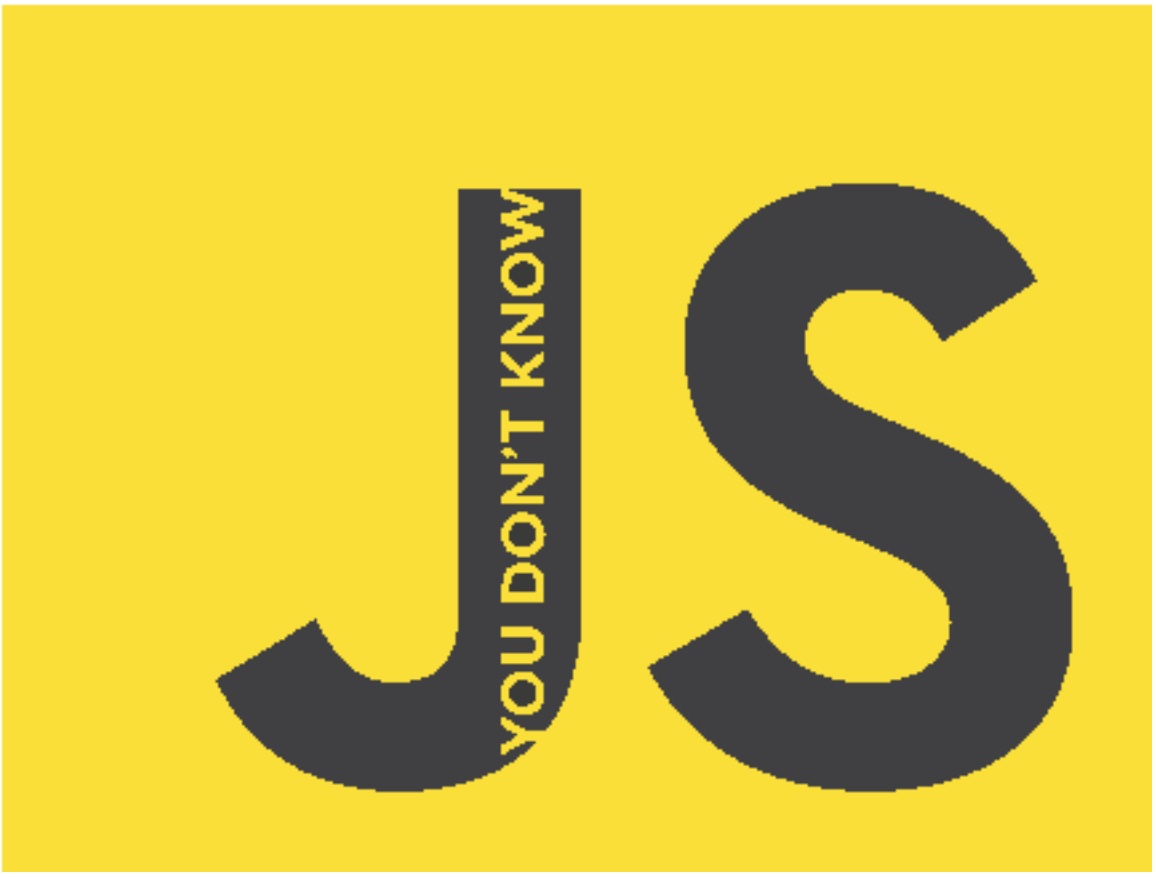
O'REILLY

"When you strive to comprehend your code, you create better software and become better at what you do. The code isn't just your job anymore, it's your craft. This is why I love Up & Going."
–JENN LUKAS, Frontend consultant

KYLE SIMPSON

UP & GOING



The *YOU DON'T KNOW JS* series includes:

- *Up & Going*
- *Scope & Closures*
- *this & Object Prototypes*
- *Types & Grammar*
- *Async & Performance*
- *ES6 & Beyond*

UP & GOING

It's easy to learn parts of JavaScript, but much harder to learn it completely—or even sufficiently—whether you're new to the language or have used it for years. With the "You Don't Know JS" book series, you'll get a more complete understanding of JavaScript, including trickier parts of the language that many experienced JavaScript programmers simply avoid.

The series' first book, ***Up & Going***, provides the necessary background for those of you with limited programming experience. By learning the basic building blocks of programming, as well as JavaScript's core mechanisms, you'll be prepared to dive into the other, more in-depth books in the series—and be well on your way toward true JavaScript.

With this book you will:

- Learn the essential programming building blocks, including operators, types, variables, conditionals, loops, and functions
- Become familiar with JavaScript's core mechanisms, such as values, function closures, *this*, and prototypes
- Get an overview of other books in the series—and learn why it's important to understand *all* parts of JavaScript

Kyle Simpson is an Open Web Evangelist from Austin, TX, who's passionate about all things JavaScript. He's an author, workshop trainer, tech speaker, and OSS contributor/leader.

JAVASCRIPT

US $4.99 CAN $6.99

ISBN: 978-1-491-92446-4

O'REILLY®

oreilly.com

YouDontKnowJS.com

Up & Going

Kyle Simpson

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

Up & Going
by Kyle Simpson

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://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Simon St.Laurent and Brian MacDonald
Production Editor: Kristen Brown
Copyeditor: Jasmine Kwityn
Proofreader: Amanda Kersey
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

April 2015: First Edition

Revision History for the First Edition
2015-03-17: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491924464* for release details.

978-1-491-92446-4

[LSI]

Table of Contents

Foreword

What was the last new thing you learned?

Perhaps it was a foreign language, like Italian or German. Or maybe it was a graphics editor, like Photoshop. Or a cooking technique or woodworking or an exercise routine. I want you to remember that feeling when you finally got it: the lightbulb moment. When things went from blurry to crystal clear, as you mastered the table saw or understood the difference between masculine and feminine nouns in French. How did it feel? Pretty amazing, right?

Now I want you to travel back a little bit further in your memory to right before you learned your new skill. How did *that* feel? Probably slightly intimidating and maybe a little bit frustrating, right? At one point, we all did not know the things that we know now, and that's totally OK; we all start somewhere. Learning new material is an exciting adventure, especially if you are looking to learn the subject efficiently.

I teach a lot of beginner coding classes. The students who take my classes have often tried teaching themselves subjects like HTML or JavaScript by reading blog posts or copying and pasting code, but they haven't been able to truly master the material that will allow them to code their desired outcome. And because they don't truly grasp the ins and outs of certain coding topics, they can't write powerful code or debug their own work because they don't really understand what is happening.

I always believe in teaching my classes the proper way, meaning I teach web standards, semantic markup, well-commented code, and other best practices. I cover the subject in a thorough manner to explain the hows and whys, without just tossing out code to copy

and paste. When you strive to comprehend your code, you create better work and become better at what you do. The code isn't just your *job* anymore, it's your *craft*. This is why I love *Up & Going*. Kyle takes us on a deep dive through syntax and terminology to give a great introduction to JavaScript without cutting corners. This book doesn't skim over the surface but really allows us to genuinely understand the concepts.

Because it's not enough to be able to duplicate jQuery snippets into your website, the same way it's not enough to learn how to open, close, and save a document in Photoshop. Sure, once I learned a few basics about the program, I could create and share a design I made. But without legitimately knowing the tools and what is behind them, how can I define a grid, or craft a legible type system, or optimize graphics for web use. The same goes for JavaScript. Without knowing how loops work, or how to define variables, or what scope is, we won't be writing the best code we can. We don't want to settle for anything less—this is, after all, our craft.

The more you are exposed to JavaScript, the clearer it becomes. Words like closures, objects, and methods might seem out of reach to you now, but this book will help those terms come into clarity. I want you to keep those two feelings of before and after you learn something in mind as you begin this book. It might seem daunting, but you've picked up this book because you are starting an awesome journey to hone your knowledge. *Up & Going* is the start of our path to understanding programming. Enjoy the lightbulb moments!

—Jenn Lukas (http://jennlukas.com, @jennlukas),
Frontend consultant

Preface

I'm sure you noticed, but "JS" in the series title is not an abbreviation for words used to curse about JavaScript, though cursing at the language's quirks is something we can probably all identify with!

From the earliest days of the Web, JavaScript has been a foundational technology that drives interactive experience around the content we consume. While flickering mouse trails and annoying pop-up prompts may be where JavaScript started, nearly two decades later, the technology and capability of JavaScript has grown many orders of magnitude, and few doubt its importance at the heart of the world's most widely available software platform: the Web.

But as a language, it has perpetually been a target for a great deal of criticism, owing partly to its heritage but even more to its design philosophy. Even the name evokes, as Brendan Eich once put it, "dumb kid brother" status next to its more mature older brother, Java. But the name is merely an accident of politics and marketing. The two languages are vastly different in many important ways. "JavaScript" is as related to "Java" as "Carnival" is to "Car."

Because JavaScript borrows concepts and syntax idioms from several languages, including proud C-style procedural roots as well as subtle, less obvious Scheme/Lisp-style functional roots, it is exceedingly approachable to a broad audience of developers, even those with little to no programming experience. The "Hello World" of JavaScript is so simple that the language is inviting and easy to get comfortable with in early exposure.

While JavaScript is perhaps one of the easiest languages to get up and running with, its eccentricities make solid mastery of the language a vastly less common occurrence than in many other lan-

guages. Where it takes a pretty in-depth knowledge of a language like C or C++ to write a full-scale program, full-scale production JavaScript can, and often does, barely scratch the surface of what the language can do.

Sophisticated concepts that are deeply rooted into the language tend instead to surface themselves in *seemingly* simplistic ways, such as passing around functions as callbacks, which encourages the JavaScript developer to just use the language as is and not worry too much about what's going on under the hood.

It is simultaneously a simple, easy-to-use language that has broad appeal, and a complex and nuanced collection of language mechanics that without careful study will elude *true understanding* even for the most seasoned of JavaScript developers.

Therein lies the paradox of JavaScript, the Achilles' heel of the language, the challenge we are presently addressing. Because JavaScript *can* be used without understanding, the understanding of the language is often never attained.

Mission

If at every point that you encounter a surprise or frustration in JavaScript, your response is to add it to the blacklist (as some are accustomed to doing), you soon will be relegated to a hollow shell of the richness of JavaScript.

While this subset has been famously dubbed "The Good Parts," I would implore you, dear reader, to instead consider it the "The Easy Parts," "The Safe Parts," or even "The Incomplete Parts."

This *You Don't Know JS* series offers a contrary challenge: learn and deeply understand *all* of JavaScript, even and especially "The Tough Parts."

Here, we address head-on the tendency of JS developers to learn just enough to get by, without ever forcing themselves to learn exactly how and why the language behaves the way it does. Furthermore, we eschew the common advice to retreat when the road gets rough.

I am not content, nor should you be, at stopping once something just works and not really knowing *why*. I gently challenge you to journey down that bumpy "road less traveled" and embrace all that JavaScript is and can do. With that knowledge, no technique, no framework, and no popular buzzword acronym of the week will be beyond your understanding.

These books each take on specific core parts of the language that are most commonly misunderstood or under-understood, and dive deep and exhaustively into them. You should come away from reading with a firm confidence in your understanding, not just of the theoretical, but the practical "what you need to know" bits.

The JavaScript you know right now is probably parts handed down to you by others who've been burned by incomplete understanding. *That* JavaScript is but a shadow of the true language. You don't really know JavaScript *yet*, but if you dig into this series, you will. Read on, my friends. JavaScript awaits you.

Review

JavaScript is awesome. It's easy to learn partially, and much harder to learn completely (or even *sufficiently*). When developers encounter confusion, they usually blame the language instead of their lack of understanding. These books aim to fix that, inspiring a strong appreciation for the language you can now, and *should*, deeply know.

Many of the examples in this book assume modern (and future-reaching) JavaScript engine environments, such as ES6. Some code may not work as described if run in older (pre-ES6) engines.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *http://bit.ly/ydkjs-up-going-code*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code

does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*You Don't Know JavaScript: Up & Going* by Kyle Simpson (O'Reilly). Copyright 2015 Getify Solutions, Inc., 978-1-491-92446-4."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/ydkjs_up-and-going*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

CHAPTER 1
Into Programming

Welcome to the *You Don't Know JS* (*YDKJS*) series.

Up & Going is an introduction to several basic concepts of programming—of course we lean toward JavaScript (often abbreviated JS) specifically—and how to approach and understand the rest of the titles in this series. Especially if you're just getting into programming and/or JavaScript, this book will briefly explore what you need to get *up and going*.

This book starts off explaining the basic principles of programming at a very high level. It's mostly intended if you are starting *YDKJS* with little to no prior programming experience, and are looking to these books to help get you started along a path to understanding programming through the lens of JavaScript.

Chapter 1 should be approached as a quick overview of the things you'll want to learn more about and practice to get *into programming*. There are also many other fantastic programming introduction resources that can help you dig into these topics further, and I encourage you to learn from them in addition to this chapter.

Once you feel comfortable with general programming basics, Chapter 2 will help guide you to a familiarity with JavaScript's flavor of programming. Chapter 2 introduces what JavaScript is about, but again, it's not a comprehensive guide—that's what the rest of the *YDKJS* books are for!

If you're already fairly comfortable with JavaScript, first check out Chapter 3 as a brief glimpse of what to expect from *YDKJS*, then jump right in!

Code

Let's start from the beginning.

A program, often referred to as *source code* or just *code*, is a set of special instructions to tell the computer what tasks to perform. Usually code is saved in a text file, although with JavaScript you can also type code directly into a developer console in a browser, which we'll cover shortly.

The rules for valid format and combinations of instructions is called a *computer language*, sometimes referred to as its *syntax*, much the same as English tells you how to spell words and how to create valid sentences using words and punctuation.

Statements

In a computer language, a group of words, numbers, and operators that performs a specific task is a *statement*. In JavaScript, a statement might look as follows:

```
a = b * 2;
```

The characters `a` and `b` are called *variables* (see "Variables" on page 14), which are like simple boxes you can store any of your stuff in. In programs, variables hold values (like the number `42`) to be used by the program. Think of them as symbolic placeholders for the values themselves.

By contrast, the `2` is just a value itself, called a *literal value*, because it stands alone without being stored in a variable.

The `=` and `*` characters are *operators* (see "Operators" on page 8)—they perform actions with the values and variables such as assignment and mathematic multiplication.

Most statements in JavaScript conclude with a semicolon (`;`) at the end.

The statement `a = b * 2;` tells the computer, roughly, to get the current value stored in the variable `b`, multiply that value by `2`, then store the result back into another variable we call `a`.

Programs are just collections of many such statements, which together describe all the steps that it takes to perform your program's purpose.

Expressions

Statements are made up of one or more *expressions*. An expression is any reference to a variable or value, or a set of variable(s) and value(s) combined with operators.

For example:

```
a = b * 2;
```

This statement has four expressions in it:

- `2` is a *literal value expression*.
- `b` is a *variable expression*, which means to retrieve its current value.
- `b * 2` is an *arithmetic expression*, which means to do the multiplication.
- `a = b * 2` is an *assignment expression*, which means to assign the result of the `b * 2` expression to the variable `a` (more on assignments later).

A general expression that stands alone is also called an *expression statement*, such as the following:

```
b * 2;
```

This flavor of expression statement is not very common or useful, as generally it wouldn't have any effect on the running of the program—it would retrieve the value of `b` and multiply it by `2`, but then wouldn't do anything with that result.

A more common expression statement is a *call expression* statement (see "Functions" on page 22), as the entire statement is the function call expression itself:

```
alert( a );
```

Executing a Program

How do those collections of programming statements tell the computer what to do? The program needs to be *executed*, also referred to as *running the program*.

Statements like `a = b * 2` are helpful for developers when reading and writing, but are not actually in a form the computer can directly understand. So a special utility on the computer (either an *interpreter* or a *compiler*) is used to translate the code you write into commands a computer can understand.

For some computer languages, this translation of commands is typically done from top to bottom, line by line, every time the program is run, which is usually called *interpreting* the code.

For other languages, the translation is done ahead of time, called *compiling* the code, so when the program *runs* later, what's running is actually the already compiled computer instructions ready to go.

It's typically asserted that JavaScript is *interpreted*, because your JavaScript source code is processed each time it's run. But that's not entirely accurate. The JavaScript engine actually *compiles* the program on the fly and then immediately runs the compiled code.

For more information on JavaScript compiling, see the first two chapters of the *Scope & Closures* title of this series.

Try It Yourself

This chapter is going to introduce each programming concept with simple snippets of code, all written in JavaScript (obviously!).

It cannot be emphasized enough: while you go through this chapter —and you may need to spend the time to go over it several times— you should practice each of these concepts by typing the code yourself. The easiest way to do that is to open up the developer tools console in your nearest browser (Firefox, Chrome, IE, etc.).

Typically, you can launch the developer console with a keyboard shortcut or from a menu item. For more detailed information about launching and using the console in your favorite browser, see "Mastering The Developer Tools Console".

To type multiple lines into the console at once, use `<shift> + <enter>` to move to the next new line. Once you hit `<enter>` by itself, the console will run everything you've just typed.

Let's get familiar with the process of running code in the console. First, I suggest opening up an empty tab in your browser. I prefer to do this by typing `about:blank` into the address bar. Then, make sure your developer console is open, as we just mentioned.

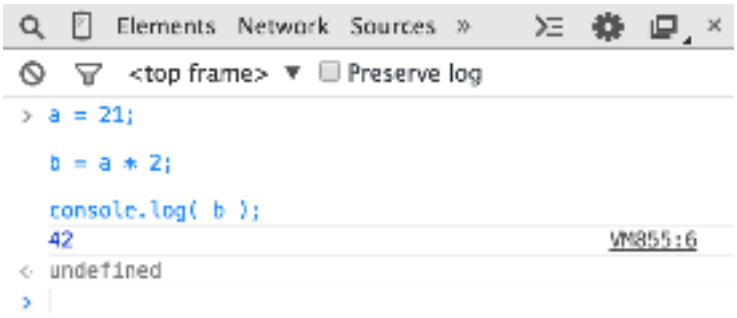
Now, type this code and see how it runs:

```
a = 21;

b = a * 2;

console.log( b );
```

Typing the preceding code into the console in Chrome should produce something like the following:



Go on, try it. The best way to learn programming is to start coding!

Output

In the previous code snippet, we used `console.log(..)`. Briefly, let's look at what that line of code is all about.

You may have guessed, but that's exactly how we print text (aka *output* to the user) in the developer console. There are two characteristics of that statement that we should explain.

First, the `log( b )` part is referred to as a function call (see "Functions" on page 22). What's happening is we're handing the `b` variable to that function, which asks it to take the value of `b` and print it to the console.

Second, the `console.` part is an object reference where the `log(..)` function is located. We cover objects and their properties in more detail in Chapter 2.

Another way of creating output that you can see is to run an `alert(..)` statement. For example:

```
alert( b );
```

If you run that, you'll notice that instead of printing the output to the console, it shows a pop-up "OK" box with the contents of the `b` variable. However, using `console.log(..)` is generally going to make learning about coding and running your programs in the console easier than using `alert(..)` because you can output many values at once without interrupting the browser interface.

For this book, we'll use `console.log(..)` for output.

Input

While we're discussing output, you may also wonder about *input* (i.e., receiving information from the user).

The most common way that happens is for the HTML page to show form elements (like text boxes) to a user that she can type into, and then use JS to read those values into your program's variables.

But there's an easier way to get input for simple learning and demonstration purposes such as what you'll be doing throughout this book. Use the `prompt(..)` function:

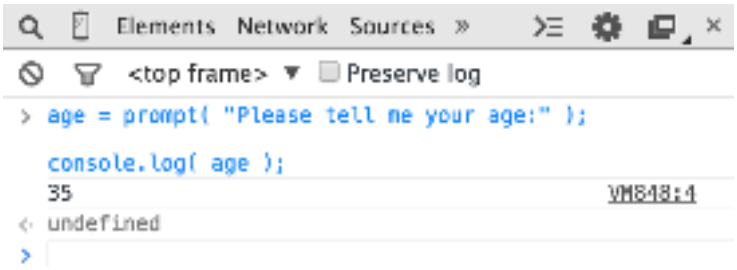
```
age = prompt( "Please tell me your age:" );

console.log( age );
```

As you may have guessed, the message you pass to `prompt(..)`—in this case, `"Please tell me your age:"`—is printed into the pop up.

This should look similar to the following:

Once you submit the input text by clicking "OK," you'll observe that the value you typed is stored in the `age` variable, which we then *output* with `console.log(..)`:



To keep things simple while we're learning basic programming concepts, the examples in this book will not require input. But now that you've seen how to use `prompt(..)`, if you want to challenge yourself, you can try to use input in your explorations of the examples.

Operators

Operators are how we perform actions on variables and values. We've already seen two JavaScript operators, the `=` and the `*`.

The `*` operator performs mathematic multiplication. Simple enough, right?

The `=` equals operator is used for *assignment*—we first calculate the value on the *right-hand side* (source value) of the `=` and then put it into the variable that we specify on the *left-hand side* (target variable).

This may seem like a strange reverse order to specify assignment. Instead of `a = 42`, some might prefer to flip the order so the source value is on the left and the target variable is on the right, like `42 -> a` (this is not valid JavaScript!). Unfortunately, the `a = 42` ordered form, and similar variations, is quite prevalent in modern programming languages. If it feels unnatural, just spend some time rehearsing that order in your mind to get accustomed to it.

Consider:

```
a = 2;
b = a + 1;
```

Here, we assign the `2` value to the `a` variable. Then, we get the value of the `a` variable (still `2`), add `1` to it resulting in the value `3`, then store that value in the `b` variable.

While not technically an operator, you'll need the keyword `var` in every program, as it's the primary way you *declare* (aka *create*) *var*iables (see "Variables" on page 14).

You should always declare the variable by name before you use it. But you only need to declare a variable once for each *scope* (see "Scope" on page 24); it can be used as many times after that as needed. For example:

```
var a = 20;

a = a + 1;
a = a * 2;
```

```
console.log( a );   // 42
```

Here are some of the most common operators in JavaScript:

Assignment
: `=`, as in `a = 2`.

Math
: `+` (addition), `-` (subtraction), `*` (multiplication), and `/` (division), as in `a * 3`.

Compound assignment
: `+=`, `-=`, `*=`, and `/=` are compound operators that combine a math operation with assignment, as in `a += 2` (same as `a = a + 2`).

Increment/decrement
: `++` (increment), `--` (decrement), as in `a++` (similar to `a = a + 1`).

Object property access
: `.` as in `console.log()`.

  Objects are values that hold other values at specific named locations called properties. `obj.a` means an object value called `obj` with a property of the name `a`. Properties can alternatively be accessed as `obj["a"]`. See Chapter 2.

Equality
: `==` (loose-equals), `===` (strict-equals), `!=` (loose not-equals), `!==` (strict not-equals), as in `a == b`.

  See "Values & Types" on page 10 and Chapter 2.

Comparison
: `<` (less than), `>` (greater than), `<=` (less than or loose-equals), `>=` (greater than or loose-equals), as in `a <= b`.

  See "Values & Types" on page 10 and Chapter 2.

Logical
: `&&` (and), `||` (or), as in `a || b` that selects either `a` *or* `b`.

  These operators are used to express compound conditionals (see "Conditionals" on page 18), like if either `a` *or* `b` is true.

For much more detail, and coverage of operators not mentioned here, see the Mozilla Developer Network (MDN)'s "Expressions and Operators".

Values & Types

If you ask an employee at a phone store how much a certain phone costs, and he says "ninety-nine, ninety-nine" (i.e., $99.99), he's giving you an actual numeric dollar figure that represents what you'll need to pay (plus taxes) to buy it. If you want to buy two of those phones, you can easily do the mental math to double that value to get $199.98 for your base cost.

If that same employee picks up another similar phone but says it's "free" (perhaps with air quotes), he's not giving you a number, but instead another kind of representation of your expected cost ($0.00) —the word "free."

When you later ask if the phone includes a charger, the answer can only be "yes" or "no."

In very similar ways, when you express values in a programs, you choose different representations for those values based on what you plan to do with them.

These different representations for values are called *types* in programming terminology. JavaScript has built-in types for each of these so-called *primitive* values:

- When you need to do math, you want a `number`.
- When you need to print a value on the screen, you need a `string` (one or more characters, words, or sentences).
- When you need to make a decision in your program, you need a `boolean` (`true` or `false`).

Values that are included directly in the source code are called *literals*. `string` literals are surrounded by double quotes (`"..."`) or single quotes (`'...'`)—the only difference is stylistic preference. `number` and `boolean` literals are just presented as is (e.g., `42`, `true`, etc.).