



# Modern JavaScript

## for the Impatient



Cay S. Horstmann



FREE SAMPLE CHAPTER

SHARE WITH OTHERS



# **Modern JavaScript for the Impatient**

*This page intentionally left blank*

# Modern JavaScript for the Impatient

**Cay S. Horstmann**

◆ Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town

Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City

São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at [corpsales@pearsoned.com](mailto:corpsales@pearsoned.com) or (800) 382-3419.

For government sales inquiries, please contact [governmentsales@pearsoned.com](mailto:governmentsales@pearsoned.com).

*Library of Congress Control Number: 2020934310*

Copyright © 2020 Pearson Education, Inc.

Cover illustration: Morphart Creation / Shutterstock

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit [www.pearson.com/permissions/](http://www.pearson.com/permissions/).

ISBN-13: 978-0-13-650214-2

ISBN-10: 0-13-650214-8

ScoutAutomatedPrintCode

*To Chi—the most patient person in my life.*

*This page intentionally left blank*

---

# Contents

Preface xv

About the Author xxi

I	VALUES AND VARIABLES	1	
1.1	Running JavaScript	1	
1.2	Types and the typeof Operator	5	
1.3	Comments	5	
1.4	Variable Declarations	6	
1.5	Identifiers	7	
1.6	Numbers	8	
1.7	Arithmetic Operators	9	
1.8	Boolean Values	12	
1.9	null and undefined	12	
1.10	String Literals	13	
1.11	Template Literals	15	
1.12	Objects	16	
1.13	Object Literal Syntax	17	
1.14	Arrays	18	

1.15	JSON	20	
1.16	Destructuring	21	
1.17	Advanced Destructuring	23	
1.17.1	More about Object Destructuring	23	
1.17.2	Rest Declarations	23	
1.17.3	Defaults	24	
	Exercises	24	
<b>2</b>	<b>CONTROL STRUCTURES</b>	<b>27</b>	
2.1	Expressions and Statements	27	
2.2	Semicolon Insertion	29	
2.3	Branches	31	
2.4	Boolishness	34	
2.5	Comparison and Equality Testing	34	
2.6	Mixed Comparisons	36	
2.7	Boolean Operators	37	
2.8	The switch Statement	39	
2.9	while and do Loops	40	
2.10	for Loops	41	
2.10.1	The Classic for Loop	41	
2.10.2	The for of Loop	42	
2.10.3	The for in Loop	43	
2.11	Breaking and Continuing	44	
2.12	Catching Exceptions	46	
	Exercises	47	
<b>3</b>	<b>FUNCTIONS AND FUNCTIONAL PROGRAMMING</b>	<b>51</b>	
3.1	Declaring Functions	51	
3.2	Higher-Order Functions	53	
3.3	Function Literals	54	
3.4	Arrow Functions	54	
3.5	Functional Array Processing	56	
3.6	Closures	57	

3.7	Hard Objects	59	
3.8	Strict Mode	61	
3.9	Testing Argument Types	62	
3.10	Supplying More or Fewer Arguments	63	
3.11	Default Arguments	64	
3.12	Rest Parameters and the Spread Operator	64	
3.13	Simulating Named Arguments with Destructuring	66	
3.14	Hoisting	67	
3.15	Throwing Exceptions	69	
3.16	Catching Exceptions	70	
3.17	The <code>finally</code> Clause	72	
	Exercises	73	
<b>4</b>	<b>OBJECT-ORIENTED PROGRAMMING</b>	<b>77</b>	
4.1	Methods	77	
4.2	Prototypes	78	
4.3	Constructors	81	
4.4	The Class Syntax	83	
4.5	Getters and Setters	85	
4.6	Instance Fields and Private Methods	85	
4.7	Static Methods and Fields	86	
4.8	Subclasses	87	
4.9	Overriding Methods	89	
4.10	Subclass Construction	90	
4.11	Class Expressions	91	
4.12	The <code>this</code> Reference	92	
	Exercises	95	
<b>5</b>	<b>NUMBERS AND DATES</b>	<b>99</b>	
5.1	Number Literals	99	
5.2	Number Formatting	100	
5.3	Number Parsing	101	
5.4	Number Functions and Constants	102	
5.5	Mathematical Functions and Constants	103	

5.6	Big Integers	105	
5.7	Constructing Dates	105	
5.8	Date Functions and Methods	109	
5.9	Date Formatting	110	
	Exercises	110	
<b>6</b>	<b>STRINGS AND REGULAR EXPRESSIONS</b>	<b>115</b>	
6.1	Converting between Strings and Code Point Sequences	115	
6.2	Substrings	116	
6.3	Other String Methods	118	
6.4	Tagged Template Literals	121	
6.5	Raw Template Literals	122	
6.6	Regular Expressions	123	
6.7	Regular Expression Literals	127	
6.8	Flags	127	
6.9	Regular Expressions and Unicode	129	
6.10	The Methods of the <code>RegExp</code> Class	130	
6.11	Groups	131	
6.12	String Methods with Regular Expressions	133	
6.13	More about Regex Replace	135	
6.14	Exotic Features	136	
	Exercises	137	
<b>7</b>	<b>ARRAYS AND COLLECTIONS</b>	<b>141</b>	
7.1	Constructing Arrays	141	
7.2	The <code>length</code> Property and Index Properties	143	
7.3	Deleting and Adding Elements	144	
7.4	Other Array Mutators	145	
7.5	Producing Elements	148	
7.6	Finding Elements	149	
7.7	Visiting All Elements	150	
7.8	Sparse Arrays	152	
7.9	Reduction	153	

7.10	Maps	156	
7.11	Sets	158	
7.12	Weak Maps and Sets	159	
7.13	Typed Arrays	160	
7.14	Array Buffers	163	
	Exercises	164	
<b>8</b>	<b>INTERNATIONALIZATION</b>	<b>167</b>	
8.1	The Locale Concept	167	
8.2	Specifying a Locale	168	
8.3	Formatting Numbers	171	
8.4	Localizing Dates and Times	172	
8.4.1	Formatting Date Objects	173	
8.4.2	Ranges	174	
8.4.3	Relative Time	175	
8.4.4	Formatting to Parts	175	
8.5	Collation	176	
8.6	Other Locale-Sensitive String Methods	177	
8.7	Plural Rules and Lists	178	
8.8	Miscellaneous Locale Features	180	
	Exercises	181	
<b>9</b>	<b>ASYNCHRONOUS PROGRAMMING</b>	<b>185</b>	
9.1	Concurrent Tasks in JavaScript	185	
9.2	Making Promises	188	
9.3	Immediately Settled Promises	191	
9.4	Obtaining Promise Results	191	
9.5	Promise Chaining	192	
9.6	Rejection Handling	194	
9.7	Executing Multiple Promises	196	
9.8	Racing Multiple Promises	197	
9.9	Async Functions	197	
9.10	Async Return Values	199	
9.11	Concurrent Await	201	

9.12	Exceptions in Async Functions	202
	Exercises	203

## IO MODULES 209



10.1	The Module Concept	209
10.2	ECMAScript Modules	210
10.3	Default Imports	211
10.4	Named Imports	212
10.5	Dynamic Imports	213
10.6	Exports	213
	10.6.1 Named Exports	213
	10.6.2 The Default Export	214
	10.6.3 Exports Are Variables	215
	10.6.4 Reexporting	216
10.7	Packaging Modules	217
	Exercises	218



## II METAPROGRAMMING 221



11.1	Symbols	221
11.2	Customization with Symbol Properties	223
	11.2.1 Customizing toString	223
	11.2.2 Controlling Type Conversion	224
	11.2.3 Species	225
11.3	Property Attributes	225
11.4	Enumerating Properties	228
11.5	Testing a Single Property	229
11.6	Protecting Objects	230
11.7	Creating or Updating Objects	231
11.8	Accessing and Updating the Prototype	231
11.9	Cloning Objects	232
11.10	Function Properties	235
11.11	Binding Arguments and Invoking Methods	236
11.12	Proxies	237
11.13	The Reflect Class	240

11.14 Proxy Invariants 242

Exercises 244

## I2 ITERATORS AND GENERATORS 249



12.1 Iterable Values 249

12.2 Implementing an Iterable 250

12.3 Closeable Iterators 252

12.4 Generators 254

12.5 Nested Yield 255

12.6 Generators as Consumers 257

12.7 Generators and Asynchronous Processing 259

12.8 Async Generators and Iterators 261

Exercises 263

## I3 AN INTRODUCTION TO TYPESCRIPT 269



13.1 Type Annotations 270

13.2 Running TypeScript 271

13.3 Type Terminology 273

13.4 Primitive Types 274

13.5 Composite Types 275

13.6 Type Inference 277

13.7 Subtypes 280

13.7.1 The Substitution Rule 280

13.7.2 Optional and Excess Properties 281

13.7.3 Array and Object Type Variance 282

13.8 Classes 284

13.8.1 Declaring Classes 284

13.8.2 The Instance Type of a Class 285

13.8.3 The Static Type of a Class 286

13.9 Structural Typing 287

13.10 Interfaces 288

13.11 Indexed Properties 290



13.12 Complex Function Parameters 291



13.12.1 Optional, Default, and Rest Parameters 291

	13.12.2	Destructuring Parameters	292
	13.12.3	Function Type Variance	293
	13.12.4	Overloads	295
13.13		Generic Programming	297 
	13.13.1	Generic Classes and Types	298
	13.13.2	Generic Functions	298
	13.13.3	Type Bounds	299
	13.13.4	Erasure	300
	13.13.5	Generic Type Variance	302
	13.13.6	Conditional Types	303
	13.13.7	Mapped Types	303
		Exercises	305
		Index	309

---

# Preface

Experienced programmers familiar with languages such as Java, C#, C, and C++ often find themselves in a position where they need to work with JavaScript. User interfaces are increasingly web-based, and JavaScript is the *lingua franca* of the web browser. The Electron framework extends this capability to rich client applications, and there are multiple solutions for producing mobile JavaScript apps. Increasingly, JavaScript is used on the server side.

Many years ago, JavaScript was conceived as a language for “programming in the small,” with a feature set that can be confusing and error-prone for larger programs. However, current standardization efforts and tool offerings go far beyond those humble beginnings.

Unfortunately, it is difficult to learn modern JavaScript without getting bogged down with obsolete JavaScript. Most books, courses, and blog posts are focused on transitioning from older JavaScript versions, which is not helpful for migrants from other languages.

That is the issue that this book addresses. I assume that you, the reader, are a competent programmer who understands branches and loops, functions, data structures, and the basics of object-oriented programming. I explain how to be productive with modern JavaScript, with only parenthetical remarks about obsolete features. You will learn how to put modern JavaScript to use, while avoiding pitfalls from the past.

JavaScript may not be perfect, but it has shown itself to be well-suited for user interface programming and many server-side tasks. As Jeff Atwood said presciently: “Any application that *can* be written in JavaScript, *will* eventually be written in JavaScript.”

Work through this book, and learn how to produce the next version of your application in modern JavaScript!

## Five Golden Rules

If you avoid a small number of “classic” features of JavaScript, you can greatly reduce the mental load of learning and using the language. These rules probably won’t make sense to you right now, but I list them here for your future reference, and to reassure you that they are few in number.

1. Declare variables with `let` or `const`, not `var`.
2. Use strict mode.
3. Know your types and avoid automatic type conversion.
4. Understand prototypes, but use modern syntax for classes, constructors, and methods.
5. Don’t use this outside constructors or methods.

And a meta-rule: *Avoid the Wat*—those snippets of confusing JavaScript code followed by a sarcastic “Wat?!” Some people find delight in demonstrating the supposed awfulness of JavaScript by dissecting obscure code. I have never learned anything useful from going down that rabbit hole. For example, what is the benefit of knowing that `2 * ['21']` is 42 but `2 + ['40']` is not, if the golden rule #3 tells you not to mess with type conversions? In general, when I run into a confusing situation, I ask myself how to avoid it, not how to explain its gory but useless details.

## The Learning Paths

When I wrote the book, I was trying to put information where you can find it when you need it. But that’s not necessarily the right place when you read the book for the first time. To help you customize your learning path, I tag each chapter with an icon that indicates its basic level. Sections that are more advanced than the chapter default get their own icons. You should absolutely skip those sections until you are ready for them.

Here are the icons:

The impatient rabbit denotes a **basic** topic that even the most impatient reader should not skip.



Alice indicates an **intermediate** topic that most programmers want to understand, but perhaps not on first reading.



The Cheshire cat points to an **advanced** topic that puts a smile on the face of a framework developer. Most application programmers can safely ignore these.



Finally, the mad hatter labels a **complex** and maddening topic, intended only for those with morbid curiosity.



## A Tour of the Book

In **Chapter 1**, we get going with the basic concepts of JavaScript: values and their types, variables, and most importantly, object literals. **Chapter 2** covers control flow. You can probably skim over it quickly if you are familiar with Java, C#, or C++. In **Chapter 3**, you will learn about functions and functional programming, which is very important in JavaScript. JavaScript has an object model that is very different from class-based programming languages. **Chapter 4** goes into detail, with a focus on modern syntax. **Chapters 5 and 6** cover the library classes that you will most often use for working with numbers, dates, strings, and regular expressions. These chapters are largely at the basic level, with a sprinkling of more advanced sections.

The next four chapters cover intermediate level topics. In **Chapter 7**, you will see how to work with arrays and the other collections that the standard JavaScript library offers. If your programs interact with users from around the world, you will want to pay special attention to the coverage of internationalization in **Chapter 8**. **Chapter 9** on asynchronous programming is very important for all programmers. Asynchronous programming used to be quite complex in JavaScript, but it has become much simpler with the introduction of promises and the `async` and `await` keywords. JavaScript now has a standard module system that is the topic of **Chapter 10**. You will see how to use modules that other programmers have written, and to produce your own.

**Chapter 11** covers metaprogramming at an advanced level. You will want to read this chapter if you need to create tools that analyze and transform arbitrary JavaScript objects. **Chapter 12** completes the coverage of JavaScript with another advanced topic: iterators and generators, which are powerful mechanisms for visiting and producing arbitrary sequences of values.

Finally, there is a bonus chapter, **Chapter 13**, on TypeScript. TypeScript is a superset of JavaScript that adds compile-time typing. It is not a part of standard JavaScript, but it is very popular. Read this chapter to decide whether you want to stick with plain JavaScript or use compile-time types.

The purpose of this book is to give you a firm grounding of the JavaScript *language* so that you can use it with confidence. However, you will need to turn elsewhere for the ever-changing landscape of tools and frameworks.

## Why I Wrote This Book

JavaScript is one of the most used programming languages on the planet. Like so many programmers, I knew a bit of *pidgin* JavaScript, and one day, I had to learn serious JavaScript in a hurry. But how?

There are any number of books that teach a little bit of JavaScript for casual web developers, but I already knew that much JavaScript. Flanagan's *Rhino book*<sup>1</sup> was great in 1996, but now it burdens readers with too many accidents from the past. Crockford's *JavaScript: The Good Parts*<sup>2</sup> was a wake-up call in 2008, but much of its message has been internalized in subsequent changes to the language. There are many books that bring old-style JavaScript programmers into the world of modern standards, but they assume an amount of "classic" JavaScript that was out of my comfort zone.

Of course, the web is awash in JavaScript-themed blogs of varying quality—some accurate but many with a tenuous grasp of the facts. I did not find it effective to scour the web for blogs and gauge their levels of truthfulness.

Oddly enough, I could not find a book for the millions of programmers who know Java or a similar language and who want to learn JavaScript as it exists today, without the historical baggage.

So I had to write it.

---

1. David Flanagan, *JavaScript: The Definitive Guide, Sixth Edition* (O'Reilly Media, 2011).  
2. Published by O'Reilly Media, 2008.

## Acknowledgments

I would like to once again thank my editor Greg Doench for supporting this project, as well as Dmitry Kirsanov and Alina Kirsanova for copyediting and typesetting the book. My special gratitude goes to the reviewers Gail Anderson, Tom Austin, Scott Davis, Scott Good, Kito Mann, Bob Nicholson, Ron Mak, and Henri Tremblay, for diligently spotting errors and providing thoughtful suggestions for improvements.

*Cay Horstmann*  
*Berlin*  
*March 2020*

*This page intentionally left blank*

---

# About the Author

**Cay S. Horstmann** is principal author of *Core Java™, Volumes I & II, Eleventh Edition* (Pearson, 2018), *Scala for the Impatient, Second Edition* (Addison-Wesley, 2016), and *Core Java SE 9 for the Impatient* (Addison-Wesley, 2017). Cay is a professor emeritus of computer science at San Jose State University, a Java Champion, and a frequent speaker at computer industry conferences.



# Functions and Functional Programming

## Topics in This Chapter

- 3.1 Declaring Functions — 51
- 3.2 Higher-Order Functions — 53
- 3.3 Function Literals — 54
- 3.4 Arrow Functions — 54
- 3.5 Functional Array Processing — 56
- 3.6 Closures — 57
-  3.7 Hard Objects — 59
- 3.8 Strict Mode — 61
- 3.9 Testing Argument Types — 62
- 3.10 Supplying More or Fewer Arguments — 63
- 3.11 Default Arguments — 64
- 3.12 Rest Parameters and the Spread Operator — 64
-  3.13 Simulating Named Arguments with Destructuring — 66
-  3.14 Hoisting — 67
- 3.15 Throwing Exceptions — 69
-  3.16 Catching Exceptions — 70
-  3.17 The finally Clause — 72
- Exercises — 73

---

# Chapter

# 3

In this chapter, you will learn how to write functions in JavaScript. JavaScript is a “functional” programming language. Functions are “first-class” values, just like numbers or strings. Functions can consume and produce other functions. Mastering a functional programming style is essential for working with modern JavaScript.

This chapter also covers the JavaScript parameter passing and scope rules, as well as the details of throwing and catching exceptions.

## 3.1 Declaring Functions

In JavaScript, you declare a function by providing

1. The name of the function
2. The names of the parameters
3. The body of the function, which computes and returns the function result

You do not specify the types of the function parameters or result. Here is an example:

```
function average(x, y) {  
    return (x + y) / 2  
}
```

The `return` statement yields the value that the function returns.

To call this function, simply pass the desired arguments:

```
let result = average(6, 7) // result is set to 6.5
```

What if you pass something other than a number? Whatever happens, happens. For example:

```
result = average('6', '7') // result is set to 33.5
```

When you pass strings, the `+` in the function body concatenates them. The resulting string `'67'` is converted to a number before the division by 2.

That looks rather casual to a Java, C#, or C++ programmer who is used to compile-time type checking. Indeed, if you mess up argument types, you only find out when something strange happens at runtime. On the flip side, you can write functions that work with arguments of multiple types, which can be convenient.

The `return` statement returns immediately, abandoning the remainder of the function. Consider this example—an `indexOf` function that computes the index of a value in an array:

```
function indexOf(arr, value) {  
  for (let i in arr) {  
    if (arr[i] === value) return i  
  }  
  return -1  
}
```

As soon as a match is found, the index is returned and the function terminates.

A function may choose not to specify a return value. If the function body exits without a `return` statement, or a `return` keyword isn't followed by an expression, the function returns the `undefined` value. This usually happens when a function is solely called for a side effect.



**TIP:** If a function sometimes returns a result, and sometimes you don't want to return anything, be explicit:

```
return undefined
```

---



**NOTE:** As mentioned in Chapter 2, a `return` statement must always have at least one token before the end of the line, to avoid automatic semicolon insertion. For example, if a function returns an object, put at least the opening brace on the same line:

```
return {
  average: (x + y) / 2,
  max: Math.max(x, y),
  . . .
}
```

## 3.2 Higher-Order Functions

JavaScript is a functional programming language. Functions are values that you can store in variables, pass as arguments, or return as function results.

For example, we can store the average function in a variable:

```
let f = average
```

Then you can call the function:

```
let result = f(6, 7)
```

When the expression `f(6, 7)` is executed, the contents of `f` is found to be a function. That function is called with arguments `6` and `7`.

We can later put another function into the variable `f`:

```
f = Math.max
```

Now when you compute `f(6, 7)`, the answer becomes `7`, the result of calling `Math.max` with the provided arguments.

Here is an example of passing a function as an argument. If `arr` is an array, the method call

```
arr.map(someFunction)
```

applies the provided function to all elements, and returns an array of the collected results (without modifying the original array). For example,

```
result = [0, 1, 2, 4].map(Math.sqrt)
```

sets `result` to

```
[0, 1, 1.4142135623730951, 2]
```

The `map` method is sometimes called a *higher-order function*: a function that consumes another function.

### 3.3 Function Literals

Let us continue the example of the preceding section. Suppose we want to multiply all array elements by 10. Of course, we can write a function

```
function multiplyBy10(x) { return x * 10 }
```

Now we can call:

```
result = [0, 1, 2, 4].map(multiplyBy10)
```

But it seems a waste to declare a new function just to use it once.

It is better to use a *function literal*. JavaScript has two syntactical variants. Here is the first one:

```
result = [0, 1, 2, 4].map(function (x) { return 10 * x })
```

The syntax is straightforward. You use the same function syntax as before, but now you omit the name. The function literal is a value that denotes the function with the specified action. That value is passed to the `map` method.

By itself, the function literal doesn't have a name, just like the array literal `[0, 1, 2, 4]` doesn't have a name. If you want to give the function a name, do what you always do when you want to give something a name—store it in a variable:

```
const average = function (x, y) { return (x + y) / 2 }
```



**TIP:** Think of anonymous function literals as the “normal” case. A named function is a shorthand for declaring a function literal and then giving it a name.

---

### 3.4 Arrow Functions

In the preceding section, you saw how to declare function literals with the `function` keyword. There is a second, more concise form that uses the `=>` operator, usually called “arrow”:

```
const average = (x, y) => (x + y) / 2
```

You provide the parameter variables to the left of the arrow and the return value to the right.

If there is a single parameter, you don't need to enclose it in parentheses:

```
const multiplyBy10 = x => x * 10
```

If the function has no parameters, use an empty set of parentheses:

```
const dieToss = () => Math.trunc(Math.random() * 6) + 1
```

Note that `dieToss` is a function, not a number. Each time you call `dieToss()`, you get a random integer between 1 and 6.

If an arrow function is more complex, place its body inside a block statement. Use the `return` keyword to return a value out of the block:

```
const indexOf = (arr, value) => {  
  for (let i in arr) {  
    if (arr[i] === value) return i  
  }  
  return -1  
}
```



**TIP:** The `=>` token must be on the same line as the parameters:

```
const average = (x, y) => // OK  
  (x + y) / 2  
const distance = (x, y) // Error  
  => Math.abs(x - y)
```

If you write an arrow function on more than one line, it is clearer to use braces:

```
const average = (x, y) => {  
  return (x + y) / 2  
}
```



**CAUTION:** If an arrow function does nothing but returns an object literal, then you must enclose the object in parentheses:

```
const stats = (x, y) => ({  
  average: (x + y) / 2,  
  distance: Math.abs(x - y)  
})
```

Otherwise, the braces would be parsed as a block.



**TIP:** As you will see in Chapter 4, arrow functions have more regular behavior than functions declared with the `function` keyword. Many JavaScript programmers prefer to use the arrow syntax for anonymous and nested functions. Some programmers use the arrow syntax for all functions, while others prefer to declare top-level functions with `function`. This is purely a matter of taste.

## 3.5 Functional Array Processing

Instead of iterating over an array with a `for of` or `for in` loop, you can use the `forEach` method. Pass a function that processes the elements and index values:

```
arr.forEach((element, index) => { console.log(`${index}: ${element}`) })
```

The function is called for each array element, in increasing index order.

If you only care about the elements, you can pass a function with one parameter:

```
arr.forEach(element => { console.log(`${element}`) })
```

The `forEach` method will call this function with both the element and the index, but in this example, the index is ignored.

The `forEach` method doesn't produce a result. Instead, the function that you pass to it must have some side effect—printing a value or making an assignment. It is even better if you can avoid side effects altogether and use methods such as `map` and `filter` that transform arrays into their desired form.

In Section 3.2, “Higher-Order Functions” (page 53), you saw the `map` method that transforms an array, applying a function to each element. Here is a practical example. Suppose you want to build an HTML list of items in an array. You can first enclose each of the items in a `li` element:

```
const enclose = (tag, contents) => `<${tag}>${contents}</${tag}>`  
const listItems = items.map(i => enclose('li', i))
```

Actually, it is safer to first escape `&` and `<` characters in the items. Let's suppose we have an `htmlEscape` function for this purpose. (You will find an implementation in the book's companion code.) Then we can first transform the items to make them safe, and then enclose them:

```
const listItems = items  
  .map(htmlEscape)  
  .map(i => enclose('li', i))
```

Now the result is an array of `li` elements. Next, we concatenate all strings with the `Array.join` method (see Chapter 7), and enclose the resulting string in a `ul` element:

```
const list = enclose('ul',  
  items  
  .map(htmlEscape)  
  .map(i => enclose('li', i))  
  .join(''))
```

Another useful array method is `filter`. It receives a *predicate* function—a function that returns a Boolean (or Boolish) value. The result is an array of all elements

that fulfill the predicate. Continuing the preceding example, we don't want to include empty strings in the list. We can remove them like this:

```
const list = enclose('ul',
  items
  .filter(i => i.trim() !== '')
  .map(htmlEscape)
  .map(i => enclose('li', i))
  .join(''))
```

This processing pipeline is a good example of a high-level “what, not how” style of programming. What do we want? Throw away empty strings, escape HTML, enclose items in `li` elements, and join them. How is this done? Ultimately, by a sequence of loops and branches, but that is an implementation detail.

## 3.6 Closures

The `setTimeout` function takes two arguments: a function to execute later, when a timeout has elapsed, and the duration of the timeout in milliseconds. For example, this call says “Goodbye” in ten seconds:

```
setTimeout(() => console.log('Goodbye'), 10000)
```

Let's make this more flexible:

```
const sayLater = (text, when) => {
  let task = () => console.log(text)
  setTimeout(task, when)
}
```

Now we can call:

```
sayLater('Hello', 1000)
sayLater('Goodbye', 10000)
```

Look at the variable `text` inside the arrow function `() => console.log(text)`. If you think about it, something nonobvious is going on. The code of the arrow function runs long after the call to `sayLater` has returned. How does the `text` variable stay around? And how can it be first `'Hello'` and then `'Goodbye'`?

To understand what is happening, we need to refine our understanding of a function. A function has three ingredients:

1. A block of code
2. Parameters
3. The free variables—that is, the variables that are used in the code but are not declared as parameters or local variables

A function with free variables is called a *closure*.

In our example, `text` is a free variable of the arrow function. The data structure representing the closure stores a reference to the variable when the function is created. We say that the variable is *captured*. That way, its value is available when the function is later called.

In fact, the arrow function `() => console.log(text)` also captures a second variable, namely `console`.

But how does `text` get to have two different values? Let's do this in slow motion. The first call to `sayLater` creates a closure that captures the `text` parameter variable holding the value `'Hello'`. When the `sayLater` method exits, that variable does not go away because it is still used by the closure. When `sayLater` is called again, a second closure is created that captures a different `text` parameter variable, this time holding `'Goodbye'`.

In JavaScript, a captured variable is a reference to another variable, not its current value. If you change the contents of the captured variable, the change is visible in the closure. Consider this case:

```
let text = 'Goodbye'  
setTimeout(() => console.log(text), 10000)  
text = 'Hello'
```

In ten seconds, the string `'Hello'` is printed, even though `text` contained `'Goodbye'` when the closure was created.



**NOTE:** The lambda expressions and inner classes in Java can also capture variables from enclosing scopes. But in Java, a captured local variable must be effectively `final`—that is, its value can never change.

Capturing mutable variables complicates the implementation of closures in JavaScript. A JavaScript closure remembers not just the initial value but the location of the captured variable. And the captured variable is kept alive for as long as the closure exists—even if it is a local variable of a terminated method.

---

The fundamental idea of a closure is very simple: A free variable inside a function means exactly what it means outside. However, the consequences are profound. It is very useful to capture variables and have them accessible indefinitely. The next section provides a dramatic illustration, by implementing objects and methods entirely with closures.

## 3.7 Hard Objects



Let's say we want to implement bank account objects. Each bank account has a balance. We can deposit and withdraw money.

We want to keep the object state private, so that nobody can modify it except through methods that we provide. Here is an outline of a factory function:

```
const createAccount = () => {  
  . . .  
  return {  
    deposit: amount => { . . . },  
    withdraw: amount => { . . . },  
    getBalance: () => . . .  
  }  
}
```

Then we can construct as many accounts as we like:

```
const harrysAccount = createAccount()  
const sallysAccount = createAccount()  
sallysAccount.deposit(500)
```

Note that an account object contains only methods, not data. After all, if we added the balance to the account object, anyone could modify it. There are no “private” properties in JavaScript.

Where do we store the data? It's simple—as local variables in the factory function:

```
const createAccount = () => {  
  let balance = 0  
  return {  
    . . .  
  }  
}
```

We capture the local data in the methods:

```
const createAccount = () => {  
  . . .  
  return {  
    deposit: amount => {  
      balance += amount  
    },  
    withdraw: amount => {  
      if (balance >= amount)  
        balance -= amount  
    },  
    getBalance: () => balance  
  }  
}
```

Each account has *its own* captured balance variable, namely the one that was created when the factory function was called.

You can provide parameters in the factory function:

```
const createAccount = (initialBalance) => {
  let balance = initialBalance + 10 // Bonus for opening the account
  return {
    . . .
  }
}
```

You can even capture the parameter variable instead of a local variable:

```
const createAccount = (balance) => {
  balance += 10 // Bonus for opening the account
  return {
    deposit: amount => {
      balance += amount
    },
    . . .
  }
}
```

At first glance, this looks like an odd way of producing objects. But these objects have two significant advantages. The state, consisting solely of captured local variables of the factory function, is automatically encapsulated. And you avoid the `this` parameter, which, as you will see in Chapter 4, is not straightforward in JavaScript.

This technique is sometimes called the “closure pattern” or “factory class pattern,” but I like the term that Douglas Crockford uses in his book *How JavaScript Works*. He calls them “hard objects.”



**NOTE:** To further harden the object, you can use the `Object.freeze` method that yields an object whose properties cannot be modified or removed, and to which no new properties can be added.

```
const createAccount = (balance) => {
  return Object.freeze({
    deposit: amount => {
      balance += amount
    },
    . . .
  })
}
```

---

## 3.8 Strict Mode

As you have seen, JavaScript has its share of unusual features, some of which have proven to be poorly suited for large-scale software development. *Strict mode* outlaws some of these features. You should always use strict mode.

To enable strict mode, place the line

```
'use strict'
```

as the first non-comment line in your file. (Double quotes instead of single quotes are OK, as is a semicolon.)

If you want to force strict mode in the Node.js REPL, start it with

```
node --use-strict
```



**NOTE:** In a browser console, you need to prefix each line that you want to execute in strict mode with `'use strict'`; or `'use strict'` followed by Shift+Enter. That is not very convenient.

---

You can apply strict mode to individual functions:

```
function strictInASeaOfSloppy() {  
  'use strict'  
  . . .  
}
```

There is no good reason to use per-function strict mode with modern code. Apply strict mode to the entire file.

Finally, strict mode is enabled inside classes (see Chapter 4) and ECMAScript modules (see Chapter 10).

For the record, here are the key features of strict mode:

- Assigning a value to a previously undeclared variable is an error and does not create a global variable. You must use `let`, `const`, or `var` for all variable declarations.
- You cannot assign a new value to a read-only global property such as `NaN` or `undefined`. (Sadly, you can still declare local variables that shadow them.)
- Functions can only be declared at the top level of a script or function, not in a nested block.
- The `delete` operator cannot be applied to “unqualified identifiers.” For example, `delete parseInt` is a syntax error. Trying to `delete` a property that is not “configurable” (such as `delete 'Hello'.length`) causes a runtime error.

- You cannot have duplicate function parameters (function `average(x, x)`). Of course, you never wanted those, but they are legal in the “sloppy” (non-strict) mode.
- You cannot use octal literals with a `0` prefix: `010` is a syntax error, not an octal 10 (which is 8 in decimal). If you want octal, use `0o10`.
- The `with` statement (which is not discussed in this book) is prohibited.



**NOTE:** In strict mode, reading the value of an undeclared variable throws a `ReferenceError`. If you need to find out whether a variable has been declared (and initialized), you can't check

```
possiblyUndefinedVariable !== undefined
```

Instead, use the condition

```
typeof possiblyUndefinedVariable !== 'undefined'
```

## 3.9 Testing Argument Types

In JavaScript, you do not specify the types of function arguments. Therefore, you can allow callers to supply an argument of one type or another, and handle that argument according to its actual type.

As a somewhat contrived example, the `average` function may accept either numbers or arrays.

```
const average = (x, y) => {
  let sum = 0
  let n = 0
  if (Array.isArray(x)) {
    for (const value of x) { sum += value; n++ }
  } else {
    sum = x; n = 1
  }
  if (Array.isArray(y)) {
    for (const value of y) { sum += value }
  } else {
    sum += y; n++
  }
  return n === 0 ? 0 : sum / n
}
```

Now you can call:

```
result = average(1, 2)
result = average([1, 2, 3], 4)
result = average(1, [2, 3, 4])
result = average([1, 2], [3, 4, 5])
```

Table 3-1 shows how to test whether an argument *x* conforms to a given type.

**Table 3-1** Type Tests

Type	Test	Notes
String	<code>typeof x === 'string'    x instanceof String</code>	<i>x</i> might be constructed as <code>new String(. . .)</code>
Regular expression	<code>x instanceof RegExp</code>	
Number	<code>typeof x === 'number'    x instanceof Number</code>	<i>x</i> might be constructed as <code>new Number(. . .)</code>
Anything that can be converted to a number	<code>typeof +x === 'number'</code>	Obtain the numeric value as <code>+x</code>
Array	<code>Array.isArray(x)</code>	
Function	<code>typeof x === 'function'</code>	



**NOTE:** Some programmers write functions that turn any argument values into numbers, such as

```
const average = (x, y) => {
  return (+x + +y) / 2
}
```

Then one can call

```
average('3', [4])
```

Is that degree of flexibility useful, harmless, or a harbinger of trouble? I don't recommend it.

## 3.10 Supplying More or Fewer Arguments

Suppose a function is declared with a particular number of parameters, for example:

```
const average = (x, y) => (x + y) / 2
```

It appears as if you must supply two arguments when you call the function. However, that is not the JavaScript way. You can call the function with more arguments—they are silently ignored:

```
let result = average(3, 4, 5) // 3.5—the last argument is ignored
```

Conversely, if you supply fewer arguments, then the missing ones are set to undefined. For example, `average(3)` is  $(3 + \text{undefined}) / 2$ , or NaN. If you want to support that call with a meaningful result, you can:

```
const average = (x, y) => y === undefined ? x : (x + y) / 2
```

## 3.11 Default Arguments

In the preceding section, you saw how to implement a function that is called with fewer arguments than parameters. Instead of manually checking for undefined argument values, you can provide default arguments in the function declaration. After the parameter, put an = and an expression for the default—that is, the value that should be used if no argument was passed.

Here is another way of making the average function work with one argument:

```
const average = (x, y = x) => (x + y) / 2
```

If you call `average(3)`, then `y` is set to `x`—that is, 3—and the correct return value is computed.

You can provide multiple default values:

```
const average = (x = 0, y = x) => (x + y) / 2
```

Now `average()` returns zero.

You can even provide a default for the first parameter and not the others:

```
const average = (x = 0, y) => y === undefined ? x : (x + y) / 2
```

If no argument (or an explicit `undefined`) is supplied, the parameter is set to the default or, if none is provided, to `undefined`:

```
average(3) // average(3, undefined)
average() // average(0, undefined)
average(undefined, 3) // average(0, 3)
```

## 3.12 Rest Parameters and the Spread Operator

As you have seen, you can call a JavaScript function with any number of arguments. To process them all, declare the last parameter of the function as a “rest” parameter by prefixing it with the `...` token:

```
const average = (first = 0, ...following) => {
  let sum = first
  for (const value of following) { sum += value }
  return sum / (1 + following.length)
}
```

When the function is called, the following parameter is an array that holds all arguments that have not been used to initialize the preceding parameters. For example, consider the call:

```
average(1, 7, 2, 9)
```

Then first is 1 and following is the array [7, 2, 9].

Many functions and methods accept variable arguments. For example, the `Math.max` method yields the largest of its arguments, no matter how many:

```
let result = Math.max(3, 1, 4, 1, 5, 9, 2, 6) // Sets result to 9
```

What if the values are already in an array?

```
let numbers = [1, 7, 2, 9]
result = Math.max(numbers) // Yields NaN
```

That doesn't work. The `Math.max` method receives an array with one element—the array [1, 7, 2, 9].

Instead, use the “spread” operator—the `...` token placed before an array *argument*:

```
result = Math.max(...numbers) // Yields 9
```

The spread operator spreads out the elements as if they had been provided separately in the call.



**NOTE:** Even though the spread operator and rest declaration look the same, their actions are the exact opposites of each other.

First, note that the spread operator is used with an argument, and the rest syntax applies to a variable declaration.

```
Math.max(...numbers) // Spread operator—argument in function call
const max = (...values) => { /* body */ }
// Rest declaration of parameter variable
```

The spread operator turns an array (or, in fact, any iterable) into a sequence of values. The rest declaration causes a sequence of values to be placed into an array.

---

Note that you can use the spread operator even if the function that you call doesn't have any rest parameters. For example, consider the `average` function of the preceding section that has two parameters. If you call

```
result = average(...numbers)
```

then all elements of `numbers` are passed as arguments to the function. The function uses the first two arguments and ignores the others.



**NOTE:** You can also use the spread operator in an array initializer:

```
let moreNumbers = [1, 2, 3, ...numbers] // Spread operator
```

Don't confuse this with the rest declaration used with destructuring. The rest declaration applies to a variable:

```
let [first, ...following] = numbers // Rest declaration
```



**TIP:** Since strings are iterable, you can use the spread operator with a string:

```
let greeting = 'Hello 🐱'
let characters = [...greeting]
```

The characters array contains the strings 'H', 'e', 'l', 'l', 'o', ' ', and '🐱'.

The syntax for default arguments and rest parameters are equally applicable to the function syntax:

```
function average(first = 0, ...following) { . . . }
```



### 3.13 Simulating Named Arguments with Destructuring

JavaScript has no “named argument” feature where you provide the parameter names in the call. But you can easily simulate named arguments by passing an object literal:

```
const result = mkString(values, { leftDelimiter: '(', rightDelimiter: ')' })
```

That is easy enough for the caller of the function. Now, let's turn to the function implementation. You can look up the object properties and supply defaults for missing values.

```
const mkString = (array, config) => {
  let separator = config.separator === undefined ? ',' : config.separator
  . . .
}
```

However, that is tedious. It is easier to use destructured parameters with defaults. (See Chapter 1 for the destructuring syntax.)

```
const mkString = (array, {
  separator = ',',
  leftDelimiter = '[',
  rightDelimiter = ']'
}) => {
  . . .
}
```

The destructuring syntax `{ separator = ',', leftDelimiter = '[', rightDelimiter = ']' }` declares three parameter variables `separator`, `leftDelimiter`, and `rightDelimiter` that are initialized from the properties with the same names. The defaults are used if the properties are absent or have undefined values.

It is a good idea to provide a default `{}` for the configuration object:

```
const mkString = (array, {
  separator = ',',
  leftDelimiter = '[',
  rightDelimiter = ']'
} = {}) => {
  . . .
}
```

Now the function can be called without any configuration object:

```
const result = mkString(values) // The second argument defaults to {}
```

## 3.14 Hoisting



In this “mad hatter” section, we take up another complex subject that you can easily avoid by following three simple rules. They are:

- Don’t use `var`
- Use strict mode
- Declare variables and functions before using them

If you want to understand what happens when you don’t follow these rules, read on.

JavaScript has an unusual mechanism for determining the *scope* of a variable—that is, is the region of a program where the variable can be accessed. Consider a local variable, declared inside a function. In programming languages such as Java, C#, or C++, the scope extends from the point where the variable is declared until the end of the enclosing block. In JavaScript, a local variable declared with `let` appears to have the same behavior:

```
function doStuff() { // Start of block
  . . . // Attempting to access someVariable throws a ReferenceError
  let someVariable // Scope starts here
  . . . // Can access someVariable, value is undefined
  someVariable = 42
  . . . // Can access someVariable, value is 42
} // End of block, scope ends here
```

However, it is not quite so simple. You *can* access local variables in functions whose declarations precede the variable declaration:

```
function doStuff() {
  function localWork() {
    console.log(someVariable) // OK to access variable
    . . .
  }
  let someVariable = 42

  localWork() // Prints 42
}
```

In JavaScript, every declaration is *hoisted* to the top of its scope. That is, the variable or function is known to exist even before its declaration, and space is reserved to hold its value.

Inside a nested function, you can reference hoisted variables or functions. Consider the `localWork` function in the preceding example. The function knows the location of `someVariable` because it is hoisted to the top of the body of `doStuff`, even though that variable is declared after the function.

Of course, it can then happen that you access a variable before executing the statement that declares it. With `let` and `const` declarations, accessing a variable before it is declared throws a `ReferenceError`. The variable is in the “temporal dead zone” until its declaration is executed.

However, if a variable is declared with the archaic `var` keyword, then its value is simply undefined until the variable is initialized.



**TIP:** Do not use `var`. It declares variables whose scope is the entire function, not the enclosing block. That is too broad:

```
function someFunction(arr) {
  // i, element already in scope but undefined
  for (var i = 0; i < arr.length; i++) {
    var element = arr[i]
    . . .
  }
  // i, element still in scope
}
```

Moreover, `var` doesn't play well with closures—see Exercise 10.

---

Since functions are hoisted, you can call a function before it is declared. In particular, you can declare mutually recursive functions:

```
function isEven(n) { return n === 0 ? true : isOdd(n - 1) }  
function isOdd(n) { return n === 0 ? false : isEven(n - 1) }
```



**NOTE:** In strict mode, named functions can only be declared at the top level of a script or function, not inside a nested block. In non-strict mode, nested named functions are hoisted to the top of their enclosing function. Exercise 12 shows why this is a bad idea.

As long as you use strict mode and avoid `var` declarations, the hoisting behavior is unlikely to result in programming errors. However, it is a good idea to structure your code so that you declare variables and functions before they are used.



**NOTE:** In ancient times, JavaScript programmers used “immediately invoked functions” to limit the scope of `var` declarations and functions:

```
(function () {  
  var someVariable = 42  
  function someFunction(. . .) { . . . }  
  . . .  
})(); // Function is called here—note the ()  
// someVariable, someFunction no longer in scope
```

After the anonymous function is called, it is never used again. The sole purpose is to encapsulate the declarations.

This device is no longer necessary. Simply use:

```
{  
  let someVariable = 42  
  const someFunction = (. . .) => { . . . }  
  . . .  
}
```

The declarations are confined to the block.

## 3.15 Throwing Exceptions

If a function is unable to compute a result, it can throw an exception. Depending on the kind of failure, this can be a better strategy than returning an error value such as `NaN` or `undefined`.

Use a `throw` statement to throw an exception:

```
throw value
```

The exception value can be a value of any type, but it is conventional to throw an error object. The `Error` function produces such an object with a given string describing the reason.

```
let reason = `Element ${elem} not found`  
throw Error(reason)
```

When the `throw` statement executes, the function is terminated immediately. No return value is produced, not even `undefined`. Execution does not continue in the function call but instead in the nearest `catch` or `finally` clause, as described in the following sections.



---

**TIP:** Exception handling is a good mechanism for unpredictable situations that the caller might not be able to handle. It is not so suitable for situations where failure is expected. Consider parsing user input. It is exceedingly likely that some users provide unsuitable input. In JavaScript, it is easy to return a “bottom” value such as `undefined`, `null`, or `NaN` (provided, of course, those could not be valid inputs). Or you can return an object that describes success or failure. For example, in Chapter 9, you will see a method that yields objects of the form `{ status: 'fulfilled', value: result }` or `{ status: 'rejected', reason: exception }`.

---



## 3.16 Catching Exceptions

To catch an exception, use a `try` statement. In Chapter 2, you saw how to catch an exception if you are not interested in the exception value. If you want to examine the exception value, add a variable to the `catch` clause:

```
try {  
  // Do work  
  . . .  
} catch (e) {  
  // Handle exceptions  
  . . .  
}
```

The variable in the catch clause (here, `e`) contains the exception value. As you saw in the preceding section, an exception value is conventionally an error object. Such an object has two properties: `name` and `message`. For example, if you call

```
JSON.parse('{ age: 42 }')
```

an exception is thrown with the name `'SyntaxError'` and message `'Unexpected token a in JSON at position 2'`. (The string in this example is invalid JSON because the `age` key is not enclosed in double quotes.)

The name of an object produced with the `Error` function is `'Error'`. The JavaScript virtual machine throws errors with names `'SyntaxError'`, `'TypeError'`, `'RangeError'`, `'ReferenceError'`, `'URIError'`, or `'InternalError'`.

In the handler, you can record that information in a suitable place. However, in JavaScript it is not usually productive to analyze the error object in detail, as you might in languages such as Java or C++.

When you log an error object on the console, JavaScript execution environments typically display the *stack trace*—the function and method calls between the throw and catch points. Unfortunately, there is no standard way of accessing the stack trace for logging it elsewhere.



**NOTE:** In Java and C++, you can catch exceptions by their type. Then you can handle errors of certain types at a low level and others at a higher level. Such strategies are not easily implemented in JavaScript. A catch clause catches *all* exceptions, and the exception objects carry limited information. In JavaScript, exception handlers typically carry out generic recovery or cleanup, without trying to analyze the cause of failure.

---

When the catch clause is entered, the exception is deemed to be handled. Processing resumes normally, executing the statements in the catch clause. The catch clause can exit with a `return` or `break` statement, or it can be completed by executing its last statement. In that case, execution moves to the next statement after the catch clause.

If you log exceptions at one level of your code but deal with failure at a higher level, then you want to *rethrow* the exception after logging it:

```
try {
  // Do work
  . . .
} catch (e) {
  console.log(e)
  throw e // Rethrow to a handler that deals with the failure
}
```



### 3.17 The finally Clause

A try statement can optionally have a finally clause. The code in the finally clause executes whether or not an exception occurred.

Let us first look at the simplest case: a try statement with a finally clause but no catch clause:

```
try {
  // Acquire resources
  . . .
  // Do work
  . . .
} finally {
  // Relinquish resources
  . . .
}
```

The finally clause is executed in all of the following cases:

- If all statements in the try clause completed without throwing an exception
- If a return or break statement was executed in the try clause
- If an exception occurred in any of the statements of the try clause

You can also have a try statement with catch and finally clauses:

```
try {
  . . .
} catch (e) {
  . . .
} finally {
  . . .
}
```

Now there is an additional pathway. If an exception occurs in the try clause, the catch clause is executed. No matter how the catch clause exits (normally or through a return/break/throw), the finally clause is executed afterwards.

The purpose of the finally clause is to have a single location for relinquishing resources (such as file handles or database connections) that were acquired in the try clause, whether or not an exception occurred.



**CAUTION:** It is legal, but confusing, to have `return/break/throw` statements in the `finally` clause. These statements take precedence over any statements in the `try` and `catch` clauses. For example:

```
try {
  // Do work
  . . .
  return true
} finally {
  . . .
  return false
}
```

If the `try` block is successful and `return true` is executed, the `finally` clause follows. Its `return false` masks the prior `return` statement.

## Exercises

1. What does the `indexOf` function of Section 3.1, “Declaring Functions” (page 51), do when an object is passed instead of an array?
2. Rewrite the `indexOf` function of Section 3.1, “Declaring Functions” (page 51), so that it has a single `return` at the end.
3. Write a function `values(f, low, high)` that yields an array of function values `[f(low), f(low + 1), . . . , f(high)]`.
4. The `sort` method for arrays can take an argument that is a comparison function with two parameters—say, `x` and `y`. The function returns a negative integer if `x` should come before `y`, zero if `x` and `y` are indistinguishable, and a positive integer if `x` should come after `y`. Write calls, using arrow functions, that sort:
  - An array of positive integers by decreasing order
  - An array of people by increasing age
  - An array of strings by increasing length
5. Using the “hard objects” technique of Section 3.7, “Hard Objects” (page 59), implement a `constructCounter` method that produces counter objects whose `count` method increments a counter and yields the new value. The initial value and an optional increment are passed as parameters. (The default increment is 1.)

```
const myFirstCounter = constructCounter(0, 2)
console.log(myFirstCounter.count()) // 0
console.log(myFirstCounter.count()) // 2
```

6. A programmer thinks that “named parameters are almost implemented in JavaScript, but order still has precedence,” offering the following “evidence” in the browser console:

```
function f(a=1, b=2){ console.log(`a=${a}, b=${b}`) }
f() // a=1, b=2
f(a=5) // a=5, b=2
f(a=7, b=10) // a=7, b=10
f(b=10, a=7) // Order is required: a=10, b=7
```

What is actually going on? (Hint: It has nothing to do with named parameters. Try it in strict mode.)

7. Write a function `average` that computes the average of an arbitrary sequence of numbers, using a rest parameter.
8. What happens when you pass a string argument to a rest parameter `...str`? Come up with a useful example to take advantage of your observation.
9. Complete the `mkString` function of Section 3.13, “Simulating Named Arguments with Destructuring” (page 66).
10. The archaic `var` keyword interacts poorly with closures. Consider this example:

```
for (var i = 0; i < 10; i++) {
  setTimeout(() => console.log(i), 1000 * i)
}
```

What does this code snippet print? Why? (Hint: What is the scope of the variable `i`?) What simple change can you make to the code to print the numbers 0, 1, 2, . . . , 9 instead?

11. Consider this declaration of the factorial function:

```
const fac = n => n > 1 ? n * fac(n - 1) : 1
```

Explain why this only works because of variable hoisting.

12. In sloppy (non-strict) mode, functions can be declared inside a nested block, and they are hoisted to the enclosing function or script. Try out the following example a few times:

```
if (Math.random() < 0.5) {
  say('Hello')
  function say(greeting) { console.log(`${greeting}!`) }
}
say('Goodbye')
```

Depending on the result of `Math.random`, what is the outcome? What is the scope of `say`? When is it initialized? What happens when you activate strict mode?

- 
13. Implement an `average` function that throws an exception if any of its arguments is not a number.
  14. Some programmers are confused by statements that contain all three of `try/catch/finally` because there are so many possible pathways of control. Show how you can always rewrite such a statement using a `try/catch` statement and a `try/finally` statement.

---

# Index

## Symbols and Numbers

- (minus sign)
  - in regular expressions, 124
- operator, 9
- operator, 10, 31
  - with arrays, 144
- \_ (underscore)
  - in identifiers, 7
  - in number literals, 100–101
- , (comma)
  - in let statements, 42
  - in URLs, 120
  - operator, in loops, 41
  - trailing, 17, 19
- ;(semicolon)
  - after statements, 28–31
  - in URLs, 120
  - terminating lines with, 6
- : (colon), in URLs, 120
- ! operator, 34, 37–38
- != operator, 35–37
- !== operator, 35
- ? (quotation mark)
  - in regular expressions, 123–127
  - in TypeScript, 291, 304
  - in URLs, 120
- ? : operator, 33–34
  - in TypeScript, 303
- ?: operator
  - in regular expressions, 126, 132
  - in TypeScript, 294
- ?! operator, 137
- ??, ?. operators, 38
- ?<=, ?= operators, 137
- / (slash)
  - in regular expressions, 127
  - in URLs, 120, 211
  - operator, 9, 105
- . (period), in regular expressions, 123–125
- ./, ../, in relative URLs, 211
- ... (ellipsis)
  - in function parameters, 64
  - in rest declarations, 23–24
  - spread operator, 115
  - with arrays, 65–66
- ` . . ` (backticks), 15
- ^ (caret)
  - operator, 39
  - in regular expressions, 123–125, 127–128
- ~ (tilde), operator, 39
- ' . . ', ". . ." (single and double quotes)
  - for string literals, 13
  - matching for, 132–133

- ' ' (empty string)
    - as a Boolean, 34
    - converting to numbers/strings, 11
  - (. . .) (parentheses)
    - in arrow functions, 55
    - in conditional statements, 31
    - in object destructuring, 23
    - in regular expressions, 123–126, 131–133
    - starting statements with, 30
  - [. . .] (square brackets)
    - for arrays, 18, 141
    - for code units, 15
    - for symbol keys, 222
    - in regular expressions, 123–125, 131–133
    - starting statements with, 30
  - {. . .} (curly braces)
    - and object literals, 18
    - around single statements, 33
    - for configuration objects, 67, 292–293
    - in arrow functions, 55
    - in export statements, 214–215
    - in import statements, 212
    - in regular expressions, 123–126, 131–133
  - @ (at), in URLs, 120
  - \$ (dollar sign)
    - in identifiers, 7
    - in regular expressions, 123–128
    - in `String.replace` method, 135
    - in URLs, 120
  - \${. . .} expression, 15, 121, 225
  - \* (asterisk)
    - in export statements, 216
    - in import statements, 212
    - in regular expressions, 123–127, 136
    - operator, 9
    - with generator functions, 254–257, 261
  - ?? operator, 137
  - \*\* operator, 10
  - \ (backslash)
    - in  $\LaTeX$ , 122
    - in regular expressions, 123–127
    - in string literals, 13
    - in template literals, 16
    - in URLs, 122
  - & (ampersand)
    - operator, 39
    - escaping, in HTML, 56
    - in URLs, 120
  - && operator, 34, 37–38
  - # (number sign)
    - in method names, 86
    - in URLs, 120
  - % (percent sign)
    - in URLs, 120
    - operator, 10, 25
  - + (plus sign)
    - in regular expressions, 123–127, 136
    - in URLs, 120
    - operator, 9–11, 224
  - += operator, 137
  - ++ operator, 10, 31
  - += operator, 10
  - < (left angle bracket)
    - escaping, in HTML, 56
    - operator, 34–36, 176
  - << operator, 39
  - <= operator, 34–36, 176
  - <. . .> (angle brackets), for generic parameters, 297
  - = (equal sign)
    - for default parameters, 64
    - in URLs, 120
  - == operator, 35–37, 225
  - === operator, 35
    - for array elements, 149
    - for maps, 158
  - => operator, 54, 176–177
  - >, >= operators, 34–36, 176
  - >>, >>> operators, 39
  - | (vertical bar)
    - operator, 39
    - for union type (TypeScript), 270, 273, 276, 279, 296
    - in regular expressions, 123–126
  - || operator, 34, 37–38
  - 0 (zero)
    - converting to Boolean, 12, 34
    - dividing by, 9
    - leading, in octal numbers, 62, 99
    - 🇺🇸, 14–15, 42, 44, 66, 115–116, 129
    - 🇨🇦, 116–118
    - Å, 178
    - $\pi$ , mathematical constant, 104
    - ß (in German), 118, 177
- ## A
- abs function (Math), 104
  - acos, acosh functions (Math), 104
  - add method (Set), 158

- AggregateError, 197
  - all method (Promise), 196, 201–202
  - AMD (Asynchronous Module Definition), 210
  - any method (Promise), 197
  - any type (TypeScript), 274–275
  - apply method
    - of Function, 237
    - of Reflect, 242
  - arithmetic operators, 9–12
  - Array class
    - concat method, 148, 223, 225
    - copyWithin method, 145–146
    - entries method, 150
    - every method, 147, 149, 151
    - fill method, 145–146
    - filter method, 56–57, 147, 150–152, 225
    - find, findIndex methods, 147, 149, 151
    - firstIndex, lastIndex methods, 147
    - flat method, 147–149, 152, 225
    - flatMap method, 147, 151–152, 225
    - forEach method, 56, 147, 150–152
    - from function, 142, 146, 151–152, 250
    - includes method, 147, 149, 271
    - index, input properties, 143
    - indexOf method, 149
    - isArray function, 279
    - join method, 56, 119, 147, 151–152
    - lastIndexOf method, 149
    - length property, 143
    - map method, 56, 147, 151, 153, 225
    - of function, 142, 164
    - pop, push methods, 144, 146
    - prototype property, 44, 232
    - reduce method, 153–156
    - reduceRight method, 153, 155
    - reverse method, 145–146
    - shift, unshift methods, 144, 146
    - slice method, 147–148, 225
    - some method, 147, 149–151
    - sort method, 145–147, 152, 176
    - splice method, 144–146, 225
    - subarray method, 225
  - array buffers, 163
  - ArrayBuffer class, 163
  - arrays, 18–20
    - building HTML lists from, 56
    - comparing to other types, 36
    - constructing, 65, 141–142
    - converting to:
      - numbers, 11
      - strings, 11, 19
    - destructuring, 21–22, 250
    - elements of:
      - adding, 144
      - computing values from, 153–156
      - deleting, 144
      - filtering, 150
      - iterating over, 42–43, 56–57, 150–151, 249–250, 255–257
      - missing, 143, 152
      - searching, 149
    - empty, 141, 154
    - flattening, 148–149, 256
    - functional processing, 56–57
    - initializing, 66
    - length of, 141, 143
    - multidimensional, 20, 148
    - sorting, 145–147, 152
    - sparse, 152
    - type tests for, 63
    - type variance of, 282–283
    - typed, 160–162
    - with spread operator, 65
  - arrow functions, 54–55
    - and this, 78, 93–95
    - vs. generators, 255
    - with async, 199
  - as keyword, 8
  - as default statement, 215
  - ASCII format, 129
  - asin, asinh, atan, atanh functions (Math), 104
  - asIntN, asUIntN functions (BigInt), 105
  - assign function (Object), 226, 231
  - async/await statements, 8, 197, 197–202
    - concurrent, 201
    - throwing/catching exceptions in, 202
    - with generators, 261–263
    - with module imports, 213
  - AsyncFunction class, 199
  - asynchronous programming, 185–202
    - using generators with, 259–260
  - asyncIterator field (Symbol), 223
  - average function, 62–65, 270
- ## B
- b, B, in binary literals, 99–100
  - \b, \B, in regular expressions, 127
  - bank account objects, 59–60

- BCP 47 memo, 170
- big integers, 105
- big-endian ordering, 163
- BigInt class, 105
- binary data, 163
- binary numbers, 99
- bind method (Function), 236
- bitwise operators, 39
- block statements, 32
- Boolean operators, 37–39
- boolean type (TypeScript), 274–275
- Boolean values, 12
  - converting to numbers/strings, 11
  - in conditions, 32, 34
  - returned by predicate functions, 57
- Booleanness, 34
- branches, 31–33
- break statement, 7, 40, 44–46
  - labeled, 45
  - semicolon insertion in, 30
- browsers
  - CORS restrictions in, 217
  - modules in, 211, 217
  - running JavaScript in, 1–2
  - safety checks in, 192
  - user preferences in, 180
- C**
- `\c`, in regular expressions, 125
- C# language
  - casts in, 279
  - classes in, 82
  - scope of variables in, 67
  - type variance in, 302
- C++ language
  - abstract methods in, 89
  - catching exceptions in, 71
  - classes in, 82
  - concurrent programs in, 185
  - error objects in, 71
  - functions in, 9
  - hash/tree maps in, 158
  - hexadecimal floating-point literals in, 100
  - methods in, 9, 78
  - no-argument constructors in, 91
  - scope of variables in, 67
  - tree nodes in, 96
- call method (Function), 236
- callback functions, 186
- callback hell, 188
- captured variables, 58–60
- case label, 7, 39
- catch method (Promise), 195, 197
- catch statement, 7, 46, 70–73
  - with promises, 190, 200
- cbrt, ceil functions (Math), 104
- character classes, 124–126
- characters
  - combined, 178
  - encoding, 14
  - in regular expressions, 123–125
- Chinese language, 168
- circular references, 234
- class statement, 7, 83–84, 91
  - using `new` operator with, 92
- classes, 83–84
  - anonymous, 91
  - executed in strict mode, 84
  - generic, 297–298
  - private fields in, 86
  - static fields/methods in, 86–87
  - subclasses in, 87–89
  - vs. functions, 82–84
  - vs. modules, 210
- clear method (Map), 157
- clones, 232–234
- closeable iterators, 252–253
- closure pattern, 60
- closures, 57–58
  - and `var`, 68, 74
  - simulating modules through, 210
- code points, 14, 115
- code units, 14, 116
- collation, 176–177
- comments, 5–6
- Common.js module system, 210
- comparisons, 34–37
- compile-time typing, 269
- concat method
  - of Array, 148, 223, 225
  - of String, 119
- concurrent tasks, 185–188
- conditional operator, 33
- conditional statements, 31
  - arbitrary values in, 32, 34
- configuration objects, 67, 292–293
- console.log method, 28, 101
- const statement, 6–7, 17
  - and scope of variables, 68
- constants, mathematical, 104

- constructor keyword, 83–84, 235
- constructors, 81–83
  - setting object properties in, 85
  - subclass/superclass, 90–91
- continue statement, 7, 46
  - labeled, 46
  - semicolon insertion in, 30
- control flow statements, 30–33
- copyWithin method (Array), 145–146
- CORS (Cross-Origin Resource Sharing), 192, 217
- cos, cosh functions (Math), 104
- country codes, 168–170
- create function (Object), 226, 231–232
- Crockford, Douglas, 60
- cross-origin errors, 192
- currency, displaying, 172

## D

- `\d`, `\D`, in regular expressions, 124–125
- data transfer, 186, 259
- DataView class, 163
- Date class, 106–108, 173
  - `getXXX` methods, 107, 109
  - mutability of, 110
  - `now`, parse functions, 109
  - `setXXX` methods, 109
  - `toXXX` methods, 107, 110
  - UTC function, 107, 109
- dates
  - constructing, 106–109
  - current, 109
  - distance between, 108
  - formatting, 110, 173–174
  - localizing, 168, 173–174
  - parsing, 109
  - ranges of, 174
- daylight savings time, 106
- debugger statement, 7
- decimal numbers, 99
- default keyword, 7, 39
- default values, 38
- `defineProperties` function (Object), 226, 228
- `defineProperty` function
  - of Object, 226–227, 239, 242
  - of Reflect, 242
- `delete` method
  - of Map, 157
  - of Set, 158
  - of WeakMap/WeakSet, 160

- `delete` statement, 7, 17
  - applied to unqualified identifiers, 61
- `deleteProperty` method (Reflect), 239, 242
- destructuring, 21–24, 66–67
- development tools console
  - enabling strict mode in, 61
  - running JavaScript in, 2–3
- `do` statement, 7, 40–41
- `Document.querySelectorAll` method, 250
- documentation comments, 6
- DOM nodes, attaching properties to, 159
- `dotAll` property (RegExp), 125, 128, 133
- duck typing, 288

## E

- e*, mathematical constant, 104
- e*, *E*, in number literals, 99–100
- Eclipse development environment, 4
- ECMAScript Internationalization API (ECMA-402), 180
- ECMAScript language
  - module system in, 209–211
  - specification of, 79
- `else` statement, 7, 32
- `else if` statement, 32
- emojis
  - encoding, 116
  - in regular expressions, 130
- empty string
  - as a Boolean, 34
  - converting to numbers/strings, 11
- `encodeURIComponent` function (String), 120
- endianness, 163
- `endsWith` method (String), 116, 119
- English language
  - dates in, 168
  - plurals in, 178–179
  - string ordering in, 176
- `entries` method
  - objects returned by, 250
  - of Array, 150
  - of Map, 158
  - of Object, 226, 229
  - of Set, 159
- enum, 7
- epoch, 106
- equality testing, 35
- Error function, 70–71
- error objects, 71
- escape sequences, 13

- every method (Array), 147, 149, 151
- exceptions
  - catching, 46–47, 70–73
  - throwing, 46, 69–72
- exec method (RegExp), 130–131, 133–134
- exp, expm1 functions (Math), 104
- export statement, 7, 213–216
- export default statement, 214–215
- expression statements, 28
- expressions, 27–29
- extends, 7, 87–91
  - in TypeScript, 300, 303–304
- F**
- \f, in regular expressions, 125
- factory class pattern, 60
- factory functions, 78
  - invoking, 81
  - local variables in, 59–60
  - parameters in, 60
- failures, 69
- false value, 7, 12, 34
- fetch function (Promise), 192–193
- Fetch API, 188, 193, 199
- fields
  - private, 86
  - private static, 87
  - static, 86–87
- File API, 163
- file:// URLs, 211
- fill method (Array), 145–146
- filter method (Array), 56–57, 147, 150–152, 225
- finally statement, 7, 72–73
- find, findIndex methods (Array), 147, 149, 151
- firstIndex method (Array), 147
- flags property (RegExp), 128, 133
- flat method (Array)
  - of Array, 147–149, 152, 225
  - of typed arrays, 162
- flatMap method
  - of Array, 147, 151–152, 225
  - of typed arrays, 162
- floating-point numbers, 8, 99–100
- FloatXXXArray classes, 160–161
- floor function (Math), 39, 104
- Flow type checker, 270
- for await of statement, 201, 261–263
- for each loop (Java), 43
- for in statement, 43–44
  - and legacy libraries, 44
  - for arrays, 150
  - iterating over a string, 44
- for method (Symbol), 222
- for of statement, 42, 249–251, 253, 255, 262
  - for arrays, 150
  - for sets, 159
- for statement, 7, 41–42
- forEach method
  - of Array, 56, 147, 150–152
  - of Map, 157
  - of Set, 159
- format method
  - of Intl.DateTimeFormat, 173
  - of Intl.ListFormat, 179
  - of Intl.NumberFormat, 171–172
  - of Intl.RelativeTimeFormat, 175
- formatRange method (Intl.DateTimeFormat), 174
- formatToParts method
  - of Intl.DateTimeFormat, 175
  - of Intl.NumberFormat, 171
  - of Intl.RelativeTimeFormat, 175
- free variables, 57–58
- freeze function (Object), 60, 226, 230
- from function (Array), 142, 146, 151–152, 250
- from keyword, 8
- fromCodePoint function (String), 115, 119
- fromEntries function (Object), 226, 231
- fround, ftrunc, ffloor, fceil functions (Math), 104
- function statement, 7, 51, 54
  - and this, 78
  - comparing to arrow functions, 55
  - default parameters in, 66
  - for nested functions, 93
  - rest declarations in, 66
  - type annotations with, 271
- Function class, 235
  - apply method, 237
  - bind method, 236
  - call method, 236
- function literals, 54
- functional array processing, 56–57
- functional programming languages, 51
- functions, 9
  - anonymous, 54–55, 69, 199
  - applying strict mode to, 61
  - arrow, 54–55, 78, 93–95, 199, 255
  - async, 199, 263

- callback, 186
- calling, 52
  - before declaring, 69
  - flexibly, 295
- configuration objects for, 67, 292–293
- declaring, 51–52, 61, 69
- executing later, 57
- factory, 59–60, 78, 81
- generic, 297–299
- higher-order, 53
- immediately invoked, 69
- mathematical, 104
- named, 54, 199
- nested, 55, 68–69, 93
- parameterless, 55
- parameters of:
  - always converted to numbers, 63
  - annotating, 270
  - default, 64, 66
  - destructuring, 292–293
  - duplicating, 62
  - named, 66
  - number of, 63–65
  - optional, 291
  - types of, 52, 62–63
- predicate, 57
- properties of, 235–236
- return value of, 52–53
- storing in variables, 53
- terminating, 52
- trap, 238–242
- type tests for, 63
- type variance of, 293–295
- vs. classes in Java/C++, 82
- with free variables, 57–58

**G**

- g flag, in regular expressions, 128, 131
- garbage collection, 160
- generators, 254–255
  - as consumers, 257–259
  - asynchronous, 259–263
- generic programming, 297–304
- get keyword, 8, 85, 227–228
  - of proxies, 242
- get method
  - of Map, 157
  - of WeakMap, 160
- getCanonicalLocales method (Intl), 180

- getIntXXX, getUIntXXX, getFloatXXX methods (DataView), 163
- getOwnPropertyXXX functions
  - of Object, 226, 228–230, 232, 239
  - of proxies, 242
- getPrototypeOf function
  - of Object, 80, 226, 231–232, 239
  - of proxies, 242
- getters, 85, 228
  - overriding, 90
  - static, 87
- getUTCXXX, getXXX methods (Date), 107, 109
- getYear method (Date), obsolete, 109
- global property (RegExp), 128, 131, 133
- Greek letters, 122
- groups property (RegExp), 133
- Gulliver's Travels*, 163

## H

- Hanafuda playing cards, 187
- hard objects, 59–60
- has method
  - of Map, 157
  - of proxies, 242
  - of WeakMap/WeakSet, 160
- hasInstance field (Symbol), 223
- hasOwnProperty method (Object), 226, 230
- hexadecimal numbers, 99
- higher-order functions, 53
- hoisting, 67–69
- HTML (HyperText Markup Language)
  - building lists from arrays in, 56
  - canvas images in, 161
  - escaping characters in, 56
  - hyperlinks in, 123
- HTMLCollection collection, 141
- hyperbolic functions, 104
- hypot function (Math), 104

## I

- i flag, in regular expressions, 127–128
- identifiers, 7–8
  - unqualified, 61
- IEEE 754 standard, 99
- if statement, 7, 31
  - vs. switch, 40
- ignoreCase property (RegExp), 127–128, 133
- images
  - in HTML canvas, 161
  - loading, 186–187, 201–202

- immediately invoked functions, 69
  - implements keyword, 8, 288
  - import statement, 7, 211–213
  - import.meta object, 217
  - in statement, 7, 143, 229
    - in TypeScript, 279
  - includes method
    - of Array, 147, 149, 271
    - of String, 116, 119
  - index property (Array), 143
  - index signatures, 290–291
  - indexOf method
    - of Array, 149
    - of String, 116, 119, 136
  - infer keyword, 303
  - Infinity variable, 100, 102
  - inheritance, 87–89
  - input property (Array), 143
  - instance fields, 85–86
  - instanceof operator, 7, 89, 232
    - in TypeScript, 279, 301
  - integers, 99
    - big, 105
    - no explicit type for, 8
    - rounding numbers to, 9
  - interface keyword, 8
  - interfaces, 288–290
  - internal slots, 79
  - internationalization, 167–181
  - Internet Engineering Task Force, 170
  - Intl.Collator class, 177
    - resolvedOptions method, 181
    - supportedLocalesOf method, 180
  - Intl.DateTimeFormat class
    - format method, 173
    - formatRange method, 174
    - formatToParts method, 175
    - supportedLocalesOf method, 180
  - Intl.DisplayNames class, 180
  - Intl.getCanonicalLocales method, 180
  - Intl.ListFormat class, 179
    - select method, 179
    - supportedLocalesOf method, 180
  - Intl.NumberFormat class
    - format method, 171–172
    - formatToParts method, 171
    - supportedLocalesOf method, 180
  - Intl.PluralRules class, 178–179
    - select method, 178
    - supportedLocalesOf method, 180
  - Intl.RelativeTimeFormat class, 175
    - format, formatToParts methods, 175
    - supportedLocalesOf method, 180
  - IntXXXArray classes, 160–161
  - is function (Object), 35
  - isArray function (Array), 279
  - isConcatSpreadable field (Symbol), 149, 223
  - isExtensible function (Object), 226, 230, 239
  - isFinite function
    - global, 102
    - of Number, 102–103
  - isFrozen function (Object), 226, 230
  - isInteger function (Number), 102
  - isNaN function
    - global, 102
    - of Number, 35, 102–103
  - ISO 8601 format, 106
  - isPrototypeOf method (Object), 226, 232
  - isSafeInteger function (Number), 102–103
  - isSealed function (Object), 226, 230
  - iterables, 42, 141, 249–250
    - async, 261
    - implementing, 250–252
  - iterator field (Symbol), 222–223, 250–253
  - iterators, 252–253
- ## J
- Japanese imperial calendar, 169
  - Java language
    - abstract methods in, 89
    - captured variables in, 58
    - casts in, 279
    - catching exceptions in, 71
    - classes in, 82
    - concurrent programs in, 185
    - Date class in, 106
    - error objects in, 71
    - extends keyword in, 89
    - for each loop in, 43
    - hash/tree maps in, 158
    - hexadecimal floating-point literals in, 100
    - methods in, 9, 78
    - modules in, 210–211
    - no-argument constructors in, 91
    - packages in, 210
    - scope of variables in, 67
    - tree nodes in, 96
    - type variance in, 302

JavaScript language  
  functional, 51  
  running, 1–4  
  stage 3 proposal, 38, 86–87, 100, 134, 171, 173, 180–181, 197, 217, 284  
  stage 4 proposal, 213  
join method (Array), 56, 119, 147, 151–152  
.js extension, 217  
JSDoc tool, 6  
JSON (JavaScript Object Notation), 20–21  
  asynchronous processing of, 199  
  parse method, 20, 47  
  stringify method, 20–21  
jump tables, 40

## K

key/value pairs, 156  
keys method  
  objects returned by, 250  
  of Map, 158  
  of Object, 226, 229, 239, 243–244  
  of Set, 159  
keywords, 7–8  
Komodo development environment, 4

## L

labeled breaks, 45  
language codes, 168–170  
lastIndex method (Array), 147  
lastIndex property (RegExp), 131  
lastIndexOf method  
  of Array, 149  
  of String, 116, 119  
L<sup>A</sup>T<sub>E</sub>X language, 122  
lazy evaluations, 37  
leap seconds, 105  
length property (Array), 143  
let statement, 6–8  
  and scope of variables, 67–68  
letters  
  in regular expressions, 130  
  lowercase/uppercase, 118, 120  
light-server http server, 187  
LinkedHashMap (Java), 157  
little-endian ordering, 163  
loadImage function (Promise), 192  
local time zone, 107–108  
localeCompare method (String), 147, 176–178  
locales, 167–168  
  dates in, 107, 110, 168, 172

  displaying, 180  
  numbers in, 167–168, 171–172  
  plurals in, 178–179  
  specifying, 168–170  
  strings in, 177–178  
  sorting, 147, 176–177  
  tags for, 180  
  time in, 172  
log, log<sub>2</sub>, log<sub>10</sub>, log<sub>1p</sub> functions (Math), 104  
logging, 21, 215  
lookahead/lookbehind operators, 137  
loops, 40–44  
  condition values for, 34  
  exiting, 44–46  
loose equality, 35–37  
lowercase letters  
  converting to, 118, 120  
  in regular expressions, 130  
lvalues, 22

## M

m flag, in regular expressions, 128  
map method (Array), 56, 147, 151, 153, 225  
Map class, 156  
  generic, 298  
  methods of, 157–158  
  size property of, 157  
mapped types, 303–304  
maps, 156–158  
  adding/removing entries of, 157  
  constructing, 157, 250  
  empty, 157  
  iterating over, 250  
  keys of, 157  
  traversing, 157  
  weak, 160  
match, matchAll fields (Symbol), 223  
match, matchAll methods (String), 120, 131, 134  
Math class, 103–104  
  constants of, 104  
  functions of, 104  
  floor, 39, 104  
  max, 33, 42, 65, 103–104  
  min, 103–104  
  pow, 104, 276  
  random, 104  
  round, 104  
  sqrt, 104, 276  
  trunc, 9, 104

- Maven artifacts, 211
  - methods, 9, 77–78
    - default values for, 38
    - intersecting invocations of, 237
    - invoked:
      - with invalid arguments, 46
      - without an object, 92
    - overriding, 89–90
    - private, 86
    - private static, 87
    - setting object properties in, 85
    - static, 86–87
    - with async, 199
  - Microsoft, 269
  - .mjs extension, 217
  - modules, 209–217
    - circular dependencies between, 215–216
    - default feature of, 211
    - exported vs. private features of, 209
    - exporting, 213–216
    - importing, 211–213
    - loading asynchronously, 210
    - named features in, 212
    - packaging, 217
    - processing, 217
    - URLs for, 211
    - vs. classes/closures, 210
  - multiline property (RegExp), 128, 133
- ## N
- n, in big integers, 105
  - \n, in regular expressions, 125
  - name conflicts, 210
  - named arguments, 66
  - NaN (Not A Number), 9, 12–13
    - arithmetic operations with, 10–11
    - assigning new values to, 61
    - checking for, 102
    - comparing to, 35–36
    - converting to Boolean, 12, 34
    - global variable for, 100, 102
  - navigator.languages property, 180
  - never type (TypeScript), 274–275
  - new operator, 7, 81–82, 84, 231
    - missing in a constructor function, 92
    - with Array, 142
    - with maps, 157
  - next method (iterators), 250, 261
  - Node.js
    - enabling strict mode in, 61
    - modules in, 210, 217
      - rejected promises in, 202
      - running JavaScript in, 3–4
  - NodeList collection, 141
  - non-strict mode, 62, 69, 74
  - normalize method (String), 178
  - Norwegian language, 169
  - now function (Date), 109
  - npm package manager, 272
  - null type (TypeScript), 274–275
  - null value, 5, 7
    - arithmetic operations with, 11
    - checking values for, 38
    - comparing to, 35, 37
    - converting to numbers/strings, 11
  - Number class, 92, 103
    - constants of, 102–103
    - isFinite function, 102–103
    - isInteger function, 102
    - isNaN function, 35, 102–103
    - isSafeInteger function, 102–103
    - parseFloat, parseInt functions, 8–9, 46, 101–102
    - toExponential, toFixed, toPrecision methods, 100, 103
    - toLocaleString method, 171–172
    - toString function, 9, 82, 100, 103
  - number type (TypeScript), 273–275
  - number[] type (TypeScript), 273
  - numbers, 8–9, 99–100
    - comparing, 34
      - to other types, 36–37
    - converting:
      - from strings, 8, 36, 92, 101
      - to strings, 9, 11, 100
    - dividing by zero, 9
    - finding largest/smallest of, 103
    - formatting, 100, 167–168, 171–172
    - in regular expressions, 130
    - parsing, 101–102
    - random, 104, 259–260
    - removing fractional part of, 39
    - rounding, 8–9, 104
    - type tests for, 63
  - Nynorsk standard, 169
- ## O
- o, 0, in octal literals, 99–100
  - obj.\_\_proto\_\_ notation, 80

- Object class, 77
    - assign function, 226, 231
    - create function, 226, 231–232
    - defineProperties function, 226, 228
    - defineProperty function, 226–227, 239, 242
    - entries method, 226, 229
    - freeze function, 60, 226, 230
    - fromEntries function, 226, 231
    - getOwnPropertyXXX functions, 226, 228–230, 232, 239
    - getPrototypeOf function, 80, 226, 231–232, 239
    - hasOwnProperty method, 226, 230
    - is function, 35
    - isExtensible function, 226, 230, 239
    - isFrozen function, 226, 230
    - isPrototypeOf method, 226, 232
    - isSealed function, 226, 230
    - keys method, 226, 229, 239, 243–244
    - preventExtensions function, 226, 230, 239
    - propertyIsEnumerable method, 226, 230
    - prototype property, 44, 82
    - seal function, 226, 230
    - setPrototypeOf function, 80, 226, 231, 239
    - toLocaleString method, 170
    - toString method, 223–224
    - values method, 226, 229
  - object literals, 17
  - object-oriented programming languages, 77
    - inheritance in, 87
  - objects, 16–17, 77
    - array-like, 142
    - as map keys, 158
    - attaching properties to, 222
    - cloning, 232–234
    - comparing, 35
    - converting to numbers/strings, 11
    - creating, 231
    - destructuring, 22–24
    - hard, 59–60
    - immutable, 230
    - internal slots of. *See* prototypes
    - iterable, 42, 141
    - keys of, 221
    - nested, 23
    - properties of. *See* properties
    - protecting, 230
    - type variance of, 282–283
    - wrapper, 5
  - octal numbers, 62, 99
    - regular expressions for, 132
  - of function
    - of Array, 142, 164
    - of typed arrays, 161
  - of keyword, 8
  - offending tokens, 29
  - One True Brace Style (1TBS), 32
  - operators
    - arithmetic, 9–12
    - bitwise, 39
    - Boolean, 37–39
    - greedy, 136
    - shift, 39
  - ORM (object-relational mapper), 237
- ## P
- \p, \P, in regular expressions, 125
  - package statement, 8
  - package.json file, 217
  - padStart, padEnd methods (String), 118–119
  - parse function (Date), 109
  - parse method (JSON), 20, 47
  - parseFloat, parseInt functions (Number), 8, 101–102
    - invalid parameters for, 9, 46
  - pipelines, 193, 197
  - plurals, 178–179
  - polymorphism, 90
  - pop method (Array), 144, 146
  - pow function (Math), 104, 276
  - predefined character classes, 124–126
  - predicate functions, 57
  - preventExtensions function (Object), 226, 230, 239
  - primitive types, 5
    - controlling conversion on, 224–225
  - printf function (C), 101
  - private keyword, 8
  - private properties, 59
  - programming
    - asynchronous, 185–202, 259–260
    - functional, 51
    - generic, 297–304
    - object-oriented, 77
  - Promise class, 188–190
    - all method, 196, 201–202
    - any method, 197
    - catch method, 195, 197
    - fetch function, 192–193

- loadImage function, 192
  - race method, 197
  - reject method, 189–191
  - resolve method, 189–191, 193–195
  - then method, 192–197
  - promises, 185, 188–202
    - making, 188–191
    - multiple:
      - chaining, 192–194
      - executing, 196
      - racing, 197
    - obtaining results of, 191–192
    - rejected, 190–191, 194–197, 202
    - settled, 190
      - immediately, 191
  - properties (of objects)
    - attributes of, 225–228
    - common to multiple objects, 79
    - descriptors of, 227
    - dynamic, 227, 244
    - enumerating, 228–229
    - iterating over, 43
    - testing, 229–230
    - writing to, 81
  - propertyIsEnumerable method (Object), 226, 230
  - protected keyword, 8
  - prototype property, 82, 235
  - prototype chain, 81, 88, 229, 231
  - [[Prototype]] internal slot, 82
  - prototypes, 78–81
    - accessing, 231
    - lookup mechanism of, 80–81
    - setting, 80
    - updating, 231
  - proxies, 237–239
    - checking return values of, 242
    - invariants for, 242–244
  - Proxy.revocable function, 239
  - public keyword, 8
  - punctuation, in regular expressions, 130
  - push method (Array), 144, 146
- Q**
- QED text editor, 124
  - querySelectorAll method (Document), 250
- R**
- \r, in regular expressions, 125
  - race method (Promise), 197
  - random function (Math), 104
  - random numbers, 104, 259–260
  - RangeError, 162
  - raw property (String), 122
  - readonly property (TypeScript), 285, 303
  - reduce method (Array), 153–156
  - reduceRight method (Array), 153, 155
  - reduction, 153–156
  - ReferenceError, 62, 68
  - Reflect class, 240–242
    - functions of, 239, 242
  - RegExp class, 127, 130–131
    - exec method, 130–131, 133–134
    - flags property, 133
    - groups property, 133
    - lastIndex property, 131
    - test method, 130, 133
  - regular expression literals, 127
  - regular expressions, 123–127
    - and Unicode, 129–130
    - character classes in, 124–126
    - flags in, 127–128
    - for numbers, 101
    - groups in, 131–133
    - lookahead/lookbehind operators in, 137
    - reserved characters in, 123–125
  - reject method (Promise), 189–191
  - repeat method (String), 118–119
  - REPL (“read-eval-print” loop), 28
  - replace method
    - of String, 120, 134–136
    - of Symbol, 223
  - resolve method (Promise), 189–191, 193–195
  - resolvedOptions method (Intl.Collator), 181
  - rest declarations, 23–24, 64–66
  - return method (iterators), 252–253, 258
  - return statement, 7
    - in arrow functions, 55
    - in function declarations, 52–53
    - semicolon insertion in, 30, 53
  - reverse method (Array), 145–146
  - revocable method (Proxy), 239
  - round function (Math), 104
  - roundoff errors, 8
  - run-time errors, 6
  - Russian language, 178–179
- S**
- s flag, in regular expressions, 128
  - \s, \S, in regular expressions, 124, 126
  - script element (HTML), 217

- seal function (Object), 226, 230
- search method
  - of String, 120, 134, 136
  - of Symbol, 223
- select method (Intl.PluralRules), 178
- semicolon insertion, 28–31, 53
- Set class, 158–159
  - add method, 158
  - delete method, 158
  - entries method, 159
  - forEach method, 159
  - generic, 298
  - keys method, 159
  - size property, 158
  - values method, 159
- set keyword, 8, 85, 227–228
- set method
  - of Map, 157
  - of typed arrays, 162
  - of WeakMap/WeakSet, 160
- setIntXXX, setUIntXXX, setFloatXXX methods (DataView), 163
- setPrototypeOf function (Object), 80, 226, 231, 239
- sets, 158–159
  - constructing, 158, 250
  - iterating over, 250
  - order of entries in, 159
  - weak, 160
- setters, 85, 228
  - overriding, 90
  - static, 87
- setTimeout function, 57
- shift method (Array), 144, 146
- shift operators, 39
- sign function (Math), 104
- sin, sinh functions (Math), 104
- size property
  - of Map, 157
  - of Set, 158
- slice method
  - of Array, 147–148, 225
  - of String, 117, 119
- sloppy mode, 62, 69, 74
- some method (Array), 147, 149–151
- sort method (Array), 145–147, 152, 176
- species field (Symbol), 222–223, 225
- splice method (Array), 144–146, 225
- split method
  - of String, 117, 120, 135
    - of Symbol, 223
- spread operator, 65–66, 115, 231, 250
  - for cloning, 232
- sqrt function (Math), 104, 276
- stack trace, 71
- startsWith method (String), 116, 119
- statements, 27–29
  - block, 32
  - conditional, 31
  - control flow, 30–33
  - expression, 28
  - terminated with semicolons, 29–31
- static keyword, 8, 86–87
- sticky property (RegExp), 128, 131, 133
- strict equality, 35
- strict mode, 61–62, 67–69
  - creating variables in, 7
  - enabling, 61
  - for classes, 84
  - for modules, 217
  - forbidden keywords in, 8
  - missing new operator in, 92
  - octal numbers in, 99, 132
  - property attributes in, 227
- String class
  - concat method, 119
  - endsWith method, 116, 119
  - fromCodePoint function, 115, 119
  - includes method, 116, 119
  - indexOf method, 116, 119, 136
  - lastIndexOf method, 116, 119
  - localeCompare method, 147, 176–178
  - match, matchAll methods, 120, 131, 134
  - normalize method, 178
  - padStart, padEnd methods, 118–119
  - raw property, 122
  - repeat method, 118–119
  - replace method, 120, 134–136
  - search method, 120, 134, 136
  - slice method, 117, 119
  - split method, 117, 120, 135
  - startsWith method, 116, 119
  - substring method, 116
  - toLocaleXXXCase methods, 177
  - toLowerCase, toUpperCase methods, 118, 120
  - trim, trimStart, trimEnd methods, 118–119
- string literals, 13–15
- string type (TypeScript), 274–275
- stringify method (JSON), 20

- strings
    - comparing, 35, 176
    - concatenating, 10–11, 119, 270
    - converting:
      - from arrays, 11, 19, 151
      - from numbers, 9, 100
      - from/to code points, 115
      - to numbers, 8, 11, 36, 92, 101
    - extracting substrings from, 9
    - iterating over, 42, 44, 250
    - length of, 118–119
    - numeric, 177
    - patterns for. *See* regular expressions
    - replacing, 120
    - sorting, 176–177
    - type tests for, 63
    - with embedded expressions. *See* template literals
    - working with locales, 177–178
  - strong element (HTML), 121
  - subarray method
    - of Array, 225
    - of typed arrays, 162
  - subclasses, 87–89
    - overriding methods in, 89–90
  - substring method (String), 116–117
  - substrings, 116–117
  - super keyword, 7, 90–91
  - superclasses, 87
    - constructors of, 237
    - overriding methods of, 89–90
  - supportedLocalesOf method (Intl.XXX), 180
  - switch statement, 7, 39–40
  - Symbol class, 221
    - asyncIterator field, 223
    - for method, 222
    - hasInstance field, 223
    - isConcatSpreadable field, 223
    - iterator field, 222–223, 250–253
    - match, matchAll fields, 223
    - replace field, 223
    - search field, 223
    - species field, 222–223, 225
    - split field, 223
    - toPrimitive field, 223–225
    - toStringTag field, 223
  - symbol type (TypeScript), 274–275
  - symbols, 221–222
    - creating, 221
    - global, 222
    - in regular expressions, 130
    - properties of, 223
    - uniqueness of, 222
- ## T
- T, in dates, 106
  - \t, in regular expressions, 125
  - tag functions, 121
  - tan, tanh functions (Math), 104
  - target keyword, 8
  - template literals, 12, 15–16
    - raw, 122–123
    - tagged, 16, 121–122
  - test method (RegExp), 130, 133
  - Thai numerals, 168–169
  - then method (Promise), 192–197
  - this reference, 7, 78, 92–95
    - and calls to super, 90
    - binding, 236
    - for object properties, 85
    - in constructor functions, 81–83
    - in TypeScript, 286
    - vs. captured variables, 60
    - with arrays, 151
  - threads, 185
  - throw method (iterators), 258–259
  - throw statement, 7, 70
    - semicolon insertion in, 30
  - time
    - current, 107–109
    - daylight savings, 106
    - formatting, 175
    - local, 107–109
    - localizing, 172, 175
    - measuring, 105
    - representing in computer, 106
  - time-consuming operations, 186
  - timeout, 57
  - ™ (trademark symbol), 178
  - toExponential, toFixed, toPrecision methods (Number), 100, 103
  - toLocaleString method
    - of Number, 171–172
    - of Object, 170
  - toLocaleXXXCase methods (String), 177
  - toLowerCase, toUpperCase methods (String), 118, 120
  - toPrimitive field (Symbol), 223–225
  - toString method
    - of Date, 110

- of Number, 9, 82, 100, 103
- of Object, 223–224
- toStringTag field (Symbol), 223
- toXXXString methods (Date), 107, 110
- trailing comma
  - in arrays, 19
  - in object literals, 17
- trap functions, 238–242
- trigonometric functions, 104
- trim, trimStart, trimEnd methods (String), 118–119
- true value, 7, 12, 34
- trunc function (Math), 9, 104
- try statement, 7, 46–47, 70–73
  - with promises, 190, 200
- tsconfig.json file, 272–273
- type annotations, 270–271
- type parameters, 297
- type variance, 282–283
  - generic, 302
  - of functions, 293–295
- typeof operator, 5, 7, 105
  - in TypeScript, 274, 279
  - with arrays, 19
  - with regular expressions, 127
  - with symbols, 222
- types, 5
  - comparing, 36–37
  - controlling conversion on, 224–225
  - generic, 297–298
  - inference of, 277–280
  - of function parameters, 52, 62–63
  - testing, 63
- TypeScript, 269–304
  - classes in, 284–287
    - declaring, 284–285
    - instances of, 285–286
    - static types of, 286–287
  - composite types in, 273, 275–277
    - number[], 273, 275
    - string[], 275
    - union, 270, 273, 276, 279, 296
  - conditional types in, 303
  - covariance in, 283
  - enumerated types in, 275
  - functions in, 276
    - destructuring parameters of, 292–293
    - overloading, 296–297
    - type guard, 279
    - type variance of, 293–295

- in statement in, 279
- index signatures in, 290–291
- instanceof operator in, 279, 301
- interfaces in, 288–290
- intersection types in, 276
- mapped types in, 303–304
- object types in, 276
- optional parameters in, 291–292
- optional/excess properties in, 281–282
- primitive types in, 273–275
- private/protected modifiers in, 285
- running, 271–273
- substitution rule in, 280–281
- tuple types in, 275
- type aliases in, 273
- type assertion in, 279
- type bounds in, 299–300
- type erasures in, 300–301
- type inference in, 277
- type parameters in, 297–304
- type variance in, 302
- typeof operator in, 274, 279

## U

- u flag, in regular expressions, 128–129
- \u{ . . . } notation, of code points, 14
  - in regular expressions, 125, 129
- UintXXXArray classes, 160–161
- undefined type (TypeScript), 274–275
- undefined value, 6, 9, 12–13
  - arithmetic operations with, 11
  - as function return value, 52
  - assigning new values to, 61
  - checking values for, 38
  - comparing to, 35
  - converting to:
    - Boolean, 12, 34
    - numbers/strings, 11
  - for function parameters, 64
- undefined variable, 13
- unicode property (RegExp), 128–129, 133
- Unicode, 14
  - combined characters in, 178
  - in regular expressions, 128–130
  - normalization forms in, 178
- union type (TypeScript), 270, 273, 276, 279, 296
- unknown type (TypeScript), 274–275
- unqualified identifiers, 61
- unshift method (Array), 144, 146

- uppercase letters
  - converting to, 118, 120
  - in regular expressions, 130
- URLs
  - \ (backslashes) in, 122
  - for modules, 211
  - safe characters for, 120
- UTC (Coordinated Universal Time), 106, 109
- UTC function (Date), 107, 109
- UTF-16 encoding, 116
  - in regular expressions, 129–130
- V**
- \v, in regular expressions, 125
- values
  - default, 38
  - finding type of, 5
  - iterable, 141, 249–250
- values method
  - objects returned by, 250
  - of Map, 158
  - of Object, 226, 229
  - of Set, 159
- var statement (obsolete), 7
  - and scope of variables, 67–69
  - with closures, 68, 74
- variable declarations, 6–7, 27
- variables
  - captured, 58–60
  - default values of, 24
  - free, in functions, 57–58
  - initializing, 6, 12
  - local, 59, 67–69
  - naming, 7–8
  - never chaining, 6
  - scope of, 67–69
  - storing functions in, 53
  - undeclared, 61–62
  - untyped, 6
- Visual Studio Code development environment, 4, 272
- void keyword, 7
- void type (TypeScript), 274–275
- W**
- \w, \W, in regular expressions, 124, 126
- weak equality, 35–37
- WeakMap class, 160
  - generic, 298
- WeakSet class, 160
- web servers, local, 187
- WebSocket API, 163
- WebStorm development environment, 4
- while statement, 7, 40–41
- whitespace
  - in regular expressions, 130
  - leading/trailing, 118–119
- with statement, 7, 62
- wrapper objects, 5
- X**
- x, X, in hexadecimal literals, 99–100
- \x, in regular expressions, 125
- XMLHttpRequest class, 186, 259
  - yielding array buffers, 163
- Y**
- y flag, in regular expressions, 128, 131
- yield statement, 8, 250, 254
  - nested, 255–257
  - semicolon insertion in, 30
  - shallowness of, 255
  - with consumers, 257–259
- Z**
- Zawinski, Jamie, 137
- zero. *See* 0