type system before presenting any code examples, so this chapter may seem a little dry. Let the reader be ensured, however, that this approach is to provide a rich context for when we finally do get to code.

JavaScript has a unique type system. In many ways it is Spartan. In many other ways it is rich and expressive. In many, many other ways it is inconsistent and treacherous. Throughout this book, you will be introduced to the type system from each of these three angles.

If you, the reader, gain a complete understanding of these types, it will either result in true enjoyment of a unique and powerful system, or well-informed hatred of a tangled mess. Read on to find out which it will be.

1.2 Datatypes at a Fundamental Level

At a fundamental level JavaScript has four basic datatypes. There are three primitive types which should be familiar to you as a programmer: `boolean`, `number` and `string`. The fourth datatype is `object`, which may be thought of as a composite type. The three primitives will behave largely in a similar fashion to what you're used to in other languages, but some of `object`'s nuances may be surprising.

Although we say there are only three primitive datatypes, JavaScript technically defines two more: `null` and `undefined`. `null` will probably look familiar to you from most any object-oriented language. `undefined` might be new to you. We'll explore these types in greater detail (and see why they're very different from the other primitives) once we've dug deeper into the type system. For now, just put them on your radar.

It's important to note that the line between primitive and object is not so clearly defined in JavaScript as it is in other languages. You'll find that nearly any value in JavaScript can *behave* like an object while remaining primitive. Some people interpret this to mean that *every value* in JavaScript is an object in disguise, but this is not the truth. We'll look at objects very closely soon enough.

1.2.1 The `boolean` Datatype

If we restrict ourselves to the fundamentals of the type system, the `boolean` datatype is as simple as can be. The ECMAScript standard explains it in a single sentence:

> The Boolean type represents a logical entity having two values, called **true** and **false**.

In practice, `true` and `false` are keywords which statically refer to the two possible boolean values (just as you find in most programming languages). These are considered "Boolean Literals."

1.2.2 The `number` Datatype

In JavaScript, there is no distinction between integral and floating-point numbers. Instead, we're given a single, unified type: `number`.

A number is stored as a signed 64-bit float according to the IEEE 754 standard. This means, under the covers, the numbers you deal with in JavaScript have the same encoding constraints as double-precision floats in Java or C.

It is worth noting that the simplicity you get by having a unified type for numbers comes at a cost of space-efficiency, performance and type-safety. On the other hand this will make some of your code cleaner because you no-longer need to coerce integers to floats or worry about loss of precision in mixed-mode division. In general, we will find that JavaScript tends to sacrifice efficiency/performance/safety in small ways in order to gain in simplicity.

A JavaScript number can be as large as $\pm 1.7976931348623157 \times 10^{308}$ or as small as $\pm 5 \times 10^{-324}$. The range of integral values which can be stored in a JavaScript `number` is $\pm 2^{53}$.

Numbers can be declared with number literals. Number literals are usually decimal (base-10) but can also be hexadecimal (base-16) using the traditional `0x` prefix. (In some implementations octal (base-8) number literals are allowed when the number is prefixed with a leading zero, although this is not standard.)

```
7

42

7.62

0xf0

0xff63ba
```

Number literals can also be written in exponential notation by suffixing a lower or upper-case letter "e" followed by a positive or negative radix.

```
6.022141e23

6.6260695729E-34
```

Negative number literals can be made in the usual way, by prefixing the literal with a minus sign. However, this isn't actually part of the literal itself, but is an application of the unary negation operator to the literal.

```
-532
```

1.2.3 The `string` Datatype

JavaScript provides `string` as a primitive type for dealing with text. All strings in JavaScript are encoded using either UTF-16 or UCS-2. (UCS-2 is a text encoding form which is extremely similar to UTF-16, but doesn't understand surrogate pair code points, which is almost never a problem.)

You just learned that JavaScript unifies integers and floats under numbers at a cost of performance and for the sake of simplicity. With strings, JavaScript makes a similar simplification: there is no `char` type. That is to say: in JavaScript any single character is of type `string` with a string-length of 1. As a result, when dealing with text, the `string` datatype is the only datatype you need.

String literals must be delimited by matching single or double quotes. In other words, strings may be delimited by either a matching start and end single quote or a matching start and end double quote but not mixed. The quote type which is not used to delimit may be used within the string literal without escaping.

```
"Hello, world."

'Hello, world with single quotes.'

"This apostrophe needn't be escaped."

'Double "quotes inside single" quotes.'
```

JavaScript `string`s support many of the escape sequences you're used to.

Sequence	Result	Code-Point
\'	A single quote.	\u0027
\"	A double quote.	\u0022
\\	A backslash.	\u005c
\u####	The unicode character represented by the hexadecimal digits ####.	
\t	A horizontal tab.	\u0009
\n	A newline.	\u000a

Several other escape sequences are available, but are not useful in most JavaScript scenarios. A full list of available escape sequences can be found in the appendix.

```
'This apostrophe\'s escape is necessary.'

"Text\n spanning\n multiple\n lines"
```

1.2.4 The `object` Datatype

As stated above, the `object` datatype may be thought of as a composite of other datatypes. In particular, an object is a collection of *named data properties*, similar to a dictionary you may be familiar with from other languages.

It is a remarkable feature of JavaScript that the `object` datatype also has a literal notation. The `object` is delimited by a pair of curly braces (i.e. "{" and "}"). Within these braces, any number of named data properties are specified by a name, followed by a colon (i.e. `:`) followed by the property's data value. These named data properties must be separated by commas.

As an example, the following is an object literal with two properties: a property named `foo` with the value of the `number 42` and another property named `bar` with the value of the `string "hello, world"`.

```
{ foo:42, bar:"hello, world" }
```

The order of the properties within the object literal is insignificant because the properties will be accessed by their names. For example, if this object were stored in the variable `x`, the `foo` property could be accessed via the familiar *dot-notation*.

```
x.foo
```

There is an additional *bracket-notation* for accessing properties by a string for their name. The following example accesses the same property as the example above, but with the alternate notation.

```
x["foo"]
```

In light of this, objects may feel significantly similar to a conventional dictionary collection; string-keys map to values. This bracket-notation makes two big things possible.

1. It allows you to access properties of an object dynamically, via a string variable. Later on, we'll loop over the properties of an object by enumerating it's property names into a key variable, and then access the property values themselves by using that key variable with the bracket-notation.
2. The bracket-notation makes possible the accessing of properties which have names with symbols that are not valid in normal property names.

   For example, a single property named `prop.erty` cannot be accessed using the dot notation because the dot in its name will be misinterpreted as accessing the `erty` subproperty of the `prop` property. Instead we can access this value in the following manner.

   ```
   x["prop.erty"]
   ```

   As you can see, because the dot is safely enclosed within the string, JavaScript can see that it is the name of a single property.

   The `object literal` syntax also allows illegal symbols in property names if you write the property name as a string. For example:

   ```
   { "name with spaces":"hello, world", "(){}[]":"another value" }
   ```

   You may notice that all of these bracket-notation examples use `string`s as keys, and you may wonder whether other datatypes — such as `number`s — may also be used for property names. In fact, any datatype can be used to specify a property, but its value will be converted to a `string` before the object is searched for a matching name.

   This means that the following two lines refer to the same property of the `object foo` — even though the datatypes in the brackets are completely different.

```
foo["42"]
foo[42]
```

So far, we've constructed objects using the object literal syntax, but all objects can also be created using another syntax which may be familiar to you: using the `new` operator and the object constructor. Par example:

```
new Object()
```

The reader's inner object-oriented programmer may immediately identify `Object` (with the initial capital) as the *class* of objects, but this is not the case. When we explore object-orientation in later chapters we'll see that `Object` is not the *class*, but merely the *constructor* for the `object` datatype.

1.3 Built-in Objects

It may seem odd that we discuss the four fundamental datatypes while not looking at arrays. Often, arrays are their own separate, special datatype, like in C. However, in JavaScript the *array datatype is implemented as an object*. More importantly, arrays are an example of a *built-in object*.

Built-in objects come from a set of classes which are provided for you by the JavaScript implementation. Often, these objects are implemented with *native code* and are part of the JavaScript runtime itself. (Put differently, they are JavaScript objects which are not written in JavaScript.) Consequently, they will behave a little differently from the objects that you create.

1.3.1 The `array` Datatype

The array datatype is a good first example of a built-in-object because it is so useful. In JavaScript, arrays are one-dimensional, untyped (potentially heterogenous) and dynamically expanding.

We can create an empty array with the array literal syntax. The array is delimited by a pair of open and close square braces (i.e. "`[`" and "`]`") and between them the elements are separated by commas.

```
[ "element 1", "element 2", "element 3" ]
```

Because arrays are really a kind of object, we can also build one with the constructor synatx:

```
new Array()
```

There are two additionnal ways to call the array constructor. It can be called with a single `number` argument to tell the runtime how many spaces to pre-allocate. Although a pre-allocated array will have the specified length, it will otherwise behave like an empty array as it does not have any elements for the allocated spaces.

```
new Array(5)

new Array(12)
```

The `Array` constructor can also be called as a variadic function accepting as arguments all of the elements with which pre-populate the array.

```
new Array("element 1", "element 2", "element 3");

new Array(1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987);
```

You may notice that there is a potential ambiguity between these two argument patterns: if the `array` constructor is called as a variadic with only a single argument. Will the constructor treat that argument as only element or as the given length of the new array?

The resolution to this issue is that the constructor tests whether that single argument is an integer, in which case it will be used as the array's new length but not as its element. In all other cases, (i.e. when the lone argument is not an integer) the argument will be used as the lone element of the array (which will then have a length of one).

Because of this, most programmers avoid the variadic version of the constructor and instead use the array literal, which achieves the same effect with less characters and no ambiguity.

The elements of any array can be indexed using the same bracket notation for accessing properties of objects. The difference is that instead of passing the `string` of the property name, we pass the inegral `number` of the element's zero-based index. For example, the following illustrates how to access the first and the twelfth element of an array named `foo`.

```
foo[0]

foo[11]
```

1.3.2 Other Built-in-Objects

JavaScript provides a whole host of useful built-in types in addition `array`. For example, JavaScript provides a complete regular expression datatype called `RegExp` which has a literal syntax of its own. JavaScript provieds a `Date` object for parsing and managing dates and times. JavaScript's DOM API (a Behemoth) provides a huge amount of built-in objects to represent the various components of a webpage.

In the next few chapters we'll learn that *functions themselves* are buit-in objects. This is a big concept, and we'll devote a lot of pages to them, but it means `function` is just another datatype. It can be stored to a variable or passed as an argument.

For the sake of this preliminary look at the type system, we'll have to leave off a complete investigation of these built-in object datatypes for later chapters. Each type has its own unique nuances, niceties and annoyances, so we'll have to take our time in order to be complete.

Chapter 2

Procedural Programming

After looking at JavaScript's basic types, now it's time to use them as building-blocks in procedural coding. We'll look at managing variables, using control structures and performing calculations. This foundation will segue into the use and writing of functions.

2.1 Core Syntax

JavaScript is syntactically based on the C family of languages. For example, statements are terminated by semicolons, whitespace is generally ignored and statement blocks are delimited by open and close curly braces. These features — as well as variable naming rules, code comments, and the much of the operator collection — are all borrowed from the syntax of C.

2.1.1 Comments

In particular, comments are written in two possible ways.

```
/* This
    is a
        multi-line
            comment
*/

// This is a single-line comment.
```

The Javadoc comment style is used frequently in JavaScript to document functions, but is not strictly part of the language. Moreover, it's a subset of the existing comment syntax. Similarly, C#-style XML comments are sometimes used in conjunction with Microsoft tools.

2.1.2 Optional Semicolons

Unlike C, the use of semicolons to terminate statements in JavaScript is optional. This language feature was initially concieved as a way to make the language more user-friendly or prettier. However, in combination with whitespace-insensitivity, it can lead to ambiguities, especially when placing multiple statements on the same line.

For example:

```
x = 4
y = x + 5

// Is equivalent to...

x = 4;
y = x + 5;

// However, with semicolons these statements could be on the same line.
// For example:

x = 4; y = x + 5;

// This would generate a syntax error if the first semicolon were omitted.
```

This is frequently called "optional semicolons" but is technically called "automatic semicolon insertion", meaning that JavaScript tries to guess where the semicolons would belong and adds them for you. In many cases JavaScript will guess wrong in such a way that the script will still parse, but will not behave in an expected manner. For this reason it's considered a best-practice to always terminate statements with semicolons — just as you would when writing in C or Java.

2.2 Declaring and using Variables

The most basic way to declare a variable in JavaScript is with the `var` keyword. As an example, a variable named `myVar` can be declared in the following way.

```
var myVar;
```

Even though the `myVar` is properly declared, it does not have a type. This declaration syntax may look a bit like a declaration from a statically-typed language like C or Java, except that, in this case, the keyword `var` is used in place of the type.

This is because JavaScript uses a *dynamic type system* — meaning, a variable can house a value of *any datatype*. The same variable that stores a `number` could be used to store an `object` later on. A variable that stores a `boolean`, could be used to store a `string` of any length.

2.2.1 `undefined`: The Universal Default Value

As a consequence of the dynamic system, when a variable has been declared but has not been assigned, it is automatically given a default value which spans *all types*: `undefined`.

In statically-typed languages, types usually have specific default values of their own. For example (depending on the compiler and architecture) an `int` in C may be automatically given a default value of zero (or perhaps the garbage value of whatever data was located at its memory space in the stack). But a `boolean` in Java may be automatically given the value `true` or `false`.

In JavaScript, since our variables do not have datatypes when they're declared, they must be automatically assigned a default value *which itself does not have a type*, and `undefined` is this value.

As stated in the chapter on the Type System, `undefined` is technically a primitive datatype of its own in order to keep it seperate form all the others. What's more, `undefined` is what's sometimes

called a *Unit Type* in Computer Type Theory, meaning a type that can only house one value. Indeed the value `undefined` *is its own datatype*, and the datatype `undefined` *is its only possible value*.

This may be confusing, but the point to take away from this is that when a variable's value is equal to `undefined`, you should interpret that it was never assigned a value. (Unless, for some reason, it was explicitly given the value `undefined`.)

We'll learn more about `undefined` (and how its subtly different from `null`) as we progress through the language.

2.2.2 Initialization Syntax

To give variables initial values (and to avoid `undefined`) you can specify a value in stride with declaration.

```
var theAnswer = 42;
```

Several variables can be declared and initialized at once using the *declaration list syntax*. Following the `var` keyword, variable names can be written together, seperated by commas, with optional initial values.

```
var
    message = "Welcome",
    prompt = "Please enter your name",
    givenName;
```

2.2.3 Variable Deletion

JavaScript provides a unary `delete` operator which can remove a variable and free its memory space — effectively reversing the effect of declaring it.

```
delete giantDataBuffer;
```

Most programmers don't find much use for the `delete` operator because the releasing of memory resources is usually handled automatically by scoping mechanisms in functions. However, larger JavaScript applications can sometimes develop memory leaks and manual deletion become necessary. `delete` can also be useful to overcome browser defects that generate memory leaks in the DOM API (such as Internet Explorer in some cases).

2.2.4 A Note on Scoping

You may encounter what is spoken of as an alternative way of defining variables. In most cases you could omit the `var` keyword and the value will be accessible and assignable and function exactly as if you did use the keyword.

```
theAnswer = 42;
```

In truth, this syntax is *not* totally equivalent to declaring a variable using `var`, but it differs in terms of the scope in which the variable is created. We'll have to take a look at functions and scoping before we appreciate this difference, so we'll revisit this topic later.

2.3 Operators

JavaScript provides many of the operators common in C or Java. Interestingly, JavaScript treats some things as operators which you may be used to as syntactical constructs in other languages. In JavaScript, the comma is an operator. So is `new`.

2.3.1 Precidence

The following table shows the operator precedence for a few of the common operators in JavaScript. For a complete precedence table see the appendix.

Precedence	Order	Assoicativity	Operator
1	Unary	RTL	`new`
1	Binary	LTR	`.` (dot-notation), `[]` (bracket-notation).
3	Unary		`++`, `--`
4	Unary		`!`, , `+`, `-`, `typeof`, `void`, `delete`
5	Binary	LTR	`*`, `/`, `%` (modulus)
6	Binary	LTR	`+`, `-`
8	Binary	LTR	`<`, `<=`, `>`, `>=`
9	Binary	LTR	`==`, `!=`, `===`, `!==`
13	Binary	LTR	`&&`
14	Binary	LTR	`||`
15	Ternary	RTL	`?:`
16	Binary	RTL	`=`, `+=`, `-=`, `*=`, `/=`, `%=`
17	Binary	LTR	`,` (comma)

These precidence rules mean that the expression `3*x+1` is equivalent to `(3*x)+1` and that `a || b && c || d` is equivalent to `a || ((b && c) || d)`.

JavaScript provides no mechanism for overloading operators.

2.4 Arithmetic

On the surface, arithmetic in JavaScript works very much like it does in other languages. However, there are some quirks, niceties and pitfalls to be aware of.

In C you need to worry about dividing integers, for example the C expression `1/2` will result in `0` because the arithmetic operation was in integer-mode and C did not know to promote the result into a float. However, in JavaScript, there is no integer type and consequently there is no integer-mode. The same expression in JavaScript would yield `0.5`.

The division operator in JavaScript is transfinite, meaning that a division by zero such as `1/0` results in the `Infinity` value rather than throw an exception or result in `NaN`. A negative infinity is also possible when the numerator is negative. On the other hand, the expression `0/0` is treated as nonsense, and will result in `NaN`. In the chapter on the `Number` datatype, we'll look closely at these special values.

2.5 Control Structures

JavaScript provides control structures to allow you to program loops and conditional blocks.

2.5.1 The `if` Control Structure

The `if` control structure is used to conditionally execute a statement or block of statements based on an expression.

For example:

```
if (true) {
    x = 4;
}

if (false) {
    x = 5;
}

// x has been set to 4.
```

It is also possible to specify `else` and `else if` conditions.

```
var n = 4;

if (n > 5) {
    x = 1;
}

else if (n <= 5 && n < 2) {
    x = 2;
}

else {
    x = 3;
}

// x has been set to 2 because it is between 2 and 5.
// The middle (else-if) condition was executed.
```

As you can see, this syntax is the same as in C and Java. In these cases one could omit the curly-braces for these blocks if they only contain one statement. For example, the following code is equivalent.

```
var n = 4;
if (n > 5)
    x = 1;
else if (n <= 5 && n < 2)
    x = 2;
else
    x = 3;
```

However, in almost all languages and coding style guides, this practice is considered bad. It is recommended that you always use curly-braces to delimit your blocks. Historically, for most languages and contexts, this recommendation is motivated by the fact that it is easier to read code with the braces, and it is harder to make certain errors. In JavaScript, however, there is the added motivation that some code optimizers work better when the braces are present.

A Note on Truthiness

The `if` control structure uses an expression to determine whether or not to execute a block. However, unlike in most languages, the expression *need not be boolean*. Moreover, the `if` control structure can use an expression of any type, and instead of checking whether it evaluates to `true` or `false`, it checks whether it is *truthy* or *falsy*.

Truthiness is a generalization of the truth value to non boolean datatypes. This feature of the JavaScript language is frequently maligned and often used in arguments that JavaScript is inconsistent. However, the rules of truthiness are quite simple and unambiguous. Usually when an expression evaluates to an unexpected truthiness value, it is the result of subtle datatype coersion that takes place before the truthiness is evaluated. We'll look at these confusing expressions later.

The rules of truthiness are as follows:

Expression Type	Truthiness
`undefined`	falsy
`null`	falsy
`boolean`	If the boolean is `true` then truthy, and if `false` then falsy.
`number`	If the number is $+0, -0$ or `NaN`, then falsy. Truthy otherwise.
`string`	If the string is empty (length is 0), then falsy. Truthy otherwise.
`object`	truthy

This means that the following `if` blocks never execute.

```
if (false)     { /* never    */ }
if (undefined) { /* ever     */ }
if (null)      { /* ever     */ }
if ("")        { /* ever     */ }
if (0)         { /* ever     */ }
if (0/0)       { /* executed */ }
```

And that the following blocks always execute.

```
if (true)    { /* executed */ }
if (1)       { /* each     */ }
if (1/0)     { /* and      */ }
if ("hello") { /* every    */ }
if ({})      { /* time     */ }
```

A Note on K&R Style

Another contentious aspect of coding style is whether to put opening curly-braces on the same line as the control structure definition or on a newline after it. These two styles are sometimes called K&R style and BSD style respectively. As an illustrative example:

```
if (true) {
    // This block is K&R style.
    // The open-curly brace is on the same line as the "if".

    // The K&R stands for Kerrighan and Richie because this style stems from their famous
    // book titled "The C Programming Language".
}

if (true)
{
    // This block is BSD style; sometimes called Allman style.
    // The open-curly brace is on a newline after the "if".

    // The BSD stands for the "Berkley Software Distribution" UNIX operating system.
    // Eric Allman pioneered this style and also wrote a great deal of BSD.
}
```

Both of these styles are valid in JavaScript and work equally well in every way *except one*. As we'll see in the chapter on functions, return statements with object literals *will not work correctly* when using BSD style. Consequently, for the sake of consistency, most JavaScript code is written using K&R style.

2.5.2 The `switch` Control Structure

TODO:

2.5.3 The `while` and do...while Control Structures

TODO:

2.5.4 The `for` and `for...in` Control Structures

TODO:

2.5.5 The `with` Control Structure

TODO:

2.5.6 The `throw` Statement

TODO:

2.5.7 The `try...catch` Control Structure

TODO:

2.6 Another Note on Scoping

No block scope... etc.

2.7 A Case Study in Procedural Programming

TODO:

Chapter 3

Functions

TODO:

3.1 Declaring Functions

TODO:

3.1.1 Declaring Functions

TODO:

3.1.2 The `return` Statment

TODO:

3.1.3 Creating Function Objects

TODO:

3.2 The `this` Keyword

TODO:

3.3 A Deeper look at Calling Functions

TODO:

3.4 Scoping

TODO:

3.5 Hoisting

TODO:

3.6 Functions as First-Class Objects

TODO:

3.7 A Case Study in Programming with Functions

TODO:

Chapter 4

Object-Oriented Programming

TODO:

4.1 Simple OOP

TODO:

4.1.1 Data Members and Methods

TODO:

4.1.2 Constructor Syntax

TODO:

4.2 Prototypal Inheritance

TODO:

4.2.1 The `prototype` Property

TODO:

4.2.2 The Role of `prototype` During Construction

TODO:

4.2.3 Derriving Classes via `prototype`

TODO:

4.3 Object Property Attributes

TODO:

4.3.1 The `value` Property Attribute

TODO:

4.3.2 The `writable` Property Attribute

TODO:

4.3.3 The `enumerable` Property Attribute

TODO:

4.3.4 The `configurable` Property Attribute

TODO:

4.4 A Case Study in Object-Oriented Prorgamming

TODO:

Part II

The Type System

Chapter 5

Truthy Values and the `boolean` Datatype

5.1 `true` and `false` Vs. Truthy and Falsy

5.2 The Truth of Expressions

5.3 A Case Study in Truth Values

Chapter 6

The `number` Datatype

6.1 Encoding and Range

6.2 Special Numbers

6.2.1 ± Zero

6.2.2 ± Infinity

6.2.3 `NaN`

6.2.4 The `Math` Object

6.3 `number`-Valued Expressions

6.4 A Case Study in the `number` Datatype

Chapter 7

The `string` Datatype

7.1 Encoding

7.2 String Methods

7.2.1 The `fromCharCode()` Method

7.2.2 The `charAt(pos)` and `charCodeAt(pos)` Methods

7.2.3 The `concat(string1, ..., stringN)` Method