Modern JavaScript for the Impatient

Cay S. Horstmann

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

Library of Congress Control Number: 2020934310

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

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*[1] was great in 1996, but now it burdens readers with too many accidents from the past. Crockford's *JavaScript: The Good Parts*[2] 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

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	`typeof x === 'string' || x instanceof String`	`x` might be constructed as `new String(. . .)`
Regular expression	`x instanceof RegExp`	
Number	`typeof x === 'number' || x instanceof Number`	`x` might be constructed as `new Number(. . .)`
Anything that can be converted to a number	`typeof +x === 'number'`	Obtain the numeric value as `+x`
Array	`Array.isArray(x)`	
Function	`typeof x === 'function'`	

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 + 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 particularly, 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