

# JavaScript

FOR IMPATIENT PROGRAMMERS



**Dr. Axel Rauschmayer**



# Contents

<b>I</b>	<b>Background</b>	<b>11</b>
<b>1</b>	<b>About this book</b>	<b>13</b>
1.1	What's in this book? . . . . .	13
1.2	What is not covered by this book? . . . . .	13
1.3	This book isn't finished, yet . . . . .	14
1.4	Can I buy a print edition? . . . . .	14
1.5	Will there be a free online version? . . . . .	14
1.6	Acknowledgements . . . . .	14
<b>2</b>	<b>FAQ: book</b>	<b>15</b>
2.1	What are the "advanced" chapters and sections about? . . . . .	15
2.2	What should I read if I'm <i>really</i> impatient? . . . . .	15
2.3	How do I submit feedback and corrections? . . . . .	16
2.4	How do I get updates for the downloads at Payhip? . . . . .	16
2.5	I'm occasionally seeing type annotations – how do those work? . . . . .	16
2.6	What do the notes with icons mean? . . . . .	16
<b>3</b>	<b>History and evolution of JavaScript</b>	<b>19</b>
3.1	How JavaScript was created . . . . .	19
3.2	Standardization . . . . .	20
3.3	Timeline of ECMAScript versions . . . . .	20
3.4	Ecma Technical Committee 39 (TC39) . . . . .	21
3.5	The TC39 process . . . . .	21
3.6	FAQ: TC39 process . . . . .	23
3.7	Evolving JavaScript: don't break the web . . . . .	23
<b>4</b>	<b>Unicode – a brief introduction</b>	<b>25</b>
4.1	Code points vs. code units . . . . .	25
4.2	Web development: UTF-16 and UTF-8 . . . . .	27
4.3	Grapheme clusters – the real characters . . . . .	28
<b>5</b>	<b>FAQ: JavaScript</b>	<b>29</b>
5.1	How do I find out what JavaScript features are supported where? . . . . .	29
5.2	Why does JavaScript fail silently so often? . . . . .	29
5.3	Why can't we clean up JavaScript, by removing quirks and outdated features? . . . . .	30

<b>II</b>	<b>First steps</b>	<b>31</b>
<b>6</b>	<b>The big picture</b>	<b>33</b>
6.1	What are you learning in this book? . . . . .	33
6.2	The structure of browsers and Node.js . . . . .	34
6.3	Trying out JavaScript code . . . . .	34
6.4	Further reading . . . . .	37
<b>7</b>	<b>Syntax</b>	<b>39</b>
7.1	An overview of JavaScript's syntax . . . . .	40
7.2	(Advanced) . . . . .	43
7.3	Identifiers . . . . .	43
7.4	Statement vs. expression . . . . .	44
7.5	Syntactically ambiguous constructs . . . . .	45
7.6	Semicolons . . . . .	48
7.7	Automatic semicolon insertion (ASI) . . . . .	49
7.8	Semicolons: best practices . . . . .	51
7.9	Strict mode . . . . .	51
<b>8</b>	<b>Assertion API</b>	<b>53</b>
8.1	Assertions in software development . . . . .	53
8.2	How assertions are used in this book . . . . .	53
8.3	Normal comparison versus deep comparison . . . . .	54
8.4	Quick reference: module assert . . . . .	55
<b>9</b>	<b>Getting started with quizzes and exercises</b>	<b>57</b>
9.1	Quizzes . . . . .	57
9.2	Exercises . . . . .	57
9.3	Unit tests in JavaScript . . . . .	58
<b>III</b>	<b>Variables and values</b>	<b>61</b>
<b>10</b>	<b>Variables and assignment</b>	<b>63</b>
10.1	let . . . . .	63
10.2	const . . . . .	64
10.3	Deciding between let and const . . . . .	65
10.4	Variables are block-scoped . . . . .	65
<b>11</b>	<b>Values</b>	<b>67</b>
11.1	What's a type? . . . . .	67
11.2	JavaScript's type hierarchy . . . . .	67
11.3	The types of the language specification . . . . .	68
11.4	Primitive values versus objects . . . . .	69
11.5	Classes and constructor functions . . . . .	71
11.6	Constructor functions associated with primitive types . . . . .	71
11.7	The operators typeof and instanceof: what's the type of a value? . . . . .	71
11.8	Converting between types . . . . .	73
<b>12</b>	<b>Operators</b>	<b>75</b>
12.1	Two important rules for operators . . . . .	75

12.2 The plus operator (+)	76
12.3 Assignment operators	77
12.4 Equality: == versus ===	78
12.5 Ordering operators	80
12.6 Various other operators	80
<b>IV Primitive values</b>	<b>81</b>
<b>13 The non-values undefined and null</b>	<b>83</b>
13.1 undefined vs. null	83
13.2 Occurrences of undefined and null	84
13.3 Checking for undefined or null	85
13.4 undefined and null don't have properties	85
<b>14 Booleans</b>	<b>87</b>
14.1 Converting to boolean	87
14.2 Falsy and truthy values	88
14.3 Conditional operator (? :)	91
14.4 Binary logical operators: And (&&), Or (  )	91
14.5 Logical Not (!)	93
<b>15 Numbers</b>	<b>95</b>
15.1 JavaScript only has floating point numbers	96
15.2 Number literals	96
15.3 Number operators	97
15.4 Converting to number	99
15.5 Error values	99
15.6 Error value: NaN	100
15.7 Error value: Infinity	101
15.8 The precision of numbers: careful with decimal fractions	102
15.9 (Advanced)	102
15.10 Background: floating point precision	102
15.11 Integers in JavaScript	103
15.12 Bitwise operators	105
15.13 Quick reference: numbers	107
<b>16 Math</b>	<b>115</b>
16.1 Data properties	115
16.2 Exponents, roots, logarithms	116
16.3 Rounding	117
16.4 Trigonometric Functions	118
16.5 asm.js helpers	120
16.6 Various other functions	120
16.7 Sources	121
<b>17 Strings</b>	<b>123</b>
17.1 Plain string literals	124
17.2 Accessing characters and code points	124
17.3 String concatenation via +	125

17.4	Converting to string . . . . .	125
17.5	Comparing strings . . . . .	127
17.6	Atoms of text: JavaScript characters, code points, grapheme clusters . . . . .	128
17.7	Quick reference: Strings . . . . .	129
<b>18</b>	<b>Using template literals and tagged templates</b>	<b>139</b>
18.1	Disambiguation: “template” . . . . .	139
18.2	Template literals . . . . .	140
18.3	Tagged templates . . . . .	140
18.4	Raw string literals . . . . .	142
18.5	(Advanced) . . . . .	143
18.6	Multi-line template literals and indentation . . . . .	143
18.7	Simple templating via template literals . . . . .	145
18.8	Further reading . . . . .	146
<b>19</b>	<b>Symbols</b>	<b>147</b>
19.1	Use cases for symbols . . . . .	147
19.2	Publicly known symbols . . . . .	149
19.3	Converting symbols . . . . .	150
19.4	Further reading . . . . .	151
<b>V</b>	<b>Control flow and data flow</b>	<b>153</b>
<b>20</b>	<b>Control flow statements</b>	<b>155</b>
20.1	Controlling loops: break and continue . . . . .	156
20.2	if statements . . . . .	157
20.3	switch statements . . . . .	158
20.4	while loops . . . . .	162
20.5	do-while loops . . . . .	162
20.6	for loops . . . . .	162
20.7	for-of loops . . . . .	164
20.8	for-await-of loops . . . . .	165
20.9	for-in loops (avoid) . . . . .	165
<b>21</b>	<b>Exception handling</b>	<b>167</b>
21.1	Motivation: throwing and catching exceptions . . . . .	167
21.2	throw . . . . .	168
21.3	try-catch-finally . . . . .	169
21.4	Error classes and their properties . . . . .	170
<b>22</b>	<b>Callable values</b>	<b>173</b>
22.1	Kinds of functions . . . . .	173
22.2	Named function expressions . . . . .	177
22.3	Arrow functions . . . . .	178
22.4	Hoisting . . . . .	180
22.5	Returning values from functions . . . . .	180
22.6	Parameter handling . . . . .	181
22.7	Understanding JavaScript’s callable values (advanced) . . . . .	185

<b>23 Variable scopes and closures (advanced)</b>	<b>187</b>
23.1 The scope of a variable . . . . .	188
23.2 Terminology: static versus dynamic . . . . .	188
23.3 Temporal dead zone (TDZ) . . . . .	189
23.4 Hoisting . . . . .	190
23.5 Global variables . . . . .	190
23.6 Ways of declaring variables . . . . .	192
23.7 Variables in detail: environments . . . . .	192
23.8 Recursion via environments . . . . .	193
23.9 Nested scopes via environments . . . . .	194
23.10 Closures . . . . .	197
 <b>VI Modularity</b>	 <b>203</b>
<b>24 Modules</b>	<b>205</b>
24.1 Before modules: scripts . . . . .	205
24.2 Module systems created prior to ES6 . . . . .	206
24.3 ECMAScript modules . . . . .	208
24.4 Named exports . . . . .	209
24.5 Default exports . . . . .	210
24.6 Naming modules . . . . .	211
24.7 Imports are read-only views on exports . . . . .	212
24.8 Module specifiers . . . . .	212
24.9 Syntactic pitfall: importing is not destructuring . . . . .	213
24.10 Preview: loading modules dynamically . . . . .	214
24.11 Further reading . . . . .	215
<b>25 Single objects</b>	<b>217</b>
25.1 The two roles of objects in JavaScript . . . . .	218
25.2 Objects as records . . . . .	219
25.3 Spreading into object literals (...) . . . . .	221
25.4 Methods . . . . .	223
25.5 Objects as dictionaries . . . . .	228
25.6 Standard methods . . . . .	234
25.7 Advanced topics . . . . .	235
<b>26 Prototype chains and classes</b>	<b>237</b>
26.1 Prototype chains . . . . .	238
26.2 Classes . . . . .	243
26.3 Private data for classes . . . . .	247
26.4 Subclassing . . . . .	248
 <b>VII Collections</b>	 <b>255</b>
<b>27 Synchronous iteration</b>	<b>257</b>
27.1 What is synchronous iteration about? . . . . .	257
27.2 Core iteration constructs: iterables and iterators . . . . .	258
27.3 Iterating manually . . . . .	259

27.4	Iteration in practice . . . . .	260
27.5	Quick reference: synchronous iteration . . . . .	261
27.6	Further reading . . . . .	262
<b>28</b>	<b>Arrays (Array)</b>	<b>263</b>
28.1	The two roles of Arrays in JavaScript . . . . .	264
28.2	Basic Array operations . . . . .	264
28.3	for-of and Arrays . . . . .	266
28.4	Array-like objects . . . . .	267
28.5	Converting iterable and Array-like values to Arrays . . . . .	268
28.6	Creating and filling Arrays with arbitrary lengths . . . . .	269
28.7	Multidimensional Arrays . . . . .	270
28.8	More Array features (advanced) . . . . .	270
28.9	Adding and removing elements (destructively and non-destructively) . . . . .	272
28.10	Methods: iteration and transformation (.find(), .map(), .filter(), etc.) . . . . .	274
28.11	.sort(): sorting Arrays . . . . .	280
28.12	Quick reference: Array<T> . . . . .	281
<b>29</b>	<b>Typed Arrays: handling binary data (Advanced)</b>	<b>289</b>
29.1	The basics of the API . . . . .	290
29.2	Foundations of the Typed Array API . . . . .	293
29.3	ArrayBuffers . . . . .	296
29.4	Typed Arrays . . . . .	297
29.5	DataViews . . . . .	304
29.6	Further reading . . . . .	305
<b>30</b>	<b>Maps (Map)</b>	<b>307</b>
30.1	Using Maps . . . . .	308
30.2	Example: Counting characters . . . . .	310
30.3	A few more details about the keys of Maps (advanced) . . . . .	310
30.4	Missing Map operations . . . . .	311
30.5	Quick reference: Map<K, V> . . . . .	313
30.6	FAQ . . . . .	316
<b>31</b>	<b>WeakMaps (WeakMap)</b>	<b>317</b>
31.1	Attaching values to objects via WeakMaps . . . . .	317
31.2	WeakMaps as black boxes . . . . .	318
31.3	Examples . . . . .	318
31.4	WeakMap API . . . . .	320
<b>32</b>	<b>Sets (Set)</b>	<b>321</b>
32.1	Using Sets . . . . .	322
32.2	Comparing Set elements . . . . .	323
32.3	Missing Set operations . . . . .	324
32.4	Quick reference: Set<T> . . . . .	325
<b>33</b>	<b>WeakSets (WeakSet)</b>	<b>329</b>
33.1	Example: Marking objects as safe to use with a method . . . . .	329
33.2	WeakSet API . . . . .	330



<b>34 Destructuring</b>	<b>331</b>
34.1 A first taste of destructuring	331
34.2 Constructing vs. extracting	332
34.3 Where can we destructure?	333
34.4 Object-destructuring	333
34.5 Array-destructuring	335
34.6 Destructuring use case: multiple return values	336
34.7 Not finding a match	337
34.8 What values can't be destructured?	338
34.9 (Advanced)	338
34.10 Default values	339
34.11 Parameter definitions are similar to destructuring	339
34.12 Nesting	340
34.13 Further reading	340
<b>35 Synchronous generators (advanced)</b>	<b>341</b>
35.1 What are synchronous generators?	341
35.2 Calling generators from generators (advanced)	345
35.3 Example: Reusing loops	347
35.4 Advanced features of generators	348
<b>VIII Asynchronicity</b>	<b>349</b>
<b>36 Asynchronous programming in JavaScript</b>	<b>351</b>
36.1 The call stack	351
36.2 The event loop	352
36.3 How to avoid blocking the JavaScript process	353
36.4 Patterns for delivering asynchronous results	355
36.5 Asynchronous code: the downsides	358
36.6 Resources	359
<b>37 Promises for asynchronous programming</b>	<b>361</b>
37.1 Overview	362
37.2 The basics of using Promises	364
37.3 Examples	368
37.4 Promise-based functions start synchronously, settle asynchronously	370
37.5 Error handling: don't mix rejections and exceptions	371
37.6 <code>Promise.all()</code> : working with Arrays of Promises	372
37.7 Tips for using Promises	375
37.8 Further reading	378
<b>38 Async functions</b>	<b>379</b>
38.1 Async functions: the basics	379
38.2 Terminology	383
38.3 <code>await</code> is shallow (you can't use it in callbacks)	383
38.4 (Advanced)	384
38.5 Immediately invoked async arrow functions	384
38.6 Concurrency and <code>await</code>	385
38.7 Tips for using async functions	386

<b>39 Asynchronous iteration</b>	<b>389</b>
39.1 Basic asynchronous iteration . . . . .	389
39.2 Asynchronous generators . . . . .	392
39.3 Async iteration over Node.js streams . . . . .	396

## **IX More standard library 399**

<b>40 Regular expressions (RegExp)</b>	<b>401</b>
40.1 Creating regular expressions . . . . .	402
40.2 Syntax . . . . .	403
40.3 Flags . . . . .	406
40.4 Properties of regular expression objects . . . . .	409
40.5 Methods for working with regular expressions . . . . .	410
40.6 Flag /g and its pitfalls . . . . .	414
40.7 Techniques for working with regular expressions . . . . .	417

<b>41 Dates (Date)</b>	<b>419</b>
41.1 Best practice: don't use the current built-in API . . . . .	419
41.2 Background: UTC vs. GMT . . . . .	420
41.3 Background: date time formats (ISO) . . . . .	420
41.4 Time values . . . . .	421
41.5 Creating Dates . . . . .	422
41.6 Getters and setters . . . . .	422
41.7 Converting Dates to strings . . . . .	423
41.8 Pitfalls of the Date API . . . . .	424

<b>42 Creating and parsing JSON (JSON)</b>	<b>425</b>
42.1 The discovery and standardization of JSON . . . . .	426
42.2 JSON syntax . . . . .	426
42.3 Using the JSON API . . . . .	427
42.4 Configuring what is stringified or parsed (advanced) . . . . .	429
42.5 FAQ . . . . .	432
42.6 Quick reference: JSON . . . . .	432

## **X Miscellaneous topics 435**

<b>43 Next steps: an overview of the web development landscape (bonus)</b>	<b>437</b>
43.1 Tips against feeling overwhelmed . . . . .	437
43.2 Things worth learning for web development . . . . .	438
43.3 Example: a tool-based JavaScript workflow . . . . .	438
43.4 An overview of JavaScript tools . . . . .	441
43.5 Tools not related to JavaScript . . . . .	443
43.6 Further reading on JavaScript . . . . .	443

## **XI Appendices 445**

<b>A Index</b>	<b>447</b>
----------------	------------

# **Part I**

## **Background**



# Chapter 1

## About this book

### Contents

---

1.1 What's in this book? . . . . .	13
1.2 What is not covered by this book? . . . . .	13
1.3 This book isn't finished, yet . . . . .	14
1.4 Can I buy a print edition? . . . . .	14
1.5 Will there be a free online version? . . . . .	14
1.6 Acknowledgements . . . . .	14

---

### 1.1 What's in this book?

Goal of this book: make JavaScript less challenging to learn for newcomers, by offering a modern view that is as consistent as possible.

Highlights:

- The book covers all essential features of the language, up to and including ECMAScript 2018 (the current standard).
  - The focus on modern JavaScript means that there are many features you don't have to learn initially – maybe ever (e.g., `var`).
  - These features are still mentioned, along with pointers to documentation.
- There are test-driven exercises and quizzes for most chapters (paid feature).
- Includes advanced sections, so you can occasionally dig deeper – if you want to.

No prior knowledge of JavaScript is required, but you should know how to program.

### 1.2 What is not covered by this book?

- Some advanced language features are not explained, but references to appropriate material are provided. For example, to my other JavaScript books at [ExploringJS.com](http://exploringjs.com)<sup>1</sup>, which are free to read

---

<sup>1</sup><http://exploringjs.com/>

online.

- This book deliberately focuses on the language. Browser-only features etc. are not included.

### 1.3 This book isn't finished, yet

More content is still to come. Buy this book now and get free updates for at least 2 years!

### 1.4 Can I buy a print edition?

The current plan is to release a print edition once the book is finished. Rough guess: late 2019.

### 1.5 Will there be a free online version?

There will eventually be a version that is free to read online (with most of the chapters, but without exercises and quizzes). Current estimate: late 2019.

### 1.6 Acknowledgements

- Cover by Fran Caye<sup>2</sup>.
- Thanks for reviewing:
  - Johannes Weber (@jowe<sup>3</sup>)

[Generated: 2019-02-18 13:17]

---

<sup>2</sup><http://francaye.net>

<sup>3</sup><https://twitter.com/jowe>

## Chapter 2

# FAQ: book

### Contents

2.1	What are the “advanced” chapters and sections about? . . . . .	15
2.2	What should I read if I’m <i>really</i> impatient? . . . . .	15
2.3	How do I submit feedback and corrections? . . . . .	16
2.4	How do I get updates for the downloads at Payhip? . . . . .	16
2.5	I’m occasionally seeing type annotations – how do those work? . . . . .	16
2.6	What do the notes with icons mean? . . . . .	16

This chapter answers questions you may have and gives tips for reading this book.

### 2.1 What are the “advanced” chapters and sections about?

Several chapters and sections are marked as “advanced”. The idea is that you can initially skip them. That is, you can get a quick working knowledge of JavaScript by only reading the basic (non-advanced) content.

As your knowledge evolves, you can later come back to some or all of the advanced content.

### 2.2 What should I read if I’m *really* impatient?

Do the following:

- Start reading with chapter “**The big picture**”.
- Skip all chapters and sections marked as “advanced”, and all quick references.

Then this book should be a fairly quick read.

## 2.3 How do I submit feedback and corrections?

The HTML version of this book (online, or ad-free archive in paid version) has a link at the end of each chapter that enables you to give feedback.

## 2.4 How do I get updates for the downloads at Payhip?

- The receipt email for the purchase includes a link. You'll always be able to download the latest version of the files at that location.
- If you opted into emails while buying, then you'll get an email whenever there is new content. To opt in later, you must contact Payhip (see bottom of [payhip.com](https://payhip.com)).

## 2.5 I'm occasionally seeing type annotations – how do those work?

For example, you may see:

```
Number.isFinite(num: number): boolean
```

The type annotations `: number` and `: boolean` are not real JavaScript. They are a notation for static typing, borrowed from the compile-to-JavaScript language TypeScript (which is mostly just JavaScript plus static typing).

Why is this notation being used? It helps give you a quick idea of how a function works.

The type notation is explained in detail in a blog post<sup>1</sup> on 2ality.

## 2.6 What do the notes with icons mean?



### Reading

Explains how to best read the content or points to additional reading elsewhere (in the book or externally).



### Tip

Gives tips related to the current content.



### Exercise

Mentions the path of a test-driven exercise that you can do at that point.

---

<sup>1</sup><http://2ality.com/2018/04/type-notation-typescript.html>





### Quiz

Indicates that there is a quiz for the current (part of a) chapter.



## Chapter 3

# History and evolution of JavaScript

### Contents

---

3.1	How JavaScript was created . . . . .	19
3.2	Standardization . . . . .	20
3.3	Timeline of ECMAScript versions . . . . .	20
3.4	Ecma Technical Committee 39 (TC39) . . . . .	21
3.5	The TC39 process . . . . .	21
3.5.1	Tip: think in individual features and stages, not ECMAScript versions . . . . .	21
3.6	FAQ: TC39 process . . . . .	23
3.6.1	How is [my favorite proposed feature] doing? . . . . .	23
3.6.2	Is there an official list of ECMAScript features? . . . . .	23
3.7	Evolving JavaScript: don't break the web . . . . .	23

---

### 3.1 How JavaScript was created

JavaScript was created in May 1995, in 10 days, by Brendan Eich. Eich worked at Netscape and implemented JavaScript for their web browser, *Netscape Navigator*.

The idea was that major interactive parts of the client-side web were to be implemented in Java. JavaScript was supposed to be a glue language for those parts and to also make HTML slightly more interactive. Given its role of assisting Java, JavaScript had to look like Java. That ruled out existing solutions such as Perl, Python, TCL and others.

Initially, JavaScript's name changed frequently:

- Its code name was *Mocha*.
- In the Netscape Navigator 2.0 betas (September 1995), it was called *LiveScript*.
- In Netscape Navigator 2.0 beta 3 (December 1995), it got its final name, *JavaScript*.

## 3.2 Standardization

There are two standards for JavaScript:

- ECMA-262 is hosted by Ecma International. It is the primary standard.
- ISO/IEC 16262 is hosted by the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC). This is a secondary standard.

The language described by these standards is called *ECMAScript*, not *JavaScript*. A different name was chosen, because Sun (now Oracle) had a trademark for the latter name. The “ECMA” in “ECMAScript” comes from the organization that hosts the primary standard.

The original name of that organization was *ECMA*, an acronym for *European Computer Manufacturers Association*. It was later changed to *Ecma International* (where *Ecma* is not an acronym, anymore), as “European” didn’t reflect the organization’s global activities. The initial all-caps acronym explains the spelling of *ECMAScript*.

In principle, JavaScript and ECMAScript mean the same thing. Sometimes, the following distinction is made:

- The term *JavaScript* refers to the language and its implementations.
- The term *ECMAScript* refers to the language standard and language versions.

Therefore, *ECMAScript 6* is a version of the language (its 6th edition).

## 3.3 Timeline of ECMAScript versions

This is a brief timeline of ECMAScript versions:

- ECMAScript 1 (June 1997): First version of the standard.
- ECMAScript 2 (June 1998): Small update, to keep ECMA-262 in sync with the ISO standard.
- ECMAScript 3 (December 1999): Adds many core features – “[...] regular expressions, better string handling, new control statements [do-while, switch], try / catch exception handling, [...]”
- ECMAScript 4 (abandoned in July 2008): Would have been a massive upgrade (with static typing, modules, namespaces and more), but ended up being too ambitious and dividing the language’s stewards. Therefore, it was abandoned.
- ECMAScript 5 (December 2009): Brought minor improvements – a few standard library features and *strict mode*.
- ECMAScript 5.1 (June 2011): Another small update to keep Ecma and ISO standards in sync.
- ECMAScript 6 (June 2015): A large update that fulfilled many of the promises of ECMAScript 4. This version is the first one whose official name – *ECMAScript 2015* – is based on the year of publication.
- ECMAScript 2016 (June 2016): First yearly release. That resulted in fewer new features per release – compared to ES6, which was a large upgrade.
- ECMAScript 2017 (June 2017)
- ECMAScript 2018 (June 2018)

## 3.4 Ecma Technical Committee 39 (TC39)

TC39 is the committee that evolves JavaScript. Its members are, strictly speaking, companies: Adobe, Apple, Facebook, Google, Microsoft, Mozilla, Opera, Twitter, and others. That is, companies that are usually fierce competitors are working together for the good of the language.

## 3.5 The TC39 process

With ECMAScript 6, two issues with the release process used at that time became obvious:

- If too much time passes between releases then features that are ready early, have to wait a long time until they can be released. And features that are ready late, risk being rushed to make the deadline.
- Features were often designed long before they were implemented and used. Design deficiencies related to implementation and use were therefore discovered too late.

In response to these issues, TC39 instituted a new process that was named *TC39 process*:

- ECMAScript features are designed independently and go through stages, starting at 0 (“straw-man”), ending at 4 (“finished”).
- Especially the later stages require prototype implementations and real-world testing, leading to feedback loops between designs and implementations.
- ECMAScript versions are released once per year and include all features that have reached stage 4 prior to a release deadline.

The result: smaller, incremental releases, whose features have already been field-tested. Fig. 3.1 illustrates the TC39 process.

ES2016 was the first ECMAScript version that was designed according to the TC39 process.

For more information on the TC39 process, consult “Exploring ES2018 and ES2019<sup>1</sup>”.

### 3.5.1 Tip: think in individual features and stages, not ECMAScript versions

Up to and including ES6, it was most common to think about JavaScript in terms of ECMAScript versions. E.g., “Does this browser support ES6, yet?”

Starting with ES2016, it’s better to think in individual features: Once a feature reaches stage 4, you can safely use it (if it’s supported by the JavaScript engines you are targeting). You don’t have to wait until the next ECMAScript release.

---

<sup>1</sup>[http://exploringjs.com/es2018-es2019/ch\\_tc39-process.html](http://exploringjs.com/es2018-es2019/ch_tc39-process.html)

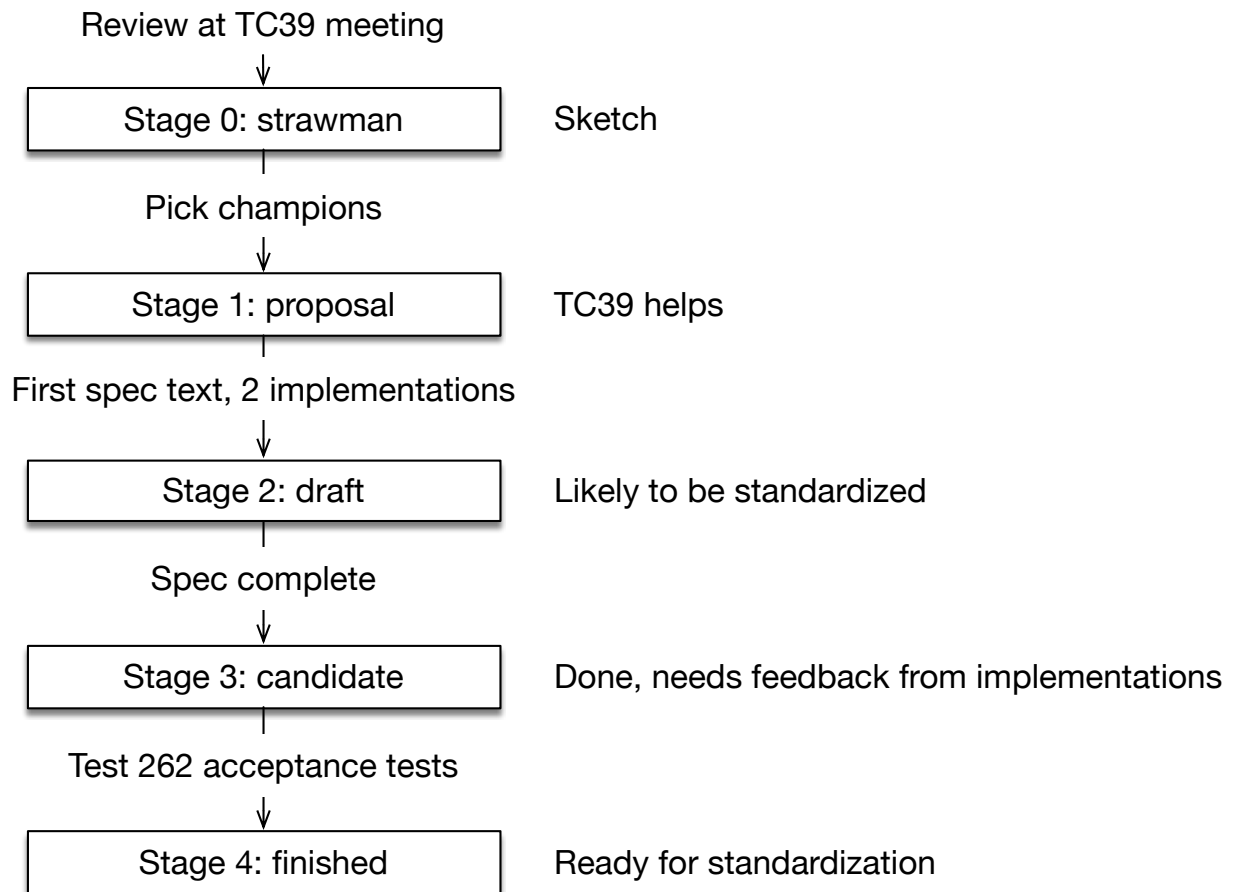


Figure 3.1: Each ECMAScript feature proposal goes through stages that are numbered from 0 to 4. *Champions* are TC39 members that support the authors of a feature. Test 262 is a suite of tests that checks JavaScript engines for compliance with the language specification.

## 3.6 FAQ: TC39 process

### 3.6.1 How is [my favorite proposed feature] doing?

If you are wondering what stages various proposed features are in, consult the readme of the ECMA-262 GitHub repository<sup>2</sup>.

### 3.6.2 Is there an official list of ECMAScript features?

Yes, the TC39 repo lists finished proposals<sup>3</sup> and mentions in which ECMAScript versions they are introduced.

## 3.7 Evolving JavaScript: don't break the web

One idea that occasionally comes up, is to clean up JavaScript, by removing old features and quirks. While the appeal of that idea is obvious, it has significant downsides.

Let's assume we create a new version of JavaScript that is not backward compatible and fixes all of its flaws. As a result, we'd encounter the following problems:

- JavaScript engines become bloated: they need to support both the old and the new version. The same is true for tools such as IDEs and build tools.
- Programmers need to know, and be continually conscious of, the differences between the versions.
- You can either migrate all of an existing code base to the new version (which can be a lot of work). Or you can mix versions and refactoring becomes harder, because can't move code between versions without changing it.
- You somehow have to specify per piece of code – be it a file or code embedded in a web page – what version it is written in. Every conceivable solution has pros and cons. For example, *strict mode* is a slightly cleaner version of ES5. One of the reasons why it wasn't as popular as it should have been: it was a hassle to opt in via a directive at the beginning of a file or function.

So what is the solution? Can we have our cake and eat it? The approach that was chosen for ES6 is called "One JavaScript":

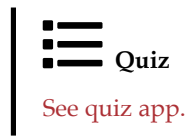
- New versions are always completely backward compatible (but there may occasionally be minor, hardly noticeable clean-ups).
- Old features aren't removed or fixed. Instead, better versions of them are introduced. One example is declaring variables via `let` – which is an improved version of `var`.
- If aspects of the language are changed, it is done so inside new syntactic constructs. That is, you opt in implicitly. For example, `yield` is only a keyword inside generators (which were introduced in ES6). And all code inside modules and classes (both introduced in ES6) is implicitly in strict mode.

For more information on One JavaScript, consult "Exploring ES6"<sup>4</sup>.

<sup>2</sup><https://github.com/tc39/ecma262/blob/master/README.md>

<sup>3</sup><https://github.com/tc39/proposals/blob/master/finished-proposals.md>

<sup>4</sup>[http://exploringjs.com/es6/ch\\_one-javascript.html](http://exploringjs.com/es6/ch_one-javascript.html)





## Chapter 4

# Unicode – a brief introduction

### Contents

<b>4.1 Code points vs. code units</b> . . . . .	<b>25</b>
4.1.1 Code points . . . . .	25
4.1.2 Encoding formats for code units: UTF-32, UTF-16, UTF-8 . . . . .	26
<b>4.2 Web development: UTF-16 and UTF-8</b> . . . . .	<b>27</b>
4.2.1 Source code internally: UTF-16 . . . . .	27
4.2.2 Strings: UTF-16 . . . . .	27
4.2.3 Source code in files: UTF-8 . . . . .	27
<b>4.3 Grapheme clusters – the real characters</b> . . . . .	<b>28</b>

Unicode is a standard for representing and managing text in most of the world’s writing systems. Virtually all modern software that works with text, supports Unicode. The standard is maintained by the Unicode Consortium. A new version of the standard is published every year (with new Emojis etc.). Unicode 1 was published in 1991.

## 4.1 Code points vs. code units

Two concepts are crucial for understanding Unicode:

- Code points: are numbers that represent Unicode characters.
- Code units: are pieces of data with fixed sizes. One or more code units encode a single code point. The size of code units depends on the encoding format. The most popular format, UTF-8, has 8-bit code units.

### 4.1.1 Code points

The first version of Unicode had 16-bit code points. Since then, the number of characters has grown considerably and the size of code points was extended to 21 bits. These 21 bits are partitioned in 17 planes, with 16 bits each:

- Plane 0: **Basic Multilingual Plane (BMP)**, 0x0000–0xFFFF
  - This is the most frequently used plane. Roughly, it comprises the original Unicode.
- Plane 1: Supplementary Multilingual Plane (SMP), 0x10000–0x1FFFF
- Plane 2: Supplementary Ideographic Plane (SIP), 0x20000–0x2FFFF
- Plane 3–13: Unassigned
- Plane 14: Supplementary Special-Purpose Plane (SSP), 0xE0000–0xEFFFF
- Plane 15–16: Supplementary Private Use Area (S PUA A/B), 0xF0000–0x10FFFF

Planes 1–16 are called supplementary planes or **astral planes**.

Let’s check the code points of a few characters:

```
> 'A'.codePointAt(0).toString(16)
'41'
> 'ü'.codePointAt(0).toString(16)
'fc'
> 'π'.codePointAt(0).toString(16)
'3c0'
> '🍷'.codePointAt(0).toString(16)
'1f642'
```

The hexadecimal number of the code points tells us that the first three characters reside in plane 0 (within 16 bits), while the emoji resides in plane 1.

## 4.1.2 Encoding formats for code units: UTF-32, UTF-16, UTF-8

Let’s cover three ways of encoding code points as code units.

### 4.1.2.1 UTF-32 (Unicode Transformation Format 32)

UTF-32 uses 32 bits to store code units, resulting in one code unit per code point. This format is the only one with *fixed-length encoding* (all others use a varying number of code units to encode a single code point).

### 4.1.2.2 UTF-16 (Unicode Transformation Format 16)

UTF-16 uses 16-bit code units. It encodes code points as follows:

- BMP (first 16 bits of Unicode): are stored in single code units.
- Astral planes: After subtracting the BMP’s count of 0x10000 characters from Unicode’s count of 0x110000 characters, 0x100000 characters (20 bits) remain. These are stored in unoccupied “holes” in the BMP:
  - Most significant 10 bits (*leading surrogate*): 0xD800–0xDBFF
  - Least significant 10 bits (*trailing surrogate*): 0xDC00–0xDFFF

As a consequence, by looking at a UTF-16 code unit, we can tell if it is a BMP character, the first part (leading surrogate) of an astral plane character or the last part (trailing surrogate) of an astral plane character.

### 4.1.2.3 UTF-8 (Unicode Transformation Format 8)

UTF-8 has 8-bit code units. It uses 1–4 code units to encode a code point:

Code points	Code units
0000–007F	0xxxxxxx (7 bits)
0080–07FF	110xxxxx, 10xxxxxx (5+6 bits)
0800–FFFF	1110xxxx, 10xxxxxx, 10xxxxxx (4+6+6 bits)
10000–1FFFFF	11110xxx, 10xxxxxx, 10xxxxxx, 10xxxxxx (3+6+6+6 bits)

Notes:

- The bit prefix of each code unit tells us:
  - Is it first in a series of code units? If yes, how many code units will follow?
  - Is it second or later in a series of code units?
- The character mappings in the 0000–007F range are the same as ASCII, which leads to a degree of backward-compatibility with older software.

## 4.2 Web development: UTF-16 and UTF-8

For web development, two Unicode encoding formats are relevant: UTF-16 and UTF-8.

### 4.2.1 Source code internally: UTF-16

The ECMAScript specification internally represents source code as UTF-16.

### 4.2.2 Strings: UTF-16

The characters in JavaScript strings are UTF-16 code units:

```
> const smiley = '😊';
> smiley.length
2
> smiley === '\uD83D\uDE42' // code units
true
> smiley === '\u{1F642}' // code point
true
```

For more information on Unicode and strings, consult [the section on atoms of text](#) in the chapter on strings.

### 4.2.3 Source code in files: UTF-8

When JavaScript is stored in `.html` and `.js` files, the encoding is almost always UTF-8, these days:

```
<!doctype html>
<html>
<head>
  <meta charset="UTF-8">
...
```

### 4.3 Grapheme clusters – the real characters

The concept of a character becomes remarkably complex, once you consider many of the world’s writing systems.

On one hand, code points can be said to represent Unicode “characters”.

On the other hand, there are *grapheme clusters*. A grapheme cluster corresponds most closely to a symbol displayed on screen or paper. It is defined as “a horizontally segmentable unit of text”. One or more code points are needed to encode a grapheme cluster.

For example, one emoji of a family is composed of 7 code points – 4 of them are graphemes themselves and they are joined by invisible code points:

```
> [...'👨👩👧👦']
[ '👨', '👩', '👧', '👦' ]
```

Another example is flag emojis:

```
> [...'🇯🇵']
[ '🇯', '🇵' ]
> '🇯🇵'.length
4
```



#### Reading: More information on grapheme clusters

For more information, consult “Let’s Stop Ascribing Meaning to Code Points<sup>a</sup>” by Manish Gore-gaokar.

<sup>a</sup><https://manishearth.github.io/blog/2017/01/14/stop-ascribing-meaning-to-unicode-code-points/>

# Chapter 5

## FAQ: JavaScript

### Contents

5.1	How do I find out what JavaScript features are supported where? . . . . .	29
5.2	Why does JavaScript fail silently so often? . . . . .	29
5.3	Why can't we clean up JavaScript, by removing quirks and outdated features? . . . .	30

### 5.1 How do I find out what JavaScript features are supported where?

The book usually mentions if a feature is part of ECMAScript 5 (as required by older browsers) or a newer version. For for detailed information (incl. pre-ES5 versions), there are several good compatibility tables available online:

- ECMAScript compatibility tables for various engines<sup>1</sup> by kangax<sup>2</sup>, webbedspace<sup>3</sup>, zloirock<sup>4</sup>
- Node.js compatibility tables<sup>5</sup> by William Kapke<sup>6</sup>
- Mozilla's MDN web docs<sup>7</sup> have tables for each feature that describe relevant ECMAScript versions and browser support.

### 5.2 Why does JavaScript fail silently so often?

JavaScript often fails silently. Let's look at two examples.

First example: If the operands of an operator don't have the appropriate types, they are converted as necessary.

---

<sup>1</sup><http://kangax.github.io/compat-table/>

<sup>2</sup><https://twitter.com/kangax>

<sup>3</sup><https://twitter.com/webbedspace>

<sup>4</sup><https://twitter.com/zloirock>

<sup>5</sup><https://node.green>

<sup>6</sup><https://twitter.com/williamkapke>

<sup>7</sup><https://developer.mozilla.org/en-US/docs/Web/JavaScript>

```
> '3' * '5'  
15
```

Second example: If an arithmetic computation fails, you get an error value, not an exception.

```
> 1 / 0  
Infinity
```

**Why is that?**

The reason is historical: JavaScript did not have exceptions until ECMAScript 3. Since then, its designers have tried to avoid silent failures.

### 5.3 Why can't we clean up JavaScript, by removing quirks and outdated features?

The chapter on the history and evolution of JavaScript has [a section](#) that answers this question.

## **Part II**

### **First steps**





## Chapter 6

# The big picture

### Contents

<b>6.1 What are you learning in this book?</b>	<b>33</b>
<b>6.2 The structure of browsers and Node.js</b>	<b>34</b>
6.2.1 The console	34
<b>6.3 Trying out JavaScript code</b>	<b>34</b>
6.3.1 Browser consoles	34
6.3.2 The Node.js REPL	36
6.3.3 Other options	36
<b>6.4 Further reading</b>	<b>37</b>

In this chapter, I'd like to paint the big picture: What are you learning in this book and how does it fit into the overall landscape of web development?

### 6.1 What are you learning in this book?

This book teaches the JavaScript language. It focuses on just the language, but offers occasional glimpses at two platforms where JavaScript can be used:

- Web browsers
- Node.js

Node.js is important for web development in three ways:

- You can use it to write server-side software in JavaScript.
- You can also use it to write software for the command line (think Unix shell, Windows PowerShell, etc.). Many JavaScript-related tools are based on (and executed via) Node.js.
- Node's package manager, npm, has become the dominant way of installing tools (such as compilers and build tools) and libraries – even for client-side development.

## 6.2 The structure of browsers and Node.js

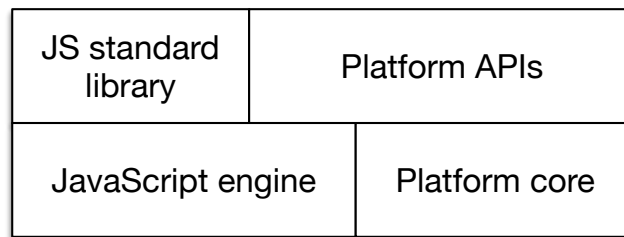


Figure 6.1: The structure of the two JavaScript platforms *web browser* and *Node.js*.

The structures of the two JavaScript platforms *web browser* and *Node.js* are similar (fig. 6.1):

- The JavaScript engine runs JavaScript code.
- The JavaScript standard library is part of JavaScript proper and runs on top of the engine.
- Platform APIs are also available from JavaScript – they provide access to platform-specific functionality. For example:
  - In browsers, you need to use platform-specific APIs if you want to do anything related to the user interface: react to mouse clicks, play sound, etc.
  - In Node.js, platform-specific APIs let you read and write files, download data via HTTP, etc.

### 6.2.1 The console

One interesting example is the JavaScript operation for printing information:

```
console.log('This text is shown on the "console"!');
```

`console.log()` is not part of JavaScript proper. That is, it is part of a platform API, but supported by both browsers and Node.js:

- On browsers, the console is a pane with text that is usually hidden, but can be brought up.
- On Node.js, anything you log to the console is printed on the command line (think `stdout`).

## 6.3 Trying out JavaScript code

You have many options for quickly running pieces of JavaScript. The following subsections describe a few of them.

### 6.3.1 Browser consoles

The consoles of browsers also let you input code. They are JavaScript command lines. How to open the console differs from browser to browser. Fig. 6.2 shows the console of Google Chrome.

To find out how to open the console in your web browser, you can do a web search for “console «name-of-your-browser»”. These are pages for some commonly used web browsers:

- Apple Safari<sup>1</sup>
- Google Chrome<sup>2</sup>
- Microsoft Edge<sup>3</sup>
- Mozilla Firefox<sup>4</sup>

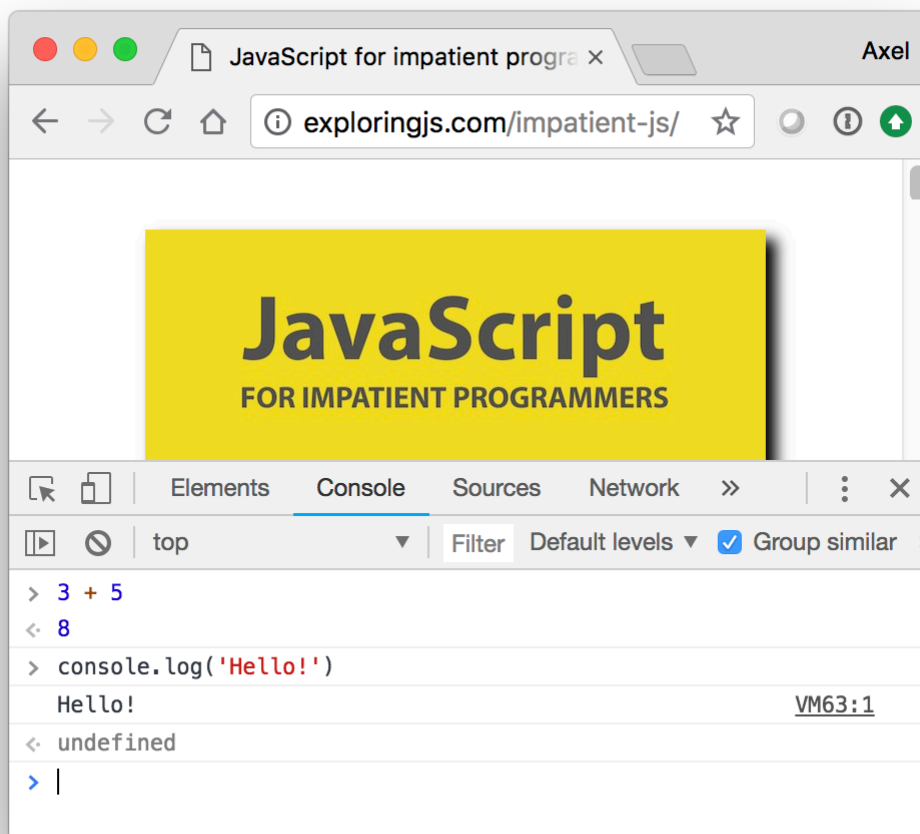


Figure 6.2: The console of the web browser “Google Chrome” is open while visiting a web page.

<sup>1</sup><https://developer.apple.com/safari/tools/>

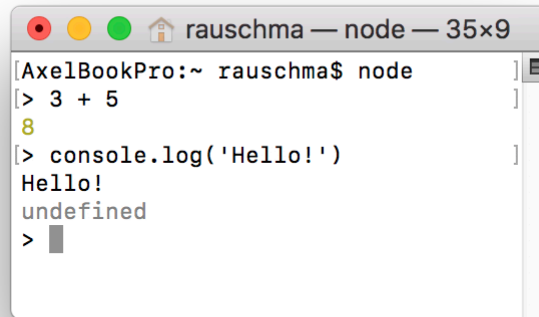
<sup>2</sup><https://developers.google.com/web/tools/chrome-devtools/console/>

<sup>3</sup><https://docs.microsoft.com/en-us/microsoft-edge/devtools-guide/console>

<sup>4</sup>[https://developer.mozilla.org/en-US/docs/Tools/Web\\_Console/Opening\\_the\\_Web\\_Console](https://developer.mozilla.org/en-US/docs/Tools/Web_Console/Opening_the_Web_Console)

### 6.3.2 The Node.js REPL

REPL stands for *read-eval-print loop* and basically means *command line*. To use it, you must first start Node.js from an operating system command line, via the command `node`. Then an interaction with it looks as depicted in fig. 6.3: The text after `>` is input from the user; everything else is output from Node.js.



```
AxelBookPro:~ rauschma$ node
[> 3 + 5
8
]> console.log('Hello!')
Hello!
undefined
>
```

Figure 6.3: Starting and using the Node.js REPL (interactive command line).



#### Reading: REPL interactions

I occasionally demonstrate JavaScript via REPL interactions. Then I also use greater-than symbols (`>`) to mark input. For example:

```
> 3 + 5
8
```

### 6.3.3 Other options

Other options include:

- There are many web apps that let you experiment with JavaScript in web browsers. For example, Babel's REPL<sup>5</sup>.
- There are also native apps and IDE plugins for running JavaScript.

---

<sup>5</sup><https://babeljs.io/repl>

## 6.4 Further reading

- The chapter “**Next steps**” at the end of this book, provides a more comprehensive look at web development.



# Chapter 7

## Syntax

### Contents

---

<b>7.1 An overview of JavaScript's syntax</b>	<b>40</b>
7.1.1 Basic syntax	40
7.1.2 Modules	42
7.1.3 Legal variable and property names	42
7.1.4 Capitalization of names	42
7.1.5 Where to put semicolons?	43
<b>7.2 (Advanced)</b>	<b>43</b>
<b>7.3 Identifiers</b>	<b>43</b>
7.3.1 Valid identifiers (variable names etc.)	43
7.3.2 Reserved words	44
<b>7.4 Statement vs. expression</b>	<b>44</b>
7.4.1 What is allowed where?	45
<b>7.5 Syntactically ambiguous constructs</b>	<b>45</b>
7.5.1 Function declaration vs. function expression	45
7.5.2 Object literal vs. block	46
7.5.3 Disambiguation	46
7.5.4 Example: Immediately Invoked Function Expression (IIFE)	46
7.5.5 Example: immediate method call	47
7.5.6 Example: destructuring via an object pattern	48
7.5.7 Example: an arrow function that returns an object literal	48
<b>7.6 Semicolons</b>	<b>48</b>
7.6.1 Rule of thumb for semicolons	48
7.6.2 Semicolons: control statements	49
<b>7.7 Automatic semicolon insertion (ASI)</b>	<b>49</b>
7.7.1 ASI triggered unexpectedly	49
7.7.2 ASI unexpectedly not triggered	50
<b>7.8 Semicolons: best practices</b>	<b>51</b>
<b>7.9 Strict mode</b>	<b>51</b>
7.9.1 Switching on strict mode	52
7.9.2 Example: strict mode in action	52

---

## 7.1 An overview of JavaScript's syntax

### 7.1.1 Basic syntax

Comments:

```
// single-line comment
```

```
/*  
Comment with  
multiple lines  
*/
```

Primitive (atomic) values:

```
// Booleans
```

```
true  
false
```

```
// Numbers (JavaScript only has one type for numbers)
```

```
-123  
1.141
```

```
// Strings (JavaScript has no type for characters)
```

```
'abc'  
"abc"
```

Checking and logging to the console:

```
// "Asserting" (checking) the expected result of an expression  
// (a method call with 2 parameters).  
// Assertions are a Node.js API that is explained in the next chapter.  
assert.equal(7 + 1, 8);
```

```
// Printing a value to standard out (another method call)
```

```
console.log('Hello!');
```

```
// Printing an error message to standard error
```

```
console.error('Something went wrong!');
```

Declaring variables:

```
let x; // declare x (mutable)
```

```
x = 3 * 5; // assign a value to x
```

```
let y = 3 * 5; // declare and assign
```

```
const z = 8; // declare z (immutable)
```

Control flow statements:

```
// Conditional statement
```

```
if (x < 0) { // is x less than zero?
```



```

    x = -x;
}

```

Ordinary function declarations:

```

// add1() has the parameters a and b
function add1(a, b) {
    return a + b;
}
// Calling function add1()
assert.equal(add1(5, 2), 7);

```

Arrow function expressions (used especially for arguments of function or method calls):

```

// The body of add2 is an expression:
const add2 = (a, b) => a + b;
// Calling function add2()
assert.equal(add2(5, 2), 7);

// The body of add3 is a code block:
const add3 = (a, b) => { return a + b };

```

Objects:

```

// Create plain object via object literal
const obj = {
    first: 'Jane', // property
    last: 'Doe', // property
    getFullName() { // property (method)
        return this.first + ' ' + this.last;
    },
};

// Get a property value
assert.equal(obj.first, 'Jane');
// Set a property value
obj.first = 'Janey';

// Call the method
assert.equal(obj.getFullName(), 'Janey Doe');

```

Arrays (Arrays are also objects):

```

// Creating an Array via an Array literal
const arr = ['a', 'b', 'c'];

// Get Array element
assert.equal(arr[1], 'b');
// Set Array element
arr[1] = 'β';

```

### 7.1.2 Modules

Each module is a single file. Consider, for example, the following two files with modules in them:

file-tools.js

main.js

The module in file-tools.js exports its function `isTextFilePath()`:

```
export function isTextFilePath(filePath) {
  return str.endsWith('.txt');
}
```

The module in main.js imports the whole module path and the function `isTextFilePath()`:

```
// Import whole module as namespace object `path`
import * as path from 'path';
// Import a single export of module file-tools.js
import {isTextFilePath} from './file-tools.js';
```

### 7.1.3 Legal variable and property names

The grammatical category of variable names and property names is called *identifier*.

Identifiers are allowed to have the following characters:

- Unicode letters: A–Z, a–z (etc.)
- \$, \_
- Unicode digits: 0–9 (etc.)
  - Variable names can't start with a digit

Some words have special meaning in JavaScript and are called *reserved*. Examples include: `if`, `true`, `const`.

Reserved words can't be used as variable names:

```
const if = 123;
// SyntaxError: Unexpected token if
```

But they are allowed as names of properties:

```
> const obj = { if: 123 };
> obj.if
123
```

### 7.1.4 Capitalization of names

Lowercase:

- Functions, variables: `myFunction`
- Methods: `obj.myMethod`
- CSS:
  - CSS entity: `special-class`
  - JS variable: `specialClass`

- Labels (break, continue): `my_label`
- Modules (imported as namespaces): `myModule` (exact style varies)

Uppercase:

- Classes: `MyConstructor`
- Constants: `MY_CONSTANT`

### 7.1.5 Where to put semicolons?

At the end of a statement:

```
const x = 123;
func();
```

But not if that statement ends with a curly brace:

```
while (false) {
  // ...
} // no semicolon

function func() {
  // ...
} // no semicolon
```

However, adding a semicolon after such a statement is not a syntax error – it is interpreted as an empty statement:

```
// Function declaration followed by empty statement:
function func() {
  // ...
};
```



Quiz: basic

See quiz app.

## 7.2 (Advanced)

All remaining sections of this chapter are advanced.

## 7.3 Identifiers

### 7.3.1 Valid identifiers (variable names etc.)

First character:

- Unicode letter (including accented characters such as é and ü and characters from non-latin alphabets, such as α)
- \$
- \_

Subsequent characters:

- Legal first characters
- Unicode digits (including Eastern Arabic numerals)
- Some other Unicode marks and punctuations

Examples:

```
const ε = 0.0001;
const строка = '';
let _tmp = 0;
const $foo2 = true;
```

### 7.3.2 Reserved words

Reserved words can't be variable names, but they can be property names. They are:

```
await break case catch class const continue debugger default delete do else export extends finally for function if import in instanceof let new return static super switch this throw try typeof var void while with yield
```

The following words are also reserved, but not used in the language, yet:

```
enum implements package protected interface private public
```

These words are not technically reserved, but you should avoid them, too, because they effectively are keywords:

```
Infinity NaN undefined, async
```

It is also a good idea to avoid the names of global variables (String, Math, etc.).

## 7.4 Statement vs. expression

*Statement* and *expression* are categories for syntactic constructs. That is, they split JavaScript's syntax into two kinds of constructs. (In this book – for the sake of simplicity – we pretend that there are only statements and expressions in JavaScript.) How do they differ?

First, statements “do something”. For example, `if` is a statement:

```
let myStr;
if (myBool) {
  myStr = 'Yes';
} else {
  myStr = 'No';
}
```

Second, expressions are evaluated. They produce values. For example, the code between the parentheses is an expression:

```
let myStr = (myBool ? 'Yes' : 'No');
```

The operator `_ ? _ : _` between the parentheses is called the *ternary operator*. It is the expression version of the `if` statement.

### 7.4.1 What is allowed where?

The current location within JavaScript source code determines which kind of syntactic constructs you are allowed to use:

- The body of a function must be a sequence of statements.
- The arguments of a function call must be expressions.

However, expressions can be used as statements. Then they are called *expression statements*. The opposite is not true: when the context requires an expression, you can't use statements.

The following is an example of a function `foo()` whose body contains three expression statements:

```
function foo() {  
  3 + 5;  
  'hello world';  
  bar();  
}
```

The first two expression statements don't do anything (as their results are ignored). The last expression statement may or may not do something – depending on whether it has side effects.

The following code demonstrates that any expression `bar()` can be either expression or statement – it depends on the context:

```
console.log(bar()); // bar() is expression  
bar(); // bar() is (expression) statement
```

## 7.5 Syntactically ambiguous constructs

JavaScript has several programming constructs that are syntactically ambiguous: They are different depending on whether they are used in statement context or in expression context. This section explores the phenomenon and its consequences.

### 7.5.1 Function declaration vs. function expression

A *function declaration* is a statement:

```
function id(x) {  
  return x;  
}
```

An *anonymous function expression* looks similar, but is an expression and works differently:

```
const id = function (x) {  
  return x;  
};
```

If you give the function expression a name, it now has the same syntax as a function declaration:

```
const id = function me(x) {  
  return x;  
};
```

Disambiguation will have to answer the following question: What happens if I use a named function expression as a statement?

### 7.5.2 Object literal vs. block

The following is an *object literal* defining an object:

```
{  
  foo: bar(3, 5)  
}
```

The created object has a single property (field), whose name is `foo`. Its value is the result of the function call `bar(3, 5)` (an expression).

But it is also a code block that contains a single line:

- The label `foo`:
- followed by the function call `bar(3, 5)` (an expression statement, in this case!).

### 7.5.3 Disambiguation

The ambiguities are only a problem in statement context: If the JavaScript parser encounters something ambiguous, it doesn't know if it's a plain statement or an expression statement. Therefore, expression statements must not start with:

- An open curly brace (`{`)
- The keyword `function`

If an expression starts with either of these tokens, you must put it in parentheses (which creates an expression context) if you want to use it as a statement. Let's continue with examples.

### 7.5.4 Example: Immediately Invoked Function Expression (IIFE)

JavaScript's `const` and `let` declarations for variables are block-scoped: the scope of a variable is its surrounding scope.

But there is also the legacy `var` declaration for variables, which is function-scoped: the scope of a variable is the whole surrounding function.

Ben Alman coined the term *IIFE* (pronounced "iffy"). It is a technique for simulating a block for `var`: a piece of code is wrapped in a function expression, which is called immediately after its creation.

```
(function () { // simulated block starts
  // x only exists inside the simulated block
  var x = 123;
})(); // simulated block ends
```

Why do we need to wrap the function expression in parentheses? What happens if we omit them?

In the following example, the IIFE in line A is incorrectly interpreted as a statement (and therefore a function declaration). That causes a syntax error, because normal function declarations must have names. They can't be immediately invoked, either.

```
let result;
assert.throws(
  () => eval("function () { result = 'success' }();"), // (A)
  {
    name: 'SyntaxError',
    message: 'Unexpected token (',
  });
```

As an aside: we need `eval()` here to delay the parsing of the code and therefore the syntax error. Otherwise, the whole example would be rejected by JavaScript, before running it.

We can fix the syntax error by putting the function expression in parentheses. Then JavaScript interprets it as an expression.

```
let result;
(function () { result = 'success' })();
assert.equal(result, 'success');
```

If an IIFE appears in expression context (not in statement context), then we don't need the parentheses:

```
const abc = function () { return 'abc' }();
assert.equal(abc, 'abc');
```

### 7.5.5 Example: immediate method call

The following code is similar to an IIFE: We create an object via an object literal and immediately call one of its methods.

```
let result;
assert.throws(
  () => eval("{ m() { result = 'yes' } }.m();"),
  {
    name: 'SyntaxError',
    message: 'Unexpected token {',
  });
```

The problem is that JavaScript thinks the initial open brace starts a code block (a statement) and not an object literal. Once again, we fix this via parentheses:

```
let result;
({ m() { result = 'yes' } }.m());
assert.equal(result, 'yes');
```

### 7.5.6 Example: destructuring via an object pattern

In the following example, we use object-destructuring to access property `.prop` of an object:

```
let p;
assert.throws(
  () => eval('{prop: p} = { prop: 123 };'),
  {
    name: 'SyntaxError',
    message: 'Unexpected token =',
  }
);
```

The problem is that JavaScript thinks the first open brace starts a code block. We fix it via parens:

```
let p;
({prop: p} = { prop: 123 });
assert.equal(p, 123);
```

### 7.5.7 Example: an arrow function that returns an object literal

You can use an arrow function with an expression body to return an object created via an object literal:

```
const func = () => ({ prop: 123 });
assert.deepEqual(func(), { prop: 123 });
```

If you don't use parentheses, JavaScript thinks the arrow function has a block body:

```
const func = () => { prop: 123 };
assert.deepEqual(func(), undefined);
```

## 7.6 Semicolons

### 7.6.1 Rule of thumb for semicolons

Each statement is terminated by a semicolon.

```
const x = 3;
someFunction('abc');
i++;
```

Except: statements ending with blocks.

```
function foo() {
  // ...
}
if (y > 0) {
  // ...
}
```

The following case is slightly tricky:



```
const func = function () {}; // semicolon!
```

The whole `const` declaration (a statement) ends with a semicolon, but inside it, there is a function expression. That is: It's not the statement per se that ends with a curly brace; it's the embedded function expression. That's why there is a semicolon at the end.

## 7.6.2 Semicolons: control statements

The body of a control statement is itself a statement. For example, this is the syntax of the `while` loop:

```
while (condition)
  statement
```

The body can be a single statement:

```
while (a > 0) a--;
```

But blocks are also statements and therefore legal bodies of control statements:

```
while (a > 0) {
  a--;
}
```

If you want a loop to have an empty body, your first option is an empty statement (which is just a semicolon):

```
while (processNextItem() > 0);
```

Your second option is an empty block:

```
while (processNextItem() > 0) {}
```

## 7.7 Automatic semicolon insertion (ASI)

While I recommend to always write semicolons, most of them are optional in JavaScript. The mechanism that makes this possible is called *automatic semicolon insertion* (ASI). In a way, it corrects syntax errors.

ASI works as follows. Parsing of a statement continues until there is either:

- A semicolon
- A line terminator followed by an illegal token

In other words, ASI can be seen as inserting semicolons at line breaks. The next subsections cover the pitfalls of ASI.

### 7.7.1 ASI triggered unexpectedly

The good news about ASI is that – if you don't rely on it and always write semicolons – there is only one pitfall that you need to be aware of. It is that JavaScript forbids line breaks after some tokens. If you do insert a line break, a semicolon will be inserted, too.

The token where this is most practically relevant is `return`. Consider, for example, the following code:

```
return
{
  first: 'jane'
};
```

This code is parsed as:

```
return;
{
  first: 'jane';
}
;
```

That is, an empty return statement, followed by a code block, followed by an empty statement.

Why does JavaScript do this? It protects against accidentally returning a value in a line after a return.

### 7.7.2 ASI unexpectedly not triggered

In some cases, ASI is *not* triggered when you think it should be. That makes life more complicated for people who don't like semicolons, because they need to be aware of those cases.

**Example 1:** Unintended function call.

```
a = b + c
(d + e).print()
```

Parsed as:

```
a = b + c(d + e).print();
```

**Example 2:** Unintended division.

```
a = b
/hi/g.exec(c).map(d)
```

Parsed as:

```
a = b / hi / g.exec(c).map(d);
```

**Example 3:** Unintended property access.

```
someFunction()
['ul', 'ol'].map(x => x + x)
```

Executed as:

```
const propKey = ('ul', 'ol');
assert.equal(propKey, 'ol'); // due to comma operator
```

```
someFunction()[propKey].map(x => x + x);
```

**Example 4:** Unintended function call.

```
const stringify = function (x) {
  return String(x)
```

```
}  
`abc`.split('')
```

Executed as:

```
const func = function (x) {  
  return String(x)  
};  
const _tmp = func`abc`;  
assert.equal(_tmp, 'abc');  
  
const stringify = _tmp.split('');  
assert.deepEqual(stringify, ['a', 'b', 'c']);
```

A function put in front of a template literal (such as ``abc``) leads to that function being called (the template literal determines what parameters it gets). More on that in the chapter on tagged templates.

## 7.8 Semicolons: best practices

I recommend that you always write semicolons:

- I like the visual structure it gives code – you clearly see when a statement ends.
- There are less rules to keep in mind.
- The majority of JavaScript programmers use semicolons.

However, there are also many people who don't like the added visual clutter of semicolons. If you are one of them: code without them is legal. I recommend that you use tools to help you avoid mistakes. The following are two examples:

- The automatic code formatter Prettier<sup>1</sup> can be configured to not use semicolons. It then automatically fixes problems. For example, if it encounters a line that starts with a square bracket, it prefixes that line with a semicolon.
- The static checker ESLint<sup>2</sup> has a rule<sup>3</sup> that warns about critical cases if you either require or forbid semicolons.

## 7.9 Strict mode

Starting with ECMAScript 5, you can optionally execute JavaScript in a so-called *strict mode*. In that mode, the language is slightly cleaner: a few quirks don't exist and more exceptions are thrown.

The default (non-strict) mode is also called *sloppy mode*.

Note that strict mode is switched on by default inside modules and classes, so you don't really need to know about it when you write modern JavaScript. In this book, I assume that strict mode is always switched on.

---

<sup>1</sup><https://prettier.io>

<sup>2</sup><https://eslint.org>

<sup>3</sup><https://eslint.org/docs/rules/semi>

### 7.9.1 Switching on strict mode

In legacy script files and CommonJS modules, you switch on strict mode for a complete file, by putting the following code in the first line:

```
'use strict';
```

The neat thing about this “directive” is that ECMAScript versions before 5 simply ignore it: it’s an expression statement that does nothing.

You can also switch on strict mode for just a single function:

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

### 7.9.2 Example: strict mode in action

Let’s look at an example where sloppy mode does something bad that strict mode doesn’t: Changing an unknown variable (that hasn’t been created via `let` or similar) creates a global variable.

```
function sloppyFunc() {
  unknownVar1 = 123;
}
sloppyFunc();
// Created global variable `unknownVar1`:
assert.equal(unknownVar1, 123);
```

Strict mode does it better:

```
function strictFunc() {
  'use strict';
  unknownVar2 = 123;
}
assert.throws(
  () => strictFunc(),
  {
    name: 'ReferenceError',
    message: 'unknownVar2 is not defined',
  });
```



Quiz: advanced

See quiz app.

## Chapter 8

# Assertion API

### Contents

<b>8.1 Assertions in software development</b>	<b>53</b>
<b>8.2 How assertions are used in this book</b>	<b>53</b>
8.2.1 Documenting results in code examples via assertions	54
8.2.2 Implementing test-driven exercises via assertions	54
<b>8.3 Normal comparison versus deep comparison</b>	<b>54</b>
<b>8.4 Quick reference: module <code>assert</code></b>	<b>55</b>
8.4.1 Normal equality	55
8.4.2 Deep equality	55
8.4.3 Expecting exceptions	55
8.4.4 Other tool functions	56

## 8.1 Assertions in software development

In software development, *assertions* make statements about values or pieces of code that must be true. If they aren't, an exception is thrown. Node.js supports assertions via its built-in module `assert`. For example:

```
import {strict as assert} from 'assert';
assert.equal(3 + 5, 8);
```

This assertion states that the expected result of 3 plus 5 is 8. The import statement uses the recommended strict version<sup>1</sup> of `assert`.

## 8.2 How assertions are used in this book

In this book, assertions are used in two ways: to document results in code examples and to implement test-driven exercises.

---

<sup>1</sup>[https://nodejs.org/api/assert.html#assert\\_strict\\_mode](https://nodejs.org/api/assert.html#assert_strict_mode)

### 8.2.1 Documenting results in code examples via assertions

In code examples, assertions express expected results. Take, for example, the following function:

```
function id(x) {
  return x;
}
```

`id()` returns its parameter. We can show it in action via an assertion:

```
assert.equal(id('abc'), 'abc');
```

In the examples, I usually omit the statement for importing `assert`.

The motivation behind using assertions is:

- You can specify precisely what is expected.
- Code examples can be tested automatically, which ensures that they really work.

### 8.2.2 Implementing test-driven exercises via assertions

The exercises for this book are test-driven, via the test framework `mocha`. Checks inside the tests are made via methods of `assert`.

The following is an example of such a test:

```
// For the exercise, you must implement the function hello().
// The test checks if you have done it properly.
test('First exercise', () => {
  assert.equal(hello('world'), 'Hello world!');
  assert.equal(hello('Jane'), 'Hello Jane!');
  assert.equal(hello('John'), 'Hello John!');
  assert.equal(hello(''), 'Hello !');
});
```

For more information, consult [the chapter on quizzes and exercises](#)

## 8.3 Normal comparison versus deep comparison

The strict `.equal()` uses `===` to compare values. That means that an object is only equal to itself – even if two objects have the same content:

```
assert.notEqual({foo: 1}, {foo: 1});
```

In such cases, you can use `.deepEqual()`:

```
assert.deepEqual({foo: 1}, {foo: 1});
```

This method works for Arrays, too:

```
assert.notEqual(['a', 'b', 'c'], ['a', 'b', 'c']);
assert.deepEqual(['a', 'b', 'c'], ['a', 'b', 'c']);
```

## 8.4 Quick reference: module assert

For the full documentation, see the Node.js docs<sup>2</sup>.

### 8.4.1 Normal equality

- function equal(actual: any, expected: any, message?: string): void  
`assert.equal(3+3, 6);`
- function notEqual(actual: any, expected: any, message?: string): void  
`assert.notEqual(3+3, 22);`

### 8.4.2 Deep equality

- function deepEqual(actual: any, expected: any, message?: string): void  
`assert.deepEqual([1,2,3], [1,2,3]);`  
`assert.deepEqual([], []);`  
  
*// To .equal(), an object is only equal to itself:*  
`assert.notEqual([], []);`
- function notDeepEqual(actual: any, expected: any, message?: string): void  
`assert.notDeepEqual([1,2,3], [1,2]);`

### 8.4.3 Expecting exceptions

If you want to (or expect to) receive an exception, you need `.throws`:

- function throws(block: Function, message?: string): void  
`assert.throws(  
 () => {  
 null.prop;  
 }  
);`
- function throws(block: Function, error: Function, message?: string): void  
`assert.throws(  
 () => {  
 null.prop;  
 },  
 TypeError  
);`
- function throws(block: Function, error: RegExp, message?: string): void

---

<sup>2</sup><https://nodejs.org/api/assert.html>

```

assert.throws(
  () => {
    null.prop;
  },
  /^TypeError: Cannot read property 'prop' of null$/
);

```

- function throws(block: Function, error: Object, message?: string): void

```

assert.throws(
  () => {
    null.prop;
  },
  {
    name: 'TypeError',
    message: `Cannot read property 'prop' of null`,
  }
);

```

#### 8.4.4 Other tool functions

- function fail(message: string | Error): never

```

try {
  functionThatShouldThrow();
  assert.fail();
} catch (_) {
  // Success
}

```



Quiz

See quiz app.



## Chapter 9

# Getting started with quizzes and exercises

### Contents

<b>9.1 Quizzes</b>	<b>57</b>
<b>9.2 Exercises</b>	<b>57</b>
9.2.1 Installing the exercises	57
9.2.2 Running exercises	58
<b>9.3 Unit tests in JavaScript</b>	<b>58</b>
9.3.1 A typical test	58
9.3.2 Asynchronous tests in mocha	59

At the end of most chapters, there are quizzes and exercises. These are a paid feature, but a comprehensive preview is available. This chapter explains how to get started with them.

## 9.1 Quizzes

Installation:

- Download and unzip `impatient-js-quiz.zip`

Running the quiz app:

- Open `impatient-js-quiz/index.html` in a web browser
- You'll see a TOC of all the quizzes.

## 9.2 Exercises

### 9.2.1 Installing the exercises

To install the exercises:

- Download and unzip `impatient-js-code.zip`
- Follow the instructions in `README.txt`

### 9.2.2 Running exercises

- Exercises are referred to by path in this book.
  - For example: `exercises/syntax/first_module_test.js`
- Within each file:
  - The first line contains the command for running the exercise.
  - The following lines describe what you have to do.

## 9.3 Unit tests in JavaScript

All exercises in this book are tests that are run via the test framework Mocha<sup>1</sup>. This section gives a brief introduction.

### 9.3.1 A typical test

Typical test code is split into two parts:

- Part 1: the code to be tested.
- Part 2: the tests for the code.

Take, for example, the following two files:

- `id.js` (code to be tested)
- `id_test.js` (tests)

#### 9.3.1.1 Part 1: the code

The code itself resides in `id.js`:

```
export function id(x) {  
  return x;  
}
```

The key thing here is: everything you want to test must be exported. Otherwise, the test code can't access it.

#### 9.3.1.2 Part 2: the tests

The tests for the code reside in `id_test.js`:

---

<sup>1</sup><https://mochajs.org>

```
import {strict as assert} from 'assert'; // (A)
import {id} from './id.js'; // (B)

test('My test', () => { // (C)
  assert.equal(id('abc'), 'abc'); // (D)
});
```

You don't need to worry too much about the syntax: You won't have to write this kind of code yourself – all tests are written for you.

The core of this test file resides in line D – a so-called *assertion*: `assert.equal()` specifies that the expected result of `id('abc')` is `'abc'`. The assertion library, a built-in Node.js module called `assert`, is documented in [the next chapter](#).

As for the other lines:

- Line A: We import the assertion library.
- Line B: We import the function to test.
- Line C: We define a test. This is done by calling the function `test()`:
  - First parameter: the name of the test.
  - Second parameter: the test code (provided via an arrow function with zero parameters).

To run the test, we execute the following in a command line:

```
npm t demos/syntax/id_test.js
```

The `t` is an abbreviation for `test`. That is, the long version of this command is:

```
npm test demos/syntax/id_test.js
```



#### Exercise: Your first exercise

The following exercise gives you a first taste of what exercises are like: `exercises/syntax/first_module_test.js`

### 9.3.2 Asynchronous tests in mocha



#### Reading

You can postpone reading this section until you get to the chapters on asynchronous programming.

Writing tests for asynchronous code requires extra work: The test receives its results later and has to signal to mocha that it isn't finished, yet, when it returns. The following subsections examine three ways of doing so.

#### 9.3.2.1 Calling `done()`

A test becomes asynchronous if it has at least one parameter. That parameter is usually called `done` and receives a function that you call once your code is finished:

```
test('addViaCallback', (done) => {
  addViaCallback(1, 2, (error, result) => {
    if (error) {
      done(error);
    } else {
      assert.strictEqual(result, 3);
      done();
    }
  });
});
```

### 9.3.2.2 Returning a Promise

A test also becomes asynchronous if it returns a Promise. Mocha considers the test to be finished once the Promise is either fulfilled or rejected. A test is considered successful if the Promise is fulfilled and failed if the Promise is rejected.

```
test('addAsync', () => {
  return addAsync(1, 2)
    .then(result => {
      assert.strictEqual(result, 3);
    });
});
```

### 9.3.2.3 The test is an async function

Async functions always return Promises. Therefore, an async function is a convenient way of implementing an asynchronous test. The following code is equivalent to the previous example.

```
test('addAsync', async () => {
  const result = await addAsync(1, 2);
  assert.strictEqual(result, 3);
  // No explicit return necessary!
});
```

You don't need to explicitly return anything: The implicitly returned undefined is used to fulfill the Promise returned by this async function. And if the test code throws an exception then the async function takes care of rejecting the returned Promise.

## **Part III**

# **Variables and values**



# Chapter 10

## Variables and assignment

### Contents

<b>10.1 let</b>	<b>63</b>
<b>10.2 const</b>	<b>64</b>
10.2.1 const and immutability	64
10.2.2 const and loops	64
<b>10.3 Deciding between let and const</b>	<b>65</b>
<b>10.4 Variables are block-scoped</b>	<b>65</b>
10.4.1 Shadowing and blocks	65

These are JavaScript's main ways of declaring variables:

- `let` declares mutable variables.
- `const` declares *constants* (immutable variables).

Before ES6, there was also `var`. But it has several quirks, so it's best to avoid it in modern JavaScript. You can read more about it in "Speaking JavaScript"<sup>1</sup>.

### 10.1 let

Variables declared via `let` are mutable:

```
let i;  
i = 0;  
i = i + 1;  
assert.equal(i, 1);
```

You can also declare and assign at the same time:

```
let i = 0;
```

---

<sup>1</sup><http://speakingjs.com/es5/ch16.html>

## 10.2 const

Variables declared via `const` are immutable. You must always initialize immediately:

```
const i = 0; // must initialize

assert.throws(
  () => { i = i + 1 },
  {
    name: 'TypeError',
    message: 'Assignment to constant variable.',
  }
);
```

### 10.2.1 const and immutability

In JavaScript, `const` only means that the *binding* (the association between variable name and variable value) is immutable. The value itself may be mutable, like `obj` in the following example.

```
const obj = { prop: 0 };

obj.prop = obj.prop + 1;
assert.equal(obj.prop, 1);
```

However:

```
const obj = { prop: 0 };

assert.throws(
  () => { obj = {} },
  {
    name: 'TypeError',
    message: 'Assignment to constant variable.',
  }
);
```

### 10.2.2 const and loops

You can use `const` with `for-of` loops, where a fresh binding is created for each iteration:

```
const arr = ['hello', 'world'];
for (const elem of arr) {
  console.log(elem);
}
// Output:
// 'hello'
// 'world'
```

In plain `for` loops, you must use `let`, however:



```
const arr = ['hello', 'world'];
for (let i=0; i<arr.length; i++) {
  const elem = arr[i];
  console.log(elem);
}
```

## 10.3 Deciding between let and const

I recommend the following rules to decide between let and const:

- const indicates an immutable binding and that a variable never changes its value. Prefer it.
- let indicates that the value of a variable changes. Use it only when you can't use const.



### Exercise: const

exercises/variables-assignment/const\_exrc.js

## 10.4 Variables are block-scoped

The *scope* of a variable is the region of a program where it can be accessed.

Like in most modern programming languages, variables declared via let and const are *block-scoped*: they can only be accessed from within the block that they are declared in.

```
{
  const x = 0;
}
assert.throws(
  () => x + 1,
  {
    name: 'ReferenceError',
    message: 'x is not defined',
  }
);
```

The curly braces enclose a code block. x only exists within that block and can't be accessed outside it.

### 10.4.1 Shadowing and blocks

You can't declare the same variable twice at the same level. You can, however, nest a block and use the same variable name x that you used outside the block:

```
const x = 1;
assert.equal(x, 1);
{
  const x = 2;
  assert.equal(x, 2);
}
```

```
}  
assert.equal(x, 1);
```

Inside the block, the inner `x` is the only accessible variable with that name. The inner `x` is said to *shadow* the outer `x`. Once you leave the block, you can access the old value again.



#### More information on variable scopes and closures

For more on variable scopes and closures, consult [the corresponding chapter](#) later in this book.



#### Quiz

[See quiz app.](#)

# Chapter 11

## Values

### Contents

---

<b>11.1 What's a type?</b> . . . . .	<b>67</b>
<b>11.2 JavaScript's type hierarchy</b> . . . . .	<b>67</b>
<b>11.3 The types of the language specification</b> . . . . .	<b>68</b>
<b>11.4 Primitive values versus objects</b> . . . . .	<b>69</b>
11.4.1 Primitive values . . . . .	69
11.4.2 Objects . . . . .	69
<b>11.5 Classes and constructor functions</b> . . . . .	<b>71</b>
<b>11.6 Constructor functions associated with primitive types</b> . . . . .	<b>71</b>
<b>11.7 The operators <code>typeof</code> and <code>instanceof</code>: what's the type of a value?</b> . . . . .	<b>71</b>
11.7.1 <code>typeof</code> . . . . .	72
11.7.2 <code>instanceof</code> . . . . .	72
<b>11.8 Converting between types</b> . . . . .	<b>73</b>
11.8.1 Explicit conversion between types . . . . .	73
11.8.2 Coercion (automatic conversion between types) . . . . .	73

---

In this chapter, we'll examine what kinds of values JavaScript has.

We'll occasionally use the strict equality operator (`===`), which is explained in [the chapter on operators](#).

### 11.1 What's a type?

For this chapter, I consider types to be sets of values. For example, the type `boolean` is the set { `false`, `true` }.

### 11.2 JavaScript's type hierarchy

Fig. 11.1 shows JavaScript's type hierarchy. What do we learn from that diagram?

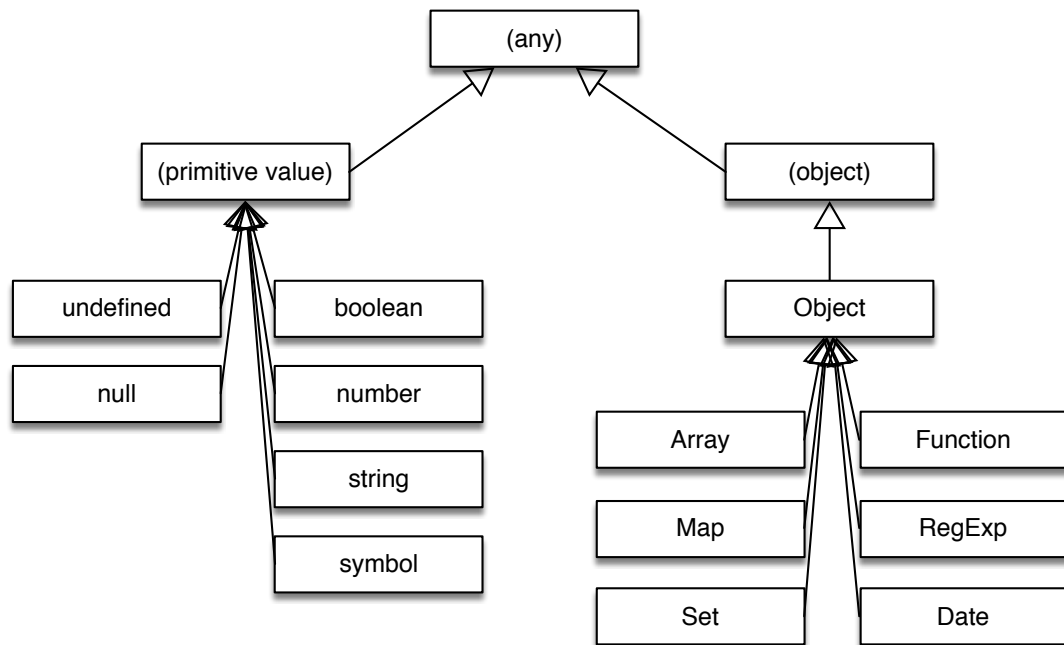


Figure 11.1: A partial hierarchy of JavaScript’s types. Missing are the classes for errors, the constructor functions associated with primitive types, and more. The diagram hints at the fact that not all objects are instances of `Object`.

- JavaScript distinguishes two kinds of values: primitive values and objects. We’ll see soon what the difference is.
- Some objects are not instances of class `Object`. In [the chapter on prototype chains and classes](#), you’ll learn how to create those special objects. However, you’ll rarely encounter them in practice.

### 11.3 The types of the language specification

The ECMAScript specification only knows a total of 7 types. The names of those types are (I’m using TypeScript’s names, not the spec’s names):

- `undefined`: with the only element `undefined`.
- `null`: with the only element `null`.
- `boolean`: with the elements `false` and `true`.
- `number`: the type of all numbers (e.g. `-123`, `3.141`).
- `string`: the type of all strings (e.g. `'abc'`).
- `symbol`: the type of all symbols (e.g. `Symbol('My Symbol')`).
- `object`: the type of all objects (different from `Object`, the type of all instances of class `Object` and its subclasses).

## 11.4 Primitive values versus objects

The specification makes an important distinction between values:

- *Primitive values* are the elements of the types `undefined`, `null`, `boolean`, `number`, `string`, `symbol`.
- All other values are *objects*.

In contrast to Java (that inspired JavaScript here), primitives are not second-class citizens. The difference between them and objects is more subtle. In a nutshell, it is:

- **Primitive values:** are atomic building blocks of data in JavaScript. They are passed and compared *by value*: We pass and compare their contents (details soon).
- **Objects:** are compound pieces of data. They are passed and compared *by reference*: We pass and compare (transparent) references to the actual objects on the heap (details soon).

Next, we'll look at primitive values and objects in more depth.

### 11.4.1 Primitive values

#### 11.4.1.1 Primitives are immutable

You can't change, add or remove the properties (fields) of primitives:

```
let num = 123;
assert.throws(
  () => { num.foo = 'abc' },
  {
    name: 'TypeError',
    message: "Cannot create property 'foo' on number '123'",
  }
);
```

#### 11.4.1.2 Primitives are passed and compared *by value*

Primitives are passed and compared *by value*. That means: Variables (incl. parameters) store the primitives themselves and when comparing primitives, we are comparing their contents.

The following code demonstrates comparing by value:

```
assert.ok(123 === 123);
assert.ok('abc' === 'abc');
```

To see what's so special about this way of comparing, read on and find out how objects are compared.

### 11.4.2 Objects

Two common literals for creating objects, are:

- Object literals:

```
const obj = {
  first: 'Jane',
  last: 'Doe',
};
```

- Array literals:

```
const arr = ['foo', 'bar'];
```

#### 11.4.2.1 Objects are mutable by default

By default, you can freely change, add and remove the properties of objects:

```
const obj = {};
```

```
obj.foo = 'abc'; // add a property
assert.equal(obj.foo, 'abc');
```

```
obj.foo = 'def'; // change a property
assert.equal(obj.foo, 'def');
```

#### 11.4.2.2 Objects are passed and compared *by reference*

Objects are passed and compared *by reference*. That means: Variables (incl. parameters) hold (transparent) references to objects on the *heap* (think shared main memory). When comparing objects, we are comparing references. Each object literal creates a fresh object on the heap and returns a reference to it.

The following code demonstrates comparing by reference:

```
const obj = {}; // fresh empty object
assert.ok(obj === obj);
assert.ok({} !== {}); // two fresh, different objects
```

The following code demonstrates passing by reference:

```
const a = {};
// Pass the reference in `a` to `b`:
const b = a;
```

```
// Now `a` and `b` point to the same object
// (they “share” that object):
assert.ok(a === b);
```

```
// Changing `a` also changes `b`:
a.foo = 123;
assert.equal(b.foo, 123);
```

JavaScript uses *garbage collection* to automatically manage memory:

```
let obj = { prop: 'value' };
obj = {};
```

Now the old value { prop: 'value' } of obj is *garbage* (not used anymore). JavaScript will automatically *garbage-collect* it (remove it from memory), at some point in time (possibly never if there is enough free memory).

## 11.5 Classes and constructor functions

JavaScript’s original factories for objects are *constructor functions*: ordinary functions that return “instances” of themselves if you invoke them via the new operator.

ES6 introduced *classes*, which are mainly better syntax for constructor functions.

In this book, I’m using the terms *constructor function* and *class* interchangeably.

Classes can be seen as partitioning the single type object of the specification into subtypes – they give us more types than the limited 7 ones of the specification. Each class is the type of the objects that were created by it.

## 11.6 Constructor functions associated with primitive types

Each primitive type (except for the spec-internal types for undefined and null) has an associated *constructor function* (think class):

- The constructor function `Boolean` is associated with booleans.
- The constructor function `Number` is associated with numbers.
- The constructor function `String` is associated with strings.
- The constructor function `Symbol` is associated with symbols.

Each of these functions plays several roles. For example, `Number`:

- You can use it as a function and convert values to numbers:

```
assert.equal(Number('123'), 123);
```

- The properties stored in `Number.prototype` are “inherited” by numbers:

```
assert.equal((123).toString(), Number.prototype.toString());
```

- It also contains tool functions for numbers. For example:

```
assert.equal(Number.isInteger(123), true);
```

- Lastly, you can also use `Number` as a class and create number objects. These objects are different from real numbers and should be avoided.

```
assert.notStrictEqual(new Number(123), 123);
assert.equal(new Number(123).valueOf(), 123);
```

## 11.7 The operators `typeof` and `instanceof`: what’s the type of a value?

The two operators `typeof` and `instanceof` let you determine what type a given value `x` has:

```
if (typeof x === 'string') ...
if (x instanceof Array) ...
```

So how do they differ?

- `typeof` distinguishes the 7 types of the specification (minus one omission, plus one addition).
- `instanceof` tests which class created a given value.

Thus, as a rough rule of thumb: `typeof` is for primitive values, `instanceof` is for objects.

### 11.7.1 `typeof`

Table 11.1: The results of the `typeof` operator.

x	typeof x
undefined	'undefined'
null	'object'
Boolean	'boolean'
Number	'number'
String	'string'
Symbol	'symbol'
Function	'function'
All other objects	'object'

Tbl. 11.1 lists all results of `typeof`. They roughly correspond to the 7 types of the language specification. Alas, there are two differences and they are language quirks:

- `typeof null` returns `'object'` and not `'null'`. That's a bug. Unfortunately, it can't be fixed. TC39 tried to do that, but it broke too much code on the web.
- `typeof` of a function should be `'object'` (functions are objects). Introducing a separate category for functions is confusing.



#### Exercises: Two exercises on `typeof`

- `exercises/operators/typeof_exrc.js`
- Bonus: `exercises/operators/is_object_test.js`

### 11.7.2 `instanceof`

This operator answers the question: has a value `x` been created by a class `C`?

```
x instanceof C
```

For example:

```
> (function() {}) instanceof Function
true
> ({} instanceof Object
```



```
true
> [] instanceof Array
true
```

Primitive values are not instances of anything:

```
> 123 instanceof Number
false
> '' instanceof String
false
> '' instanceof Object
false
```



#### Exercise: instanceof

`exercises/operators/instanceof_exrc.js`

## 11.8 Converting between types

There are two ways in which values are converted to other types in JavaScript:

- Explicit conversion: via functions such as `String()`.
- Coercion (automatic conversion): happens when an operation receives operands/parameters that it can't work with.

### 11.8.1 Explicit conversion between types

The function associated with a primitive type explicitly converts values to that type:

```
> Boolean(0)
false
> Number('123')
123
> String(123)
'123'
```

You can also use `Object()` to convert values to objects:

```
> 123 instanceof Number
false
> Object(123) instanceof Number
true
```

### 11.8.2 Coercion (automatic conversion between types)

For many operations, JavaScript automatically converts the operands/parameters if their types don't fit. This kind of automatic conversion is called *coercion*.

For example, the multiplication operator coerces its operands to numbers:

```
> '7' * '3'  
21
```

Many built-in functions coerce, too. For example, `parseInt()` coerces its parameter to string (parsing stops at the first character that is not a digit):

```
> parseInt(123.45)  
123
```



#### Exercise: Converting values to primitives

[exercises/values/conversion\\_exrc.js](#)



#### Quiz

[See quiz app.](#)

# Chapter 12

## Operators

### Contents

---

<b>12.1 Two important rules for operators</b>	<b>75</b>
12.1.1 Operators coerce their operands to appropriate types	75
12.1.2 Most operators only work with primitive values	76
<b>12.2 The plus operator (+)</b>	<b>76</b>
<b>12.3 Assignment operators</b>	<b>77</b>
12.3.1 The plain assignment operator	77
12.3.2 Compound assignment operators	77
12.3.3 All compound assignment operators	77
<b>12.4 Equality: == versus ===</b>	<b>78</b>
12.4.1 Lenient equality (== and !=)	78
12.4.2 Strict equality (=== and !==)	78
12.4.3 Recommendation: always use strict equality	79
<b>12.5 Ordering operators</b>	<b>80</b>
<b>12.6 Various other operators</b>	<b>80</b>

---

### 12.1 Two important rules for operators

- Operators coerce their operands to appropriate types
- Most operators only work with primitive values

#### 12.1.1 Operators coerce their operands to appropriate types

If an operator gets operands that don't have the proper types, it rarely throws an exception. Instead, it *coerces* (automatically converts) the operands so that it can work with them. Let's look at two examples.

First, the multiplication operator can only work with numbers. Therefore, it converts strings to numbers before computing its result.

```
> '7' * '3'
21
```

Second, the square brackets operator ([ ]) for accessing the properties of an object can only handle strings and symbols. All other values are coerced to string:

```
const obj = {};
obj['true'] = 123;

// Coerce true to the string 'true'
assert.equal(obj[true], 123);
```

### 12.1.2 Most operators only work with primitive values

The main rule to keep in mind for JavaScript's operators is:

Most operators only work with primitive values.

If an operand is an object, it is usually coerced to a primitive value.

For example:

```
> [1,2,3] + [4,5,6]
'1,2,34,5,6'
```

Why? The plus operator first coerces its operands to primitive values:

```
> String([1,2,3])
'1,2,3'
> String([4,5,6])
'4,5,6'
```

Next, it concatenates the two strings:

```
> '1,2,3' + '4,5,6'
'1,2,34,5,6'
```

## 12.2 The plus operator (+)

The plus operator works as follows in JavaScript: It first converts both operands to primitive values. Then it switches to one of two modes:

- String mode: If one of the two results is a string then convert the other result to string, too, and concatenate both strings.
- Number mode: Otherwise, convert both operands to numbers and add them.

String mode lets you use + to assemble strings:

```
> 'There are ' + 3 + ' items'
'There are 3 items'
```

Number mode means that if neither operand is a string (or an object that becomes a string) then everything is coerced to numbers:

```
> 4 + true
5
(Number(true) is 1.)
```

## 12.3 Assignment operators

### 12.3.1 The plain assignment operator

- `x = value`  
Assign to a previously declared variable.
- `const x = value`  
Declare and assign at the same time.
- `obj.propKey = value`  
Assign to a property.
- `arr[index] = value`  
Assign to an Array element.

### 12.3.2 Compound assignment operators

Given an operator `op`, the following two ways of assigning are equivalent:

```
myvar op= value
myvar = myvar op value
```

If, for example, `op` is `+` then we get the operator `+=` that works as follows.

```
let str = '';
str += '<b>';
str += 'Hello!';
str += '</b>';

assert.equal(str, '<b>Hello!</b>');
```

### 12.3.3 All compound assignment operators

- Arithmetic operators:  
`+= -= *= /= %= **=`  
`+=` also works for string concatenation
- Bitwise operators:  
`<<= >>= >>>= &= ^= |=`

## 12.4 Equality: == versus ===

JavaScript has two kinds of equality operators: lenient equality (==) and strict equality (===). The recommendation is to always use the latter.

### 12.4.1 Lenient equality (== and !=)

Lenient equality is one of JavaScript's quirks. It often coerces operands. Some of those coercions make sense:

```
> '123' == 123
true
> false == 0
true
```

Others less so:

```
> '' == 0
true
```

Objects are coerced to primitives if (and only if!) the other operand is primitive:

```
> [1, 2, 3] == '1,2,3'
true
> ['1', '2', '3'] == '1,2,3'
true
```

If both operands are objects, they are only equal if they are the same object:

```
> [1, 2, 3] == ['1', '2', '3']
false
> [1, 2, 3] == [1, 2, 3]
false
```

```
> const arr = [1, 2, 3];
> arr == arr
true
```

Lastly, == considers undefined and null to be equal:

```
> undefined == null
true
```

### 12.4.2 Strict equality (=== and !==)

Strict equality never coerces. Two values are only equal if they have the same type. Let's revisit our previous interaction with the == operator and see what the === operator does:

```
> false === 0
false
> '123' === 123
false
```

An object is only equal to another value if that value is the same object:

```
> [1, 2, 3] === '1,2,3'
false
> ['1', '2', '3'] === '1,2,3'
false

> [1, 2, 3] === ['1', '2', '3']
false
> [1, 2, 3] === [1, 2, 3]
false

> const arr = [1, 2, 3];
> arr === arr
true
```

The === operator does not consider undefined and null to be equal:

```
> undefined === null
false
```

### 12.4.3 Recommendation: always use strict equality

I recommend to always use ===. It makes your code easier to understand and spares you from having to think about the quirks of ==.

Let's look at two use cases for == and what I recommend to do instead.

#### 12.4.3.1 Use case for ==: comparing with a number or a string

== lets you check if a value x is a number or that number as a string – with a single comparison:

```
if (x == 123) {
  // x is either 123 or '123'
}
```

I prefer either of the following two alternatives:

```
if (x === 123 || x === '123') ...
if (Number(x) === 123) ...
```

You can also convert x to a number when you first encounter it.

#### 12.4.3.2 Use case for ==: comparing with undefined or null

Another use case for == is to check if a value x is either undefined or null:

```
if (x == null) {
  // x is either null or undefined
}
```

The problem with this code is that you can't be sure if someone meant to write it that way or if they made a typo and meant `=== null`.

I prefer either of the following two alternatives:

```
if (x === undefined || x === null) ...
if (x) ...
```

The second alternative is even more sloppy than using `==`, but it is a well-established pattern in JavaScript (to be explained in detail in [the chapter on booleans](#), when we look at truthiness and falsiness).

## 12.5 Ordering operators

Table 12.1: JavaScript's ordering operators.

Operator	name
<	less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal

JavaScript's ordering operators (tbl. 12.1) work for both numbers and strings:

```
> 5 >= 2
true
> 'bar' < 'foo'
true
```

Caveat: These operators don't work well for comparing text in a human language (capitalization, accents, etc.). The details are explained in [the chapter on strings](#).

## 12.6 Various other operators

- Comma operator<sup>1</sup>: `a, b`
- void operator<sup>2</sup>: `void 0`
- Operators for booleans, strings, numbers, objects: are covered elsewhere in this book.



Quiz

[See quiz app.](#)

<sup>1</sup>[http://speakingjs.com/es5/ch09.html#comma\\_operator](http://speakingjs.com/es5/ch09.html#comma_operator)

<sup>2</sup>[http://speakingjs.com/es5/ch09.html#void\\_operator](http://speakingjs.com/es5/ch09.html#void_operator)



## **Part IV**

# **Primitive values**



## Chapter 13

# The non-values `undefined` and `null`

### Contents

---

<b>13.1 <code>undefined</code> vs. <code>null</code></b> . . . . .	<b>83</b>
13.1.1 The history of <code>undefined</code> and <code>null</code> . . . . .	83
<b>13.2 Occurrences of <code>undefined</code> and <code>null</code></b> . . . . .	<b>84</b>
13.2.1 Occurrences of <code>undefined</code> . . . . .	84
13.2.2 Occurrences of <code>null</code> . . . . .	84
<b>13.3 Checking for <code>undefined</code> or <code>null</code></b> . . . . .	<b>85</b>
<b>13.4 <code>undefined</code> and <code>null</code> don't have properties</b> . . . . .	<b>85</b>

---

### 13.1 `undefined` vs. `null`

Many programming languages, especially object-oriented ones, have the non-value `null` indicating that a variable does not currently point to an object. For example, when it hasn't been initialized, yet.

In addition to `null`, JavaScript also has the similar non-value `undefined`. So what is the difference between them? This is a rough description of how they work:

- `undefined` means: something does not exist or is uninitialized. This value is often produced by the language. It exists at a more fundamental level than `null`.
- `null` means: something is switched off. This value is often produced by code.

In reality, both non-values are often used interchangeably and many approaches for detecting `undefined` also detect `null`.

We'll soon see examples of where the two are used, which should give you a clearer idea of their natures.

#### 13.1.1 The history of `undefined` and `null`

When it came to picking one or more non-values for JavaScript, inspiration was taken from Java where initialization values depend on the static type of a variable:

- Variables with object types are initialized with `null`.
- Each primitive type has its own initialization value. For example, `int` variables are initialized with `0`.

Therefore, the original idea was:

- `null` means: not an object.
- `undefined` means: neither a primitive value nor an object.

## 13.2 Occurrences of `undefined` and `null`

The following subsections describe where `undefined` and `null` appear in the language. We'll encounter several mechanisms that are explained in more detail later in this book.

### 13.2.1 Occurrences of `undefined`

Uninitialized variable `myVar`:

```
let myVar;
assert.equal(myVar, undefined);
```

Parameter `x` is not provided:

```
function func(x) {
  return x;
}
assert.equal(func(), undefined);
```

Property `.unknownProp` is missing:

```
const obj = {};
assert.equal(obj.unknownProp, undefined);
```

If you don't explicitly specify the result of a function via the `return` operator, JavaScript returns `undefined` for you:

```
function func() {}
assert.equal(func(), undefined);
```

### 13.2.2 Occurrences of `null`

Last member of a prototype chain:

```
> Object.getPrototypeOf(Object.prototype)
null
```

Result if a regular expression `/a/` does not match a string `'x'`:

```
> /a/.exec('x')
null
```

The JSON data format does not support `undefined`, only `null`:

```
> JSON.stringify({a: undefined, b: null})
'{"b":null}'
```

## 13.3 Checking for undefined or null

Checking for either:

```
if (x === null) ...
if (x === undefined) ...
```

Does x have a value?

```
if (x !== undefined && x !== null) {
  // ...
}
if (x) { // truthy?
  // x is neither: undefined, null, false, 0, NaN, ''
}
```

Is x either undefined or null?

```
if (x === undefined || x === null) {
  // ...
}
if (!x) { // falsy?
  // x is: undefined, null, false, 0, NaN, ''
}
```

*Truthy* means “is true if coerced to boolean”. *Falsy* means “is false if coerced to boolean”. Both concepts are explained properly in [the chapter on booleans](#)).

## 13.4 undefined and null don't have properties

undefined and null are the two only JavaScript values where you get an exception if you try to read a property. To explore this phenomenon, let's use the following function, which reads (“gets”) property `.foo` and returns the result.

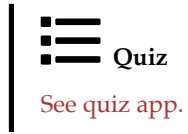
```
function getFoo(x) {
  return x.foo;
}
```

If we apply `getFoo()` to various value, we can see that it only fails for undefined and null:

```
> getFoo(undefined)
TypeError: Cannot read property 'foo' of undefined
> getFoo(null)
TypeError: Cannot read property 'foo' of null

> getFoo(true)
undefined
```

```
> getFoo({})  
undefined
```



See quiz app.

# Chapter 14

## Booleans

### Contents

<b>14.1 Converting to boolean</b> . . . . .	<b>87</b>
14.1.1 Ways of converting to boolean . . . . .	88
<b>14.2 Falsy and truthy values</b> . . . . .	<b>88</b>
14.2.1 Pitfall: truthiness checks are imprecise . . . . .	89
14.2.2 Checking for truthiness or falsiness . . . . .	89
14.2.3 Use case: was a parameter provided? . . . . .	90
14.2.4 Use case: does a property exist? . . . . .	90
<b>14.3 Conditional operator (?:)</b> . . . . .	<b>91</b>
<b>14.4 Binary logical operators: And (&amp;&amp;), Or (  )</b> . . . . .	<b>91</b>
14.4.1 Logical And (&&) . . . . .	92
14.4.2 Logical Or (  ) . . . . .	92
14.4.3 Default values via logical Or (  ) . . . . .	93
<b>14.5 Logical Not (!)</b> . . . . .	<b>93</b>

The primitive type *boolean* comprises two values:

```
> typeof false
'boolean'
> typeof true
'boolean'
```

### 14.1 Converting to boolean

Table 14.1: Converting values to booleans.

x	Boolean(x)
undefined	false
null	false

x	Boolean(x)
boolean value	x (no change)
number value	0 → false, NaN → false other numbers → true
string value	' ' → false other strings → true
object value	always true

Tbl. 14.1 describes how various values are converted to boolean.

### 14.1.1 Ways of converting to boolean

These are three ways in which you can convert an arbitrary value *x* to a boolean.

- `Boolean(x)`  
Most descriptive; recommended.
- `x ? true : false`  
Uses the conditional operator (explained [later in this chapter](#)).
- `!!x`  
Uses the [logical Not operator \(!\)](#). This operator coerces its operand to boolean. It is applied a second time to get a non-negated result.

## 14.2 Falsy and truthy values

In JavaScript, if you try to read something that doesn't exist, you often get `undefined` as a result. In these cases, an existence check amounts to comparing an expression with `undefined`. For example, the following code checks if object `obj` has the property `.prop`:

```
if (obj.prop !== undefined) {  
  // obj has property .prop  
}
```

To simplify this check, we can use the fact that the `if` statement always converts its conditional value to boolean:

```
if ('abc') { // true, if converted to boolean  
  console.log('Yes!');  
}
```

Therefore, we can use the following code to check if `obj.prop` exists. That is less precise than comparing with `undefined`, but also more succinct:

```
if (obj.prop) {  
  // obj has property .prop  
}
```

This simplified check is so popular that the following two names were introduced:



- A value is called *truthy* if it is `true` when converted to boolean.
- A value is called *falsy* if it is `false` when converted to boolean.

Consulting tbl. 14.1, we can make an exhaustive list of falsy values:

- undefined, null
- Booleans: `false`
- Numbers: `0`, `NaN`
- Strings: `''`

All other values (incl. *all* objects) are truthy:

```
> Boolean('abc')
true
> Boolean([])
true
> Boolean({})
true
```

### 14.2.1 Pitfall: truthiness checks are imprecise

Truthiness checks have one pitfall: they are not very precise. Consider this previous example:

```
if (obj.prop) {
  // obj has property .prop
}
```

The body of the `if` statement is skipped if:

- `obj.prop` is missing (in which case, JavaScript returns `undefined`).

However, it is also skipped if:

- `obj.prop` is `undefined`.
- `obj.prop` is any other falsy value (`null`, `0`, `''`, etc.).

In practice, this rarely causes problems, but you have to be aware of this pitfall.

### 14.2.2 Checking for truthiness or falsiness

```
if (x) {
  // x is truthy
}

if (!x) {
  // x is falsy
}

if (x) {
  // x is truthy
} else {
  // x is falsy
}
```

```
}
```

```
const result = x ? 'truthy' : 'falsy';
```

The conditional operator that is used in the last line, is explained [later in this chapter](#).

### 14.2.3 Use case: was a parameter provided?

A truthiness check is often used to determine if the caller of a function provided a parameter:

```
function func(x) {
  if (!x) {
    throw new Error('Missing parameter x');
  }
  // ...
}
```

On the plus side, this pattern is established and concise. It correctly throws errors for undefined and null.

On the minus side, there is the previously mentioned pitfall: the code also throws errors for all other falsy values.

An alternative is to check for undefined:

```
if (x === undefined) {
  throw new Error('Missing parameter x');
}
```

### 14.2.4 Use case: does a property exist?

Truthiness checks are also often used to determine if a property exists:

```
function readFile(fileDesc) {
  if (!fileDesc.path) {
    throw new Error('Missing property: .path');
  }
  // ...
}
readFile({ path: 'foo.txt' }); // no error
```

This pattern is also established and has the usual caveat: it not only throws if the property is missing, but also if it exists and has any of the falsy values.

If you truly want to check if the property exists, you have to use [the in operator](#):

```
if (!('path' in fileDesc)) {
  throw new Error('Missing property: .path');
}
```



Exercise: Truthiness

```
exercises/booleans/truthiness_exrc.js
```

## 14.3 Conditional operator (? :)

The conditional operator is the expression version of the `if` statement. Its syntax is:

```
«condition» ? «thenExpression» : «elseExpression»
```

It is evaluated as follows:

- If condition is truthy, evaluate and return `thenExpression`.
- Otherwise, evaluate and return `elseExpression`.

The conditional operator is also called *ternary operator*, because it has three operands.

Examples:

```
> true ? 'yes' : 'no'
'yes'
> false ? 'yes' : 'no'
'no'
> '' ? 'yes' : 'no'
'no'
```

The following code demonstrates that, whichever of the two branches “then” and “else” is chosen via the condition – only that branch is evaluated. The other branch isn’t.

```
const x = (true ? console.log('then') : console.log('else'));

// Output:
// 'then'
```

## 14.4 Binary logical operators: And (&&), Or (||)

The operators `&&` and `||` are *value-preserving* and *short-circuiting*. What does that mean?

*Value-preservation* means that operands are interpreted as booleans, but returned unchanged:

```
> 12 || 'hello'
12
> 0 || 'hello'
'hello'
```

*Short-circuiting* means: If the first operand already determines the result, the second operand is not evaluated. The only other operator that delays evaluating its operands is the conditional operator: Usually, all operands are evaluated before performing an operation.

For example, logical And (`&&`) does not evaluate its second operand if the first one is falsy:

```
const x = false && console.log('hello');
// No output
```

If the first operand is truthy, the `console.log()` is executed:

```
const x = true && console.log('hello');

// Output:
// 'hello'
```

### 14.4.1 Logical And (&&)

The expression `a && b` (“a And b”) is evaluated as follows:

- Evaluate `a`.
- Is the result falsy? Return it.
- Otherwise, evaluate `b` and return the result.

In other words, the following two expressions are roughly equivalent:

```
a && b
!a ? a : b
```

Examples:

```
> false && true
false
> false && 'abc'
false

> true && false
false
> true && 'abc'
'abc'

> '' && 'abc'
''
```

### 14.4.2 Logical Or (||)

The expression `a || b` (“a Or b”) is evaluated as follows:

- Evaluate `a`.
- Is the result truthy? Return it.
- Otherwise, evaluate `b` and return the result.

In other words, the following two expressions are roughly equivalent:

```
a || b
a ? a : b
```

Examples:

```
> true || false
true
> true || 'abc'
true
```

```
> false || true
true
> false || 'abc'
'abc'

> 'abc' || 'def'
'abc'
```

### 14.4.3 Default values via logical Or (||)

Sometimes you receive a value and only want to use it if it isn't either `null` or `undefined`. Otherwise, you'd like to use a default value, as a fallback. You can do that via the `||` operator:

```
const valueToUse = valueReceived || defaultValue;
```

The following code shows a real-world example:

```
function countMatches(regex, str) {
  const matchResult = str.match(regex); // null or Array
  return (matchResult || []).length;
}
```

If there are one or more matches for `regex` inside `str` then `.match()` returns an `Array`. If there are no matches, it unfortunately returns `null` (and not the empty `Array`). We fix that via the `||` operator.



#### Exercise: Default values via the Or operator (||)

`exercises/booleans/default_via_or_exrc.js`

## 14.5 Logical Not (!)

The expression `!x` ("Not `x`") is evaluated as follows:

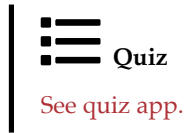
- Evaluate `x`.
- Is it `truthy`? Return `false`.
- Otherwise, return `true`.

Examples:

```
> !false
true
> !true
false

> !0
true
> !123
false
```

```
> !''  
true  
> !'abc'  
false
```



See quiz app.

# Chapter 15

## Numbers

### Contents

---

<b>15.1 JavaScript only has floating point numbers</b>	<b>96</b>
<b>15.2 Number literals</b>	<b>96</b>
15.2.1 Integer literals	96
15.2.2 Floating point literals	97
15.2.3 Syntactic pitfall: properties of integer literals	97
<b>15.3 Number operators</b>	<b>97</b>
15.3.1 Binary arithmetic operators	97
15.3.2 Unary plus and negation	98
15.3.3 Incrementing (++) and decrementing (--)	98
<b>15.4 Converting to number</b>	<b>99</b>
<b>15.5 Error values</b>	<b>99</b>
<b>15.6 Error value: NaN</b>	<b>100</b>
15.6.1 Checking for NaN	100
15.6.2 Finding NaN in Arrays	100
<b>15.7 Error value: Infinity</b>	<b>101</b>
15.7.1 Infinity as a default value	101
15.7.2 Checking for Infinity	101
<b>15.8 The precision of numbers: careful with decimal fractions</b>	<b>102</b>
<b>15.9 (Advanced)</b>	<b>102</b>
<b>15.10 Background: floating point precision</b>	<b>102</b>
<b>15.11 Integers in JavaScript</b>	<b>103</b>
15.11.1 Converting to integer	103
15.11.2 Ranges of integers in JavaScript	104
15.11.3 Safe integers	104
<b>15.12 Bitwise operators</b>	<b>105</b>
15.12.1 Binary bitwise operators	105
15.12.2 Bitwise Not	106
15.12.3 Bitwise shift operators	106
<b>15.13 Quick reference: numbers</b>	<b>107</b>
15.13.1 Converting to number	107

15.13.2 Arithmetic operators . . . . .	107
15.13.3 Bitwise operators . . . . .	108
15.13.4 <code>Number</code> . <code>*</code> data properties . . . . .	109
15.13.5 <code>Number</code> . <code>*</code> methods . . . . .	109
15.13.6 <code>Number.prototype</code> . <code>*</code> . . . . .	111
15.13.7 Sources . . . . .	112

---

This chapter covers JavaScript’s single type for numbers, `number`.

## 15.1 JavaScript only has floating point numbers

You can express both integers and floating point numbers in JavaScript:

```
98
123.45
```

However, there is only a single type for all numbers: They are all *doubles*, 64-bit floating point numbers implemented according to the IEEE Standard for Floating-Point Arithmetic (IEEE 754).

Integers are simply floating point numbers without a decimal fraction:

```
> 98 === 98.0
true
```

Note that, under the hood, most JavaScript engines are often still able to use real integers, with all associated performance and storage size benefits.

## 15.2 Number literals

Let’s examine literals for numbers.

### 15.2.1 Integer literals

Several *integer literals* let you express integers with various bases:

```
// Binary (base 2)
assert.equal(0b11, 3);

// Octal (base 8)
assert.equal(0o10, 8);

// Decimal (base 10):
assert.equal(35, 35);

// Hexadecimal (base 16)
assert.equal(0xE7, 231);
```



### 15.2.2 Floating point literals

Floating point numbers can only be expressed in base 10.

Fractions:

```
> 35.0
35
```

Exponent:  $eN$  means  $\times 10^N$

```
> 3e2
300
> 3e-2
0.03
> 0.3e2
30
```

### 15.2.3 Syntactic pitfall: properties of integer literals

Accessing a property of an integer literal entails a pitfall: If the integer literal is immediately followed by a dot then that dot is interpreted as a decimal dot:

```
7.toString(); // syntax error
```

There are four ways to work around this pitfall:

```
7.0.toString()
(7).toString()
7..toString()
7 .toString() // space before dot
```

## 15.3 Number operators

### 15.3.1 Binary arithmetic operators

```
assert.equal(1 + 4, 5); // addition
assert.equal(6 - 3, 3); // subtraction

assert.equal(2 * 1.25, 2.5); // multiplication
assert.equal(6 / 4, 1.5); // division
assert.equal(6 % 4, 2); // remainder

assert.equal(2 ** 3, 8); // exponentiation
```

`%` is a remainder operator (not a modulo operator) – its result has the sign of the first operand:

```
> 3 % 2
1
> -3 % 2
-1
```

### 15.3.2 Unary plus and negation

```
assert.equal(+(-3), -3); // unary plus
assert.equal(-(-3), 3); // unary negation
```

Both operators coerce their operands to numbers:

```
> +'123'
123
```

### 15.3.3 Incrementing (++) and decrementing (--)

The incrementation operator ++ exists in a prefix version and a suffix version and destructively adds one to its operand. Therefore, its operand must be a storage location, so that it can be changed. The decrementation operator -- works the same, but subtracts one from its operand. The next two examples explain the difference between the prefix and the suffix versions.

Prefix ++ and prefix --: change and then return.

```
let foo = 3;
assert.equal(++foo, 4);
assert.equal(foo, 4);
```

```
let bar = 3;
assert.equal(--bar, 2);
assert.equal(bar, 2);
```

Suffix ++ and prefix --: return and then change.

```
let foo = 3;
assert.equal(foo++, 3);
assert.equal(foo, 4);
```

```
let bar = 3;
assert.equal(bar--, 3);
assert.equal(bar, 2);
```

#### 15.3.3.1 Operands: not just variables

You can also apply these operators to property values:

```
const obj = { a: 1 };
++obj.a;
assert.equal(obj.a, 2);
```

And to Array elements:

```
const arr = [ 4 ];
arr[0]++;
assert.deepEqual(arr, [5]);
```

**Exercise: Number operators**`exercises/numbers-math/is_odd_test.js`

## 15.4 Converting to number

Three ways of converting values to numbers:

- `Number(value)`
- `+value`
- `parseFloat(value)` (avoid; different than other two!)

Recommendation: use the descriptive `Number()`.

Examples:

```
assert.equal(Number(undefined), NaN);
assert.equal(Number(null), 0);

assert.equal(Number(false), 0);
assert.equal(Number(true), 1);

assert.equal(Number(123), 123);

assert.equal(Number(''), 0);
assert.equal(Number('123'), 123);
assert.equal(Number('xyz'), NaN);
```

How objects are converted to numbers can be configured via several special methods. For example, `.valueOf()`:

```
> Number({ valueOf() { return 123 } })
123
```

**Exercise: Converting to number**`exercises/numbers-math/parse_number_test.js`

## 15.5 Error values

Two number values are returned when errors happen:

- `NaN`
- `Infinity`

## 15.6 Error value: NaN

NaN is an abbreviation of “not a number”. Ironically, JavaScript considers it to be a number:

```
> typeof NaN
'number'
```

When is NaN returned?

NaN is returned if a number can’t be parsed:

```
> Number('$$$')
NaN
> Number(undefined)
NaN
```

NaN is returned if an operation can’t be performed:

```
> Math.log(-1)
NaN
> Math.sqrt(-1)
NaN
```

NaN is returned if an operand or argument is NaN (to propagate errors):

```
> NaN - 3
NaN
> 7 ** NaN
NaN
```

### 15.6.1 Checking for NaN

NaN is the only JavaScript value that is not strictly equal to itself:

```
const n = NaN;
assert.equal(n === n, false);
```

These are several ways of checking if a value *x* is NaN:

```
const x = NaN;

assert.ok(Number.isNaN(x)); // preferred
assert.ok(x !== x);
assert.ok(Object.is(x, NaN));
```

### 15.6.2 Finding NaN in Arrays

Some Array methods can’t find NaN:

```
> [NaN].indexOf(NaN)
-1
```

Others can:

```

> [NaN].includes(NaN)
true
> [NaN].findIndex(x => Number.isNaN(x))
0
> [NaN].find(x => Number.isNaN(x))
NaN

```

## 15.7 Error value: Infinity

When is the error value Infinity returned?

Infinity is returned if a number is too large:

```

> Math.pow(2, 1023)
8.98846567431158e+307
> Math.pow(2, 1024)
Infinity

```

Infinity is returned if there is a division by zero:

```

> 5 / 0
Infinity
> -5 / 0
-Infinity

```

### 15.7.1 Infinity as a default value

Infinity is larger than all other numbers (except NaN), making it a good default value:

```

function findMinimum(numbers) {
  let min = Infinity;
  for (const n of numbers) {
    if (n < min) min = n;
  }
  return min;
}

assert.equal(findMinimum([5, -1, 2]), -1);
assert.equal(findMinimum([]), Infinity);

```

### 15.7.2 Checking for Infinity

These are two common ways of checking if a value x is Infinity:

```

const x = Infinity;

assert.ok(x === Infinity);
assert.ok(!Number.isFinite(x));

```

**Exercise: Comparing numbers**

`exercises/numbers-math/find_max_test.js`

## 15.8 The precision of numbers: careful with decimal fractions

Internally, JavaScript floating point numbers are represented with base 2 (according to the IEEE 754 standard). That means that decimal fractions (base 10) can't always be represented precisely:

```
> 0.1 + 0.2
0.30000000000000004
> 1.3 * 3
3.9000000000000004
> 1.4 * 1000000000000000
1399999999999999.98
```

You therefore need to take rounding errors into consideration when performing arithmetic in JavaScript. Read on for an explanation of this phenomenon.

**Quiz: basic**

[See quiz app.](#)

## 15.9 (Advanced)

All remaining sections of this chapter are advanced.

## 15.10 Background: floating point precision

In this section, we explore how JavaScript represents floating point numbers internally. It uses three numbers to do so (which take up a total of 64 bits of storage):

- the sign (1 bit)
- the fraction (52 bits)
- the exponent (11 bits)

The value of a represented number is computed as follows:

$$(-1)^{\text{sign}} \times 0b1.\text{fraction} \times 2^{\text{exponent}}$$

To make things easier to understand, we make two changes:

- Second component: we switch from a fraction to a *mantissa* (an integer).
- Third component: we switch from base 2 (binary) to base 10 (decimal).

How does this representation work for numbers with *decimals* (digits after the decimal point)? We move the trailing point of an integer (the mantissa), by multiplying it with 10 to the power of a negative exponent.

For example, this is how we move the trailing point of 15 so that it becomes 1.5:

```
> 15 * (10 ** -1)
1.5
```

This is another example. This time, we move the point by two digits:

```
> 325 * (10 ** -2)
3.25
```

If we write negative exponents as fractions with positive exponents, we can see why some fractions can be represented as floating point numbers, while others can't:

- Base 10: mantissa /  $10^{\text{exponent}}$ .
  - Can be represented:  $1/10$
  - Can be represented:  $1/2 (=5/10)$
  - Cannot be represented:  $1/3$ 
    - \* Why? We can't get a three into the denominator.
- Base 2: mantissa /  $2^{\text{exponent}}$ .
  - Can be represented:  $1/2$
  - Can be represented:  $1/4$
  - Cannot be represented:  $1/10 (=1/(2 \times 5))$ 
    - \* Why? We can't get a five into the denominator.

## 15.11 Integers in JavaScript

Integers are simply (floating point) numbers without a decimal fraction:

```
> 1 === 1.0
true
> Number.isInteger(1.0)
true
```

### 15.11.1 Converting to integer

The recommended way of converting numbers to integers is to use one of the rounding methods of the `Math` object (which is documented [in the next chapter](#)):

- `Math.floor()`: closest lower integer
- `Math.ceil()`: closest higher integer
- `Math.round()`: closest integer (.5 is rounded up). For example:
  - `Math.round(2.5)` rounds up to 3.
  - `Math.round(-2.5)` rounds up to -2.
- `Math.trunc()`: remove the fraction

Tbl. 15.1 shows the results of these functions for various inputs.

Table 15.1: Functions for converting numbers to integers.

	-2.9	-2.5	-2.1	2.1	2.5	2.9
<code>Math.floor</code>	-3	-3	-3	2	2	2
<code>Math.ceil</code>	-2	-2	-2	3	3	3
<code>Math.round</code>	-3	-2	-2	2	3	3
<code>Math.trunc</code>	-2	-2	-2	2	2	2

### 15.11.2 Ranges of integers in JavaScript

It's good to be aware of the following ranges of integers in JavaScript:

- **Safe integers:** can be represented “safely” by JavaScript (more on what that means next)
  - Precision: 53 bits plus sign
  - Range:  $(-2^{53}, 2^{53})$
- **Array indices**
  - Precision: 32 bits, unsigned
  - Range:  $[0, 2^{32}-1)$  (excluding the maximum length)
  - Typed Arrays have a larger range of 53 bits (safe and unsigned)
- **Bitwise operands** (bitwise Or etc.)
  - Precision: 32 bits
  - Range of unsigned right shift ( $\gg$ ): unsigned,  $[0, 2^{32})$
  - Range of all other bitwise operators: signed,  $[-2^{31}, 2^{31})$

### 15.11.3 Safe integers

Recall that this is how JavaScript represents floating point numbers:

$$(-1)^{\text{sign}} \times 0b1.\text{fraction} \times 2^{\text{exponent}}$$

For integers, JavaScript mainly relies on the fraction. Once integers grow beyond the capacity of the fraction, some of them can still be represented as numbers (with the help of the exponent), but there are now gaps between them.

For example, the smallest positive *unsafe* integer is `2 ** 53`:

```
> 2 ** 53
9007199254740992
> 9007199254740992 + 1
9007199254740992
```

We can see that the JavaScript number 9007199254740992 represents both the corresponding integer and the corresponding integer plus one. That is, at this point, only every second integer can be represented precisely by JavaScript.

The following properties of `Number` help determine if an integer is safe:

- `Number.isSafeInteger(number)`
- `Number.MIN_SAFE_INTEGER = (2 ** 53) - 1`
- `Number.MAX_SAFE_INTEGER = -Number.MIN_SAFE_INTEGER`



`Number.isSafeInteger()` can be implemented as follows.

```
function isSafeInteger(n) {
  return (typeof n === 'number' &&
    Math.trunc(n) === n &&
    Number.MIN_SAFE_INTEGER <= n &&
    n <= Number.MAX_SAFE_INTEGER);
}
```

### 15.11.3.1 Safe computations

Let's look at computations involving unsafe integers.

The following result is incorrect and unsafe, even though both of its operands are safe.

```
> 9007199254740990 + 3
9007199254740992
```

The following result is incorrect, but safe. Only one of the operands is unsafe.

```
> 9007199254740995 - 10
9007199254740986
```

Therefore, the result of an expression `a op b` is correct if and only if:

```
isSafeInteger(a) && isSafeInteger(b) && isSafeInteger(a op b)
```



#### Exercise: Safe integers

`exercises/numbers-math/is_safe_integer_test.js`

## 15.12 Bitwise operators

JavaScript's bitwise operators work as follows:

- First, the operands are converted to numbers (64-bit double floating point numbers), then to 32-bit integers.
- Then the bitwise operation is performed.
- Last, the result is converted back to a double and returned.

Internally, the operators use the following integer ranges (input and output):

- Unsigned right shift operator (`>>>`): 32 bits, range  $[0, 2^{32})$
- All other bitwise operators: 32 bits including a sign, range  $[-2^{31}, 2^{31})$

### 15.12.1 Binary bitwise operators







### 15.13.4 `Number.*` data properties

- `Number.EPSILON`: `number` <sup>[ES6]</sup>

The difference between 1 and the next representable floating point number. In general, a machine epsilon<sup>1</sup> provides an upper bound for rounding errors in floating point arithmetic.

– Approximately:  $2.2204460492503130808472633361816 \times 10^{-16}$

- `Number.MAX_SAFE_INTEGER`: `number` <sup>[ES6]</sup>

The largest integer that JavaScript can represent uniquely. The same as  $(2^{53} - 1)$ .

- `Number.MAX_VALUE`: `number` <sup>[ES1]</sup>

The largest positive finite JavaScript number.

– Approximately:  $1.7976931348623157 \times 10^{308}$

- `Number.MIN_SAFE_INTEGER`: `number` <sup>[ES6]</sup>

The smallest integer that JavaScript can represent precisely (with smaller integers, more than one integer is represented by the same number). The same as `-Number.MAX_SAFE_INTEGER`.

- `Number.MIN_VALUE`: `number` <sup>[ES1]</sup>

The smallest positive JavaScript number. Approximately  $5 \times 10^{-324}$ .

- `Number.NaN`: `number` <sup>[ES1]</sup>

The same as the global variable `NaN`.

- `Number.NEGATIVE_INFINITY`: `number` <sup>[ES1]</sup>

The same as `-Number.POSITIVE_INFINITY`.

- `Number.POSITIVE_INFINITY`: `number` <sup>[ES1]</sup>

The same as the global variable `Infinity`.

### 15.13.5 `Number.*` methods

- `Number.isFinite(num: number)`: `boolean` <sup>[ES6]</sup>

Returns true if `num` is an actual number (neither `Infinity` nor `-Infinity` nor `NaN`).

```
> Number.isFinite(Infinity)
false
> Number.isFinite(-Infinity)
false
> Number.isFinite(NaN)
false
> Number.isFinite(123)
true
```

This method does not coerce its parameter to number (whereas the global function `isFinite()` does):

---

<sup>1</sup>[https://en.wikipedia.org/wiki/Machine\\_epsilon](https://en.wikipedia.org/wiki/Machine_epsilon)

```
> isFinite('123')
true
> Number.isFinite('123')
false
```

- `Number.isInteger(num: number): boolean` <sup>[ES6]</sup>

Returns true if num is a number and does not have a decimal fraction.

```
> Number.isInteger(-17)
true
> Number.isInteger(33)
true
> Number.isInteger(33.1)
false
> Number.isInteger('33')
false
> Number.isInteger(NaN)
false
> Number.isInteger(Infinity)
false
```

- `Number.isNaN(num: number): boolean` <sup>[ES6]</sup>

Returns true if num is the value NaN. In contrast to the global function `isNaN()`, this method does not coerce its parameter to number:

```
> isNaN('???')
true
> Number.isNaN('???')
false
```

- `Number.isSafeInteger(num: number): boolean` <sup>[ES6]</sup>

Returns true if num is a number and uniquely represents an integer.

- `Number.parseFloat(str: string): number` <sup>[ES6]</sup>

Coerces its parameter to string and parses it as a floating point number. Works the same as the global function `parseFloat()`. For converting strings to numbers, `Number()` (which ignores leading and trailing whitespace) is usually a better choice than `Number.parseFloat()` (which ignores leading whitespace and any illegal trailing characters and can hide problems).

```
> Number.parseFloat(' 123.4#')
123.4
> Number(' 123.4#')
NaN
```

- `Number.parseInt(str: string, radix=10): number` <sup>[ES6]</sup>

Coerces its parameter to string and parses it as an integer, ignoring illegal trailing characters. Same as the global function `parseInt()`.

```
> Number.parseInt('101#', 2)
5
> Number.parseInt('FF', 16)
```

255

Do not use this method to convert numbers to integers (use the rounding methods of `Math`, instead): it is inefficient and can produce incorrect results:

```
> Number.parseInt(1e21, 10) // wrong
1
> Math.trunc(1e21) // correct
1e+21
```

### 15.13.6 `Number.prototype.*`

- `toExponential(fractionDigits?: number): string` <sup>[ES3]</sup>

Returns a string that represents the number via exponential notation. With `fractionDigits`, you can specify, how many digits should be shown of the number that is multiplied with the exponent (the default is to show as many digits as necessary).

Example: number too small to get a positive exponent via `.toString()`.

```
> 1234..toString()
'1234'

> 1234..toExponential()
'1.234e+3'
> 1234..toExponential(5)
'1.23400e+3'
```

Example: fraction not small enough to get a negative exponent via `.toString()`.

```
> 0.003.toString()
'0.003'

> 0.003.toExponential()
'3e-3'
> 0.003.toExponential(4)
'3.0000e-3'
```

- `toFixed(fractionDigits=0): string` <sup>[ES3]</sup>

Returns an exponent-free representation of the number, rounded to `fractionDigits` digits.

```
> 0.0000003.toString()
'3e-7'

> 0.0000003.toFixed(10)
'0.0000003000'
> 0.0000003.toFixed()
'0'
```

If the number is  $10^{21}$  or greater, even `.toFixed()` uses an exponent:

```
> (10 ** 21).toFixed()
'1e+21'
```

- `toPrecision(precision?: number): string` <sup>[ES3]</sup>

Works like `.toString()`, but prunes the mantissa to precision digits before returning a result. If precision is missing, `.toString()` is used.

```
> 1234..toPrecision(3) // requires exponential notation
'1.23e+3'
```

```
> 1234..toPrecision(4)
'1234'
```

```
> 1234..toPrecision(5)
'1234.0'
```

```
> 1.234.toPrecision(3)
'1.23'
```

- `toString(radix=10): string` <sup>[ES1]</sup>

Returns a string representation of the number.

Returning a result with base 10:

```
> 123.456.toString()
'123.456'
```

Returning results with bases other than 10 (specified via `radix`):

```
> 4..toString(2)
'100'
```

```
> 4.5.toString(2)
'100.1'
```

```
> 255..toString(16)
'ff'
```

```
> 255.66796875.toString(16)
'ff.ab'
```

```
> 1234567890..toString(36)
'kf12oi'
```

You can use `parseInt()` to convert integer results back to numbers:

```
> parseInt('kf12oi', 36)
1234567890
```

### 15.13.7 Sources

- Wikipedia
- TypeScript's built-in typings<sup>2</sup>
- MDN web docs for JavaScript<sup>3</sup>

<sup>2</sup><https://github.com/Microsoft/TypeScript/blob/master/lib/>

<sup>3</sup><https://developer.mozilla.org/en-US/docs/Web/JavaScript>



- ECMAScript language specification<sup>4</sup>



Quiz: advanced

See quiz app.

---

<sup>4</sup><https://tc39.github.io/ecma262/>



# Chapter 16

## Math

### Contents

16.1 Data properties . . . . .	115
16.2 Exponents, roots, logarithms . . . . .	116
16.3 Rounding . . . . .	117
16.4 Trigonometric Functions . . . . .	118
16.5 asm.js helpers . . . . .	120
16.6 Various other functions . . . . .	120
16.7 Sources . . . . .	121

Math is an object with data properties and methods for processing numbers.

You can see it as a poor man's module. Today, it would probably be created as a module, but it has existed since long before modules.

### 16.1 Data properties

- `Math.E`: number <sup>[ES1]</sup>  
Euler's number, base of the natural logarithms, approximately 2.7182818284590452354.
- `Math.LN10`: number <sup>[ES1]</sup>  
The natural logarithm of 10, approximately 2.302585092994046.
- `Math.LN2`: number <sup>[ES1]</sup>  
The natural logarithm of 2, approximately 0.6931471805599453.
- `Math.LOG10E`: number <sup>[ES1]</sup>  
The logarithm of  $e$  to base 10, approximately 0.4342944819032518.
- `Math.LOG2E`: number <sup>[ES1]</sup>  
The logarithm of  $e$  to base 2, approximately 1.4426950408889634.

- `Math.PI`: `number` <sup>[ES1]</sup>

The mathematical constant  $\pi$ , ratio of a circle's circumference to its diameter, approximately 3.1415926535897932.

- `Math.SQRT1_2`: `number` <sup>[ES1]</sup>

The square root of  $1/2$ , approximately 0.7071067811865476.

- `Math.SQRT2`: `number` <sup>[ES1]</sup>

The square root of 2, approximately 1.4142135623730951.

## 16.2 Exponents, roots, logarithms

- `Math.cbrt(x: number): number` <sup>[ES6]</sup>

Returns the cube root of  $x$ .

```
> Math.cbrt(8)
2
```

- `Math.exp(x: number): number` <sup>[ES1]</sup>

Returns  $e^x$  ( $e$  being Euler's number). The inverse of `Math.log()`.

```
> Math.exp(0)
1
> Math.exp(1) === Math.E
true
```

- `Math.expm1(x: number): number` <sup>[ES6]</sup>

Returns `Math.exp(x) - 1`. The inverse of `Math.log1p()`. Very small numbers (fractions close to 0) are represented with a higher precision. This function returns such values whenever the result of `.exp()` is close to 1.

- `Math.log(x: number): number` <sup>[ES1]</sup>

Returns the natural logarithm of  $x$  (to base  $e$ , Euler's number). The inverse of `Math.exp()`.

```
> Math.log(1)
0
> Math.log(Math.E)
1
> Math.log(Math.E ** 2)
2
```

- `Math.log1p(x: number): number` <sup>[ES6]</sup>

Returns `Math.log(1 + x)`. The inverse of `Math.expm1()`. Very small numbers (fractions close to 0) are represented with a higher precision. This function receives such numbers whenever a parameter to `.log()` is close to 1.

- `Math.log10(x: number): number` <sup>[ES6]</sup>

Returns the logarithm of  $x$  to base 10. The inverse of `10 ** x`.

```
> Math.log10(1)
0
> Math.log10(10)
1
> Math.log10(100)
2
```

- `Math.log2(x: number): number` <sup>[ES6]</sup>

Returns the logarithm of  $x$  to base 2. The inverse of  $2^{**} x$ .

```
> Math.log2(1)
0
> Math.log2(2)
1
> Math.log2(4)
2
```

- `Math.pow(x: number, y: number): number` <sup>[ES1]</sup>

Returns  $x^y$ ,  $x$  to the power of  $y$ . The same as  $x^{**} y$ .

```
> Math.pow(2, 3)
8
> Math.pow(25, 0.5)
5
```

- `Math.sqrt(x: number): number` <sup>[ES1]</sup>

Returns the square root of  $x$ . The inverse of  $x^{**} 2$ .

```
> Math.sqrt(9)
3
```

## 16.3 Rounding

Rounding means converting an arbitrary number to an integer (a number without a decimal fraction).

Tbl. 16.1 lists the available functions and what they return for a few representative inputs.

Table 16.1: Rounding functions of `Math`.

	-2.9	-2.5	-2.1	2.1	2.5	2.9
<code>Math.floor</code>	-3	-3	-3	2	2	2
<code>Math.ceil</code>	-2	-2	-2	3	3	3
<code>Math.round</code>	-3	-2	-2	2	3	3
<code>Math.trunc</code>	-2	-2	-2	2	2	2

- `Math.ceil(x: number): number` <sup>[ES1]</sup>

Returns the smallest (closest to  $-\infty$ ) integer  $i$  with  $x \leq i$ .

```
> Math.ceil(1.9)
```

```
2
> Math.ceil(2.1)
3
```

- `Math.floor(x: number): number` <sup>[ES1]</sup>

Returns the greatest (closest to  $+\infty$ ) integer  $i$  with  $i \leq x$ .

```
> Math.floor(1.9)
1
> Math.floor(2.1)
2
```

- `Math.round(x: number): number` <sup>[ES1]</sup>

Returns the integer that is closest to  $x$ . If the decimal fraction of  $x$  is  $.5$  then `.round()` rounds up (to the integer closer to positive infinity):

```
> Math.round(2.5)
3
> Math.round(-2.5)
-2
```

- `Math.trunc(x: number): number` <sup>[ES6]</sup>

Removes the decimal fraction of  $x$  and returns the resulting integer.

```
> Math.trunc(1.9)
1
> Math.trunc(2.1)
2
```

## 16.4 Trigonometric Functions

All angles are specified in radians. Use the following two functions to convert between degrees and radians.

```
function toRadians(degrees) {
  return degrees / 180 * Math.PI;
}
function toDegrees(radians) {
  return radians / Math.PI * 180;
}
```

- `Math.acos(x: number): number` <sup>[ES1]</sup>

Returns the arc cosine (inverse cosine) of  $x$ .

```
> Math.acos(0)
1.5707963267948966
> Math.acos(1)
0
```

- `Math.acosh(x: number): number` <sup>[ES6]</sup>  
Returns the inverse hyperbolic cosine of `x`.
- `Math.asin(x: number): number` <sup>[ES1]</sup>  
Returns the arc sine (inverse sine) of `x`.  
  

```
> Math.asin(0)
0
> Math.asin(1)
1.5707963267948966
```
- `Math.asinh(x: number): number` <sup>[ES6]</sup>  
Returns the inverse hyperbolic sine of `x`.
- `Math.atan(x: number): number` <sup>[ES1]</sup>  
Returns the arc tangent (inverse tangent) of `x`.
- `Math.atanh(x: number): number` <sup>[ES6]</sup>  
Returns the inverse hyperbolic tangent of `x`.
- `Math.atan2(y: number, x: number): number` <sup>[ES1]</sup>  
Returns the arc tangent of the quotient `y/x`.
- `Math.cos(x: number): number` <sup>[ES1]</sup>  
Returns the cosine of `x`.  
  

```
> Math.cos(0)
1
> Math.cos(Math.PI)
-1
```
- `Math.cosh(x: number): number` <sup>[ES6]</sup>  
Returns the hyperbolic cosine of `x`.
- `Math.hypot(...values: number[]): number` <sup>[ES6]</sup>  
Returns the square root of the sum of the squares of `values` (Pythagoras' theorem):  
  

```
> Math.hypot(3, 4)
5
```
- `Math.sin(x: number): number` <sup>[ES1]</sup>  
Returns the sine of `x`.  
  

```
> Math.sin(0)
0
> Math.sin(Math.PI / 2)
1
```
- `Math.sinh(x: number): number` <sup>[ES6]</sup>  
Returns the hyperbolic sine of `x`.

- `Math.tan(x: number): number` <sup>[ES1]</sup>

Returns the tangent of x.

```
> Math.tan(0)
0
> Math.tan(1)
1.5574077246549023
```

- `Math.tanh(x: number): number`; <sup>[ES6]</sup>

Returns the hyperbolic tangent of x.

## 16.5 asm.js helpers

WebAssembly is a virtual machine based on JavaScript that is supported by most JavaScript engines.

asm.js is a precursor to WebAssembly. It is a subset of JavaScript that produces fast executables if static languages (such as C++) are compiled to it. In a way, it is also a virtual machine, within the confines of JavaScript.

The following two methods help asm.js and have few use cases, otherwise.

- `Math.fround(x: number): number` <sup>[ES6]</sup>

Rounds x to a 32-bit floating point value (`float`). Used by asm.js to tell an engine to internally use a `float` value (normal numbers are doubles and take up 64 bits).

- `Math.imul(x: number, y: number): number` <sup>[ES6]</sup>

Multiplies the two 32 bit integers x and y and returns the lower 32 bits of the result. Needed by asm.js: All other basic 32-bit math operations can be simulated by coercing 64-bit results to 32 bits. With multiplication, you may lose bits for results beyond 32 bits.

## 16.6 Various other functions

- `Math.abs(x: number): number` <sup>[ES1]</sup>

Returns the absolute value of x.

```
> Math.abs(3)
3
> Math.abs(-3)
3
> Math.abs(0)
0
```

- `Math.clz32(x: number): number` <sup>[ES6]</sup>

Counts the leading zero bits in the 32-bit integer x. Used in DSP algorithms.

```
> Math.clz32(0b01000000000000000000000000000000)
1
```



```

> Math.clz32(0b00100000000000000000000000000000)
2
> Math.clz32(2)
30
> Math.clz32(1)
31

```

- `Math.max(...values: number[]): number` <sup>[ES1]</sup>

Converts values to numbers and returns the largest one.

```

> Math.max(3, -5, 24)
24

```

- `Math.min(...values: number[]): number` <sup>[ES1]</sup>

Converts values to numbers and returns the smallest one.

```

> Math.min(3, -5, 24)
-5

```

- `Math.random(): number` <sup>[ES1]</sup>

Returns a pseudo-random number  $n$  where  $0 \leq n < 1$ .

Computing a random integer  $i$  where  $0 \leq i < \text{max}$ :

```

function getRandomInteger(max) {
  return Math.floor(Math.random() * max);
}

```

- `Math.sign(x: number): number` <sup>[ES6]</sup>

Returns the sign of a number:

```

> Math.sign(-8)
-1
> Math.sign(0)
0
> Math.sign(3)
1

```

## 16.7 Sources

- Wikipedia
- TypeScript's built-in typings<sup>1</sup>
- MDN web docs for JavaScript<sup>2</sup>
- ECMAScript language specification<sup>3</sup>

<sup>1</sup><https://github.com/Microsoft/TypeScript/blob/master/lib/>

<sup>2</sup><https://developer.mozilla.org/en-US/docs/Web/JavaScript>

<sup>3</sup><https://tc39.github.io/ecma262/>



# Chapter 17

## Strings

### Contents

---

<b>17.1 Plain string literals</b>	<b>124</b>
17.1.1 Escaping	124
<b>17.2 Accessing characters and code points</b>	<b>124</b>
17.2.1 Accessing JavaScript characters	124
17.2.2 Accessing Unicode code points via <code>for-of</code> and <code>spreading</code>	124
<b>17.3 String concatenation via <code>+</code></b>	<b>125</b>
<b>17.4 Converting to string</b>	<b>125</b>
17.4.1 Stringifying objects	126
17.4.2 Customizing the stringification of objects	126
17.4.3 An alternate way of stringifying values	127
<b>17.5 Comparing strings</b>	<b>127</b>
<b>17.6 Atoms of text: JavaScript characters, code points, grapheme clusters</b>	<b>128</b>
17.6.1 Working with code points	128
17.6.2 Working with code units	129
17.6.3 Caveat: grapheme clusters	129
<b>17.7 Quick reference: Strings</b>	<b>129</b>
17.7.1 Converting to string	130
17.7.2 Numeric values of characters and code points	131
17.7.3 String operators	131
17.7.4 <code>String.prototype</code> : finding and matching	131
17.7.5 <code>String.prototype</code> : extracting	133
17.7.6 <code>String.prototype</code> : combining	134
17.7.7 <code>String.prototype</code> : transforming	135
17.7.8 <code>String.prototype</code> : chars, char codes, code points	136
17.7.9 Sources	137

---

Strings are primitive values in JavaScript and immutable. That is, string-related operations always produce new strings and never change existing strings.

## 17.1 Plain string literals

Plain string literals are delimited by either single quotes or double quotes:

```
const str1 = 'abc';
const str2 = "abc";
assert.equal(str1, str2);
```

Single quotes are used more often, because it makes it easier to mention HTML with its double quotes.

The next chapter covers *template literals*, which give you:

- String interpolation
- Multiple lines
- Raw string literals (backslash has no special meaning)

### 17.1.1 Escaping

The backslash lets you create special characters:

- Unix line break: `'\n'`
- Windows line break: `'\r\n'`
- Tab: `'\t'`
- Backslash: `'\\'`

The backslash also lets you use the delimiter of a string literal inside that literal:

```
'She said: "Let\'s go!"'
"She said: \"Let's go!\""
```

## 17.2 Accessing characters and code points

### 17.2.1 Accessing JavaScript characters

JavaScript has no extra data type for characters – characters are always transported as strings.

```
const str = 'abc';

// Reading a character at a given index
assert.equal(str[1], 'b');

// Counting the characters in a string:
assert.equal(str.length, 3);
```

### 17.2.2 Accessing Unicode code points via for-of and spreading

Iterating over strings via for-of or spreading (...) visits Unicode code points. Each code point is represented by 1–2 JavaScript characters. For more information, see [the section on the atoms of text](#) in this chapter.

This is how you iterate over the code points of a string via `for-of`:

```
for (const ch of 'abc') {
  console.log(ch);
}
// Output:
// 'a'
// 'b'
// 'c'
```

And this is how you convert a string into an Array of code points via spreading:

```
assert.deepEqual([...'abc'], ['a', 'b', 'c']);
```

## 17.3 String concatenation via +

If at least one operand is a string, the plus operator (+) converts any non-strings to strings and concatenates the result:

```
assert.equal(3 + ' times ' + 4, '3 times 4');
```

The assignment operator += is useful if you want to assemble a string, piece by piece:

```
let str = ''; // must be `let`!
str += 'Say it';
str += ' one more';
str += ' time';

assert.equal(str, 'Say it one more time');
```

As an aside, this way of assembling strings is quite efficient, because most JavaScript engines internally optimize it.



### Exercise: Concatenating strings

exercises/strings/concat\_string\_array\_test.js

## 17.4 Converting to string

These are three ways of converting a value `x` to a string:

- `String(x)`
- `''+x`
- `x.toString()` (does not work for `undefined` and `null`)

Recommendation: use the descriptive and safe `String()`.

Examples:

```
assert.equal(String(undefined), 'undefined');
assert.equal(String(null), 'null');
```

```
assert.equal(String(false), 'false');
assert.equal(String(true), 'true');

assert.equal(String(123.45), '123.45');
```

Pitfall for booleans: If you convert a boolean to a string via `String()`, you can't convert it back via `Boolean()`.

```
> String(false)
'false'
> Boolean('false')
true
```

### 17.4.1 Stringifying objects

Plain objects have a default representation that is not very useful:

```
> String({a: 1})
'[object Object]'
```

Arrays have a better string representation, but it still hides much information:

```
> String(['a', 'b'])
'a,b'
> String(['a', ['b']])
'a,b'
```

```
> String([1, 2])
'1,2'
> String(['1', '2'])
'1,2'
```

```
> String([true])
'true'
> String(['true'])
'true'
> String(true)
'true'
```

Stringifying functions returns their source code:

```
> String(function f() {return 4})
'function f() {return 4}'
```

### 17.4.2 Customizing the stringification of objects

You can override the built-in way of stringifying objects by implementing the method `toString()`:

```
const obj = {
  toString() {
```

```

    return 'hello';
  }
};

assert.equal(String(obj), 'hello');
```

### 17.4.3 An alternate way of stringifying values

The JSON data format is a text representation of JavaScript values. Therefore, `JSON.stringify()` can also be used to stringify data:

```

> JSON.stringify({a: 1})
'{"a":1}'
> JSON.stringify(['a', ['b']])
'["a",["b"]]'
```

The caveat is that JSON only supports `null`, booleans, numbers, strings, Arrays and objects (which it always treats as if they were created by object literals).

Tip: The third parameter lets you switch on multi-line output and specify how much to indent. For example:

```
console.log(JSON.stringify({first: 'Jane', last: 'Doe'}, null, 2));
```

This statement produces the following output.

```

{
  "first": "Jane",
  "last": "Doe"
}
```

## 17.5 Comparing strings

Strings can be compared via the following operators:

```
< <= > >=
```

There is one important caveat to consider: These operators compare based on the numeric values of JavaScript characters. That means that the order that JavaScript uses for strings is different from the one used in dictionaries and phone books:

```

> 'A' < 'B' // ok
true
> 'a' < 'B' // not ok
false
> 'ä' < 'b' // not ok
false
```

Properly comparing text is beyond the scope of this book. It is supported via the ECMAScript Internationalization API<sup>1</sup> (Intl).

---

<sup>1</sup>[https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global\\_Objects/Intl](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Intl)

## 17.6 Atoms of text: JavaScript characters, code points, grapheme clusters

Quick recap of [the chapter on Unicode](#):

- Code points: Unicode characters, with a range of 21 bits.
- UTF-16 code units: JavaScript characters, with a range of 16 bits. Code points are encoded as 1–2 UTF-16 code units.
- Grapheme clusters: *Graphemes* are written symbols, as displayed on screen or paper. A *grapheme cluster* is a sequence of 1 or more code points that encodes a grapheme.

To represent code points in JavaScript strings, one or two JavaScript characters are used. You can see that when counting characters via `.length`:

```
// 3 Unicode code points, 3 JavaScript characters:
assert.equal('abc'.length, 3);

// 1 Unicode code point, 2 JavaScript characters:
assert.equal('㊦'.length, 2);
```

### 17.6.1 Working with code points

Let's explore JavaScript's tools for working with code points.

*Code point escapes* let you specify code points hexadecimally. They expand to one or two JavaScript characters.

```
> '\u{1F642}'
'㊦'
```

Converting from code points:

```
> String.fromCodePoint(0x1F642)
'㊦'
```

Converting to code points:

```
> '㊦'.codePointAt(0).toString(16)
'1f642'
```

Iteration honors code points. For example, the iteration-based `for-of` loop:

```
const str = '㊦a';
assert.equal(str.length, 3);

for (const codePoint of str) {
  console.log(codePoint);
}

// Output:
// '㊦'
// 'a'
```



Or iteration-based *spreading* (`...`):

```
> [...'@a']  
[ '@', 'a' ]
```

Spreading is therefore a good tool for counting code points:

```
> [...'@a'].length  
2  
> '@a'.length  
3
```

### 17.6.2 Working with code units

Indices and lengths of strings are based on JavaScript characters (i.e., code units).

To specify code units numerically, you can use *code unit escapes*:

```
> '\uD83D\uDE42'  
'@'
```

And you can use so-called *char codes*:

```
> String.fromCharCode(0xD83D) + String.fromCharCode(0xDE42)  
'@'
```

To get the char code of a character, use `.charCodeAt()`:

```
> '@'.charCodeAt(0).toString(16)  
'd83d'
```

### 17.6.3 Caveat: grapheme clusters

When working with text that may be written in any human language, it's best to split at the boundaries of grapheme clusters, not at the boundaries of code units.

TC39 is working on `Intl.Segmenter`<sup>2</sup>, a proposal for the ECMAScript Internationalization API to support Unicode segmentation (along grapheme cluster boundaries, word boundaries, sentence boundaries, etc.).

Until that proposal becomes a standard, you can use one of several libraries that are available (do a web search for “JavaScript grapheme”).

## 17.7 Quick reference: Strings

Strings are immutable, none of the string methods ever modify their strings.

---

<sup>2</sup><https://github.com/tc39/proposal-intl-segmenter>

### 17.7.1 Converting to string

Tbl. 17.1 describes how various values are converted to strings.

Table 17.1: Converting values to strings.

x	String(x)
undefined	'undefined'
null	'null'
Boolean value	false → 'false', true → 'true'
Number value	Example: 123 → '123'
String value	x (input, unchanged)
An object	Configurable via, e.g., toString()

### 17.7.2 Numeric values of characters and code points

- **Char codes:** Unicode UTF-16 code units as numbers
  - `String.fromCharCode()`, `String.prototype.charCodeAt()`
  - Precision: 16 bits, unsigned
- **Code points:** Unicode code points as numbers
  - `String.fromCodePoint()`, `String.prototype.codePointAt()`
  - Precision: 21 bits, unsigned (17 planes, 16 bits each)

### 17.7.3 String operators

```
// Access characters via []
const str = 'abc';
assert.equal(str[1], 'b');

// Concatenate strings via +
assert.equal('a' + 'b' + 'c', 'abc');
assert.equal('take ' + 3 + ' oranges', 'take 3 oranges');
```

### 17.7.4 String.prototype: finding and matching

- `.endsWith(searchString: string, endPos=this.length): boolean` <sup>[ES6]</sup>  
 Returns true if the string would end with `searchString` if its length were `endPos`. Returns false, otherwise.  

```
> 'foo.txt'.endsWith('.txt')
true
> 'abcde'.endsWith('cd', 4)
true
```
- `.includes(searchString: string, startPos=0): boolean` <sup>[ES6]</sup>  
 Returns true if the string contains the `searchString` and false, otherwise. The search starts at `startPos`.  

```
> 'abc'.includes('b')
true
```

```
> 'abc'.includes('b', 2)
false
```

- `.indexOf(searchString: string, minIndex=0): number` <sup>[ES1]</sup>

Returns the lowest index at which `searchString` appears within the string, or -1, otherwise. Any returned index will be `minIndex` or higher.

```
> 'abab'.indexOf('a')
0
> 'abab'.indexOf('a', 1)
2
> 'abab'.indexOf('c')
-1
```

- `.lastIndexOf(searchString: string, maxIndex=Infinity): number` <sup>[ES1]</sup>

Returns the highest index at which `searchString` appears within the string, or -1, otherwise. Any returned index will be `maxIndex` or lower.

```
> 'abab'.lastIndexOf('ab', 2)
2
> 'abab'.lastIndexOf('ab', 1)
0
> 'abab'.lastIndexOf('ab')
2
```

- `.match(regExp: string | RegExp): RegExpMatchArray | null` <sup>[ES3]</sup>

If `regExp` is a regular expression with flag `/g` not set, then `.match()` returns the first match for `regExp` within the string. Or `null` if there is no match. If `regExp` is a string, it is used to create a regular expression before performing the previous steps.

The result has the following type:

```
interface RegExpMatchArray extends Array<string> {
  index: number;
  input: string;
  groups: undefined | {
    [key: string]: string
  };
}
```

Numbered capture groups become Array indices. Named capture groups<sup>3</sup> (ES2018) become properties of `.groups`. In this mode, `.match()` works like `RegExp.prototype.exec()`.

Examples:

```
> 'ababb'.match(/a(b+)/)
{ 0: 'ab', 1: 'b', index: 0, input: 'ababb', groups: undefined }
> 'ababb'.match(/a(?<foo>b+)/)
{ 0: 'ab', 1: 'b', index: 0, input: 'ababb', groups: { foo: 'b' } }
> 'abab'.match(/x/)
null
```

---

<sup>3</sup>[http://exploringjs.com/es2018-es2019/ch\\_regexp-named-capture-groups.html](http://exploringjs.com/es2018-es2019/ch_regexp-named-capture-groups.html)

- `.match(regExp: RegExp): string[] | null` <sup>[ES3]</sup>

If flag `/g` of `regExp` is set, `.match()` returns either an Array with all matches or `null` if there was no match.

```
> 'ababb'.match(/a(b+)/g)
[ 'ab', 'abb' ]
> 'ababb'.match(/a(?<foo>b+)/g)
[ 'ab', 'abb' ]
> 'abab'.match(/x/g)
null
```

- `.search(regExp: string | RegExp): number` <sup>[ES3]</sup>

Returns the index at which `regExp` occurs within the string. If `regExp` is a string, it is used to create a regular expression.

```
> 'a2b'.search(/[0-9]/)
1
> 'a2b'.search('[0-9]')
1
```

- `.startsWith(searchString: string, startPos=0): boolean` <sup>[ES6]</sup>

Returns `true` if `searchString` occurs in the string at index `startPos`. Returns `false`, otherwise.

```
> '.gitignore'.startsWith('.')
true
> 'abcde'.startsWith('bc', 1)
true
```

### 17.7.5 String.prototype: extracting

- `.slice(start=0, end=this.length): string` <sup>[ES3]</sup>

Returns the substring of the string that starts at (including) index `start` and ends at (excluding) index `end`. You can use negative indices where `-1` means `this.length-1` (etc.).

```
> 'abc'.slice(1, 3)
'bc'
> 'abc'.slice(1)
'bc'
> 'abc'.slice(-2)
'bc'
```

- `.split(separator: string | RegExp, limit?: number): string[]` <sup>[ES3]</sup>

Splits the string into an Array of substrings – the strings that occur between the separators. The separator can either be a string or a regular expression. Captures made by groups in the regular expression are included in the result.

```
> 'abc'.split('')
[ 'a', 'b', 'c' ]
> 'a | b | c'.split('|')
[ 'a ', ' b ', ' c' ]
```

```
> 'a : b : c'.split(/ *: */)
[ 'a', 'b', 'c' ]
> 'a : b : c'.split(/( *):( */)
[ 'a', ' ', ' ', ' ', 'b', ' ', ' ', ' ', 'c' ]
```

- `.substring(start: number, end=this.length): string` <sup>[ES1]</sup>

Use `.slice()` instead of this method. `.substring()` wasn't implemented consistently in older engines and doesn't support negative indices.

### 17.7.6 String.prototype: combining

- `.concat(...strings: string[]): string` <sup>[ES3]</sup>

Returns the concatenation of the string and strings. `'a'+'b'` is equivalent to `'a'.concat('b')` and more concise.

```
> 'ab'.concat('cd', 'ef', 'gh')
'abcdefgh'
```

- `.padEnd(len: number, fillString=' '): string` <sup>[ES2017]</sup>

Appends `fillString` to the string until it has the desired length `len`.

```
> '#'.padEnd(2)
'# '
> 'abc'.padEnd(2)
'abc'
> '#'.padEnd(5, 'abc')
'#abca'
```

- `.padStart(len: number, fillString=' '): string` <sup>[ES2017]</sup>

Prepends `fillString` to the string until it has the desired length `len`.

```
> '#'.padStart(2)
' #'
> 'abc'.padStart(2)
'abc'
> '#'.padStart(5, 'abc')
'abca#'
```

- `.repeat(count=0): string` <sup>[ES6]</sup>

Returns a string that is the string, repeated `count` times.

```
> '*'.repeat()
''
> '*'.repeat(3)
'***'
```

### 17.7.7 String.prototype: transforming

- `.normalize(form: 'NFC'|'NFD'|'NFKC'|'NFKD' = 'NFC'): string` <sup>[ES6]</sup>

Normalizes the string according to the Unicode Normalization Forms<sup>4</sup>.

- `.replace(searchValue: string | RegExp, replaceValue: string): string` <sup>[ES3]</sup>

Replace matches of `searchValue` with `replaceValue`. If `searchValue` is a string, only the first verbatim occurrence is replaced. If `searchValue` is a regular expression without flag `/g`, only the first match is replaced. If `searchValue` is a regular expression with `/g` then all matches are replaced.

```
> 'x.x.'.replace('.', '#')
'x#x.'
> 'x.x.'.replace(/./, '#')
'#.x.'
> 'x.x.'.replace(/./g, '#')
'####'
```

Special characters in `replaceValue` are:

- `$$`: becomes `$`
- `$n`: becomes the capture of numbered group `n` (alas, `$0` does not work)
- `&$`: becomes the complete match
- `$``: becomes everything before the match
- `$'`: becomes everything after the match

Examples:

```
> 'a 2020-04 b'.replace(/([0-9]{4})-([0-9]{2})/, '|$2|')
'a |04| b'
> 'a 2020-04 b'.replace(/([0-9]{4})-([0-9]{2})/, '|$&|')
'a |2020-04| b'
> 'a 2020-04 b'.replace(/([0-9]{4})-([0-9]{2})/, '|$`|')
'a |a | b'
```

Named capture groups<sup>5</sup> (ES2018) are supported, too:

- `$<name>` becomes the capture of named group `name`

Example:

```
> 'a 2020-04 b'.replace(/(?<year>[0-9]{4})-(?<month>[0-9]{2})/, '|$<month>|')
'a |04| b'
```

- `.replace(searchValue: string | RegExp, replacer: (...args: any[]) => string): string` <sup>[ES3]</sup>

If the second parameter is a function occurrences are replaced with the strings it returns. Its parameters `args` are:

- `matched`: string: the complete match
- `g1`: string|undefined: the capture of numbered group 1
- `g2`: string|undefined: the capture of numbered group 2

<sup>4</sup><https://unicode.org/reports/tr15/>

<sup>5</sup>[http://exploringjs.com/es2018-es2019/ch\\_regexp-named-capture-groups.html](http://exploringjs.com/es2018-es2019/ch_regexp-named-capture-groups.html)

- (Etc.)
- offset: number: where was the match found in the input string?
- input: string: the whole input string

```
const regexp = /([0-9]{4})-([0-9]{2})/;
const replacer = (all, year, month) => '|' + all + '|';
assert.equal(
  'a 2020-04 b'.replace(regexp, replacer),
  'a |2020-04| b');
```

Named capture groups<sup>6</sup> (ES2018) are supported, too. If there are any, a last parameter contains an object whose properties contain the captures:

```
const regexp = /(?<year>[0-9]{4})-(?<month>[0-9]{2})/;
const replacer = (...args) => {
  const groups=args.pop();
  return '|' + groups.month + '|';
};
assert.equal(
  'a 2020-04 b'.replace(regexp, replacer),
  'a |04| b');
```

- `.toUpperCase(): string` <sup>[ES1]</sup>

Returns a copy of the string in which all lowercase alphabetic characters are converted to uppercase. How well that works for various alphabets depends on the JavaScript engine.

```
> '-a2b-'.toUpperCase()
'-A2B-'
> 'αβγ'.toUpperCase()
'ABΓ'
```

- `.toLowerCase(): string` <sup>[ES1]</sup>

Returns a copy of the string in which all uppercase alphabetic characters are converted to lowercase. How well that works for various alphabets depends on the JavaScript engine.

```
> '-A2B-'.toLowerCase()
'-a2b-'
> 'ABΓ'.toLowerCase()
'αβγ'
```

- `.trim(): string` <sup>[ES5]</sup>

Returns a copy of the string in which all leading and trailing whitespace is gone.

```
> '\r\n# \t'.trim()
'#'
```

### 17.7.8 String.prototype: chars, char codes, code points

- `.charAt(pos: number): string` <sup>[ES1]</sup>

<sup>6</sup>[http://exploringjs.com/es2018-es2019/ch\\_regexp-named-capture-groups.html](http://exploringjs.com/es2018-es2019/ch_regexp-named-capture-groups.html)



Returns the character at index `pos`, as a string (JavaScript does not have a datatype for characters). `str[i]` is equivalent to `str.charAt(i)` and more concise (caveat: may not work on old engines).

```
> 'abc'.charAt(1)
'b'
```

- `.charCodeAt(pos: number): number` <sup>[ES1]</sup>

Returns the 16-bit number (0–65535) of the UTF-16 code unit (character) at index `pos`.

```
> 'abc'.charCodeAt(1)
98
```

- `.codePointAt(pos: number): number | undefined` <sup>[ES6]</sup>

Returns the 21-bit number of the Unicode code point of the 1–2 characters at index `pos`. If there is no such index, it returns `undefined`.

### 17.7.9 Sources

- TypeScript’s built-in typings<sup>7</sup>
- MDN web docs for JavaScript<sup>8</sup>
- ECMAScript language specification<sup>9</sup>



#### Exercise: Using string methods

`exercises/strings/remove_extension_test.js`



#### Quiz

See quiz app.

<sup>7</sup><https://github.com/Microsoft/TypeScript/blob/master/lib/>

<sup>8</sup><https://developer.mozilla.org/en-US/docs/Web/JavaScript>

<sup>9</sup><https://tc39.github.io/ecma262/>



## Chapter 18

# Using template literals and tagged templates

### Contents

---

<b>18.1 Disambiguation: “template”</b>	<b>139</b>
<b>18.2 Template literals</b>	<b>140</b>
<b>18.3 Tagged templates</b>	<b>140</b>
18.3.1 Tag function library: lit-html	141
18.3.2 Tag function library: re-template-tag	142
18.3.3 Tag function library: graphql-tag	142
<b>18.4 Raw string literals</b>	<b>142</b>
<b>18.5 (Advanced)</b>	<b>143</b>
<b>18.6 Multi-line template literals and indentation</b>	<b>143</b>
18.6.1 Fix: template tag for dedenting	144
18.6.2 Fix: .trim()	144
<b>18.7 Simple templating via template literals</b>	<b>145</b>
18.7.1 A more complex example	145
18.7.2 Simple HTML-escaping	146
<b>18.8 Further reading</b>	<b>146</b>

---

Before we dig into the two features *template literal* and *tagged template*, let’s first examine the multiple meanings of the term *template*.

### 18.1 Disambiguation: “template”

The following three things are significantly different, despite all having *template* in their names and despite all of them looking similar:

- A *web template* is a function from data to text. It is frequently used in web development and often defined via text files. For example, the following text defines a template for the library Handlebars<sup>1</sup>:

---

<sup>1</sup><https://handlebarsjs.com>

```
<div class="entry">
  <h1>{{title}}</h1>
  <div class="body">
    {{body}}
  </div>
</div>
```

- A *template literal* is a string literal with more features. For example, interpolation. It is delimited by backticks:

```
const num = 5;
assert.equal(`Count: ${num}!`, 'Count: 5!');
```

- A *tagged template* is a function followed by a template literal. It results in that function being called and the contents of the template literal being fed into it as parameters.

```
const getArgs = (...args) => args;
assert.deepEqual(
  getArgs`Count: ${5}!`,
  [['Count: ', '!'], 5] );
```

Note that `getArgs()` receives both the text of the literal and the data interpolated via `${}`.

## 18.2 Template literals

Template literals have two main benefits, compared to normal string literals.

First, they support *string interpolation*: you can insert expressions if you put them inside `${}` and ```:

```
const MAX = 100;
function doSomeWork(x) {
  if (x > MAX) {
    throw new Error(`At most ${MAX} allowed: ${x}!`);
  }
  // ...
}
assert.throws(
  () => doSomeWork(101),
  {message: 'At most 100 allowed: 101!'});
```

Second, template literals can span multiple lines:

```
const str = `this is
a text with
multiple lines`;
```

Template literals always produce strings.

## 18.3 Tagged templates

The expression in line A is a *tagged template*:

```
const first = 'Lisa';
const last = 'Simpson';

const result = tagFunction`Hello ${first} ${last}!`; // A
```

The last line is equivalent to:

```
const result = tagFunction(['Hello ', ' ', '!'], first, last);
```

The parameters of `tagFunction` are:

- Template strings (first parameter): an Array with the text fragments surrounding the interpolations (`${...}`).
  - In the example: `['Hello ', ' ', '!']`
- Substitutions (remaining parameters): the interpolated values.
  - In the example: `'Lisa'` and `'Simpson'`

The static (fixed) parts of the literal (the template strings) are separated from the dynamic parts (the substitutions).

`tagFunction` can return arbitrary values and gets two views of the template strings as input (only the cooked view is shown in the previous example):

- A *cooked view* where, e.g.:
  - `\t` becomes a tab
  - `\\` becomes a single backslash
- A *raw view* where, e.g.:
  - `\t` becomes a slash followed by a `t`
  - `\\` becomes two backslashes

The raw view enables raw string literals via `String.raw` (described later) and similar applications.

Tagged templates are great for supporting small embedded languages (so-called *domain-specific languages*). We'll continue with a few examples.

### 18.3.1 Tag function library: lit-html

`lit-html`<sup>2</sup> is a templating library that is based on tagged templates and used by the frontend framework `Polymer`<sup>3</sup>:

```
import {html, render} from 'lit-html';

const template = (items) => html`
  <ul>
    ${
      repeat(items,
        (item) => item.id,
        (item, index) => html`<li>${index}. ${item.name}</li>`
      )
    }
  </ul>`
```

---

<sup>2</sup><https://github.com/Polymer/lit-html>

<sup>3</sup><https://www.polymer-project.org/>

```

    </ul>
`;

```

`repeat()` is a custom function for looping. Its 2nd parameter produces unique keys for the values returned by the 3rd parameter. Note the nested tagged template used by that parameter.

### 18.3.2 Tag function library: `re-template-tag`

`re-template-tag` is a simple library for composing regular expressions. Templates tagged with `re` produce regular expressions. The main benefit is that you can interpolate regular expressions and plain text via `${}` (see `RE_DATE`):

```

import {re} from 're-template-tag';

const RE_YEAR = re`(?<year>[0-9]{4})`;
const RE_MONTH = re`(?<month>[0-9]{2})`;
const RE_DAY = re`(?<day>[0-9]{2})`;
const RE_DATE = re`/${RE_YEAR}-${RE_MONTH}-${RE_DAY}/u`;

const match = RE_DATE.exec('2017-01-27');
assert.equal(match.groups.year, '2017');

```

### 18.3.3 Tag function library: `graphql-tag`

The library `graphql-tag`<sup>4</sup> lets you create GraphQL queries via tagged templates:

```

import gql from 'graphql-tag';

const query = gql`
  {
    user(id: 5) {
      firstName
      lastName
    }
  }
`;

```

Additionally, there are plugins for pre-compiling such queries in Babel, TypeScript, etc.

## 18.4 Raw string literals

Raw string literals are implemented via the tag function `String.raw`. They are a string literal where backslashes don't do anything special (such as escaping characters etc.):

```

assert.equal(String.raw`back`, '\\back');

```

One example where that helps is strings with regular expressions:

---

<sup>4</sup><https://github.com/apollographql/graphql-tag>

```
const regex1 = /^\. /;
const regex2 = new RegExp('^\\. ');
const regex3 = new RegExp(String.raw`^\. `);
```

All three regular expressions are equivalent. You can see that with a string literal, you have to write the backslash twice to escape it for that literal. With a raw string literal, you don't have to do that.

Another example where raw string literal are useful is Windows paths:

```
const WIN_PATH = String.raw`C:\foo\bar`;
assert.equal(WIN_PATH, 'C:\\foo\\bar');
```

## 18.5 (Advanced)

All remaining sections are advanced

## 18.6 Multi-line template literals and indentation

If you put multi-line text in template literals, two goals are in conflict: On one hand, the text should be indented to fit inside the source code. On the other hand, its lines should start in the leftmost column.

For example:

```
function div(text) {
  return `
    <div>
      ${text}
    </div>
  `;
}
console.log('Output:');
console.log(div('Hello!'))
// Replace spaces with mid-dots:
.replace(/ /g, '·')
// Replace \n with #\n:
.replace(/\n/g, '#\n');
```

Due to the indentation, the template literal fits well into the source code. Alas, the output is also indented. And we don't want the return at the beginning and the return plus two spaces at the end.

Output:

```
#
····<div>#
·····Hello!#
····</div>#
..
```

There are two ways to fix this: via a tagged template or by trimming the result of the template literal.

### 18.6.1 Fix: template tag for dedenting

The first fix is to use a custom template tag that removes the unwanted whitespace. It uses the first line after the initial line break to determine in which column the text starts and cuts off the indents everywhere. It also removes the line break at the very beginning and the indentation at the very end. One such template tag is `dedent` by Desmond Brand<sup>5</sup>:

```
import dedent from 'dedent';
function divDedented(text) {
  return dedent`
    <div>
      ${text}
    </div>
  `;
}
console.log('Output:');
console.log(divDedented('Hello!'));
```

This time, the output is not indented:

Output:  
`<div>`  
Hello!  
`</div>`

### 18.6.2 Fix: `.trim()`

The second fix is quicker, but also dirtier:

```
function divDedented(text) {
  return `
    <div>
      ${text}
    </div>
  `.trim();
}
console.log('Output:');
console.log(divDedented('Hello!'));
```

The string method `.trim()` removes the superfluous whitespace at the beginning and at the end, but the content itself must start in the leftmost column. The advantage of this solution is not needing a custom tag function. The downside is that it looks ugly.

The output looks like it did with `dedent` (however, there is no line break at the end):

Output:  
`<div>`  
Hello!  
`</div>`

---

<sup>5</sup><https://github.com/dmnd/dedent>



## 18.7 Simple templating via template literals

While template literals look like web templates, it is not immediately obvious how to use them for (web) templating: A web template gets its data from an object, while a template literal gets its data from variables. The solution is to use a template literal in the body of a function whose parameter receives the templating data. For example:

```
const tmpl = (data) => `Hello ${data.name}!`;
assert.equal(tmpl({name: 'Jane'}), 'Hello Jane!');
```

### 18.7.1 A more complex example

As a more complex example, we'd like to take an Array of addresses and produce an HTML table. This is the Array:

```
const addresses = [
  { first: '<Jane>', last: 'Bond' },
  { first: 'Lars', last: '<Croft>' },
];
```

The function `tmpl()` that produces the HTML table looks as follows.

```
1  const tmpl = (addrs) => `
2  <table>
3    ${addrs.map(
4      (addr) => `
5        <tr>
6          <td>${escapeHtml(addr.first)}</td>
7          <td>${escapeHtml(addr.last)}</td>
8        </tr>
9        `
10     ).join('')}
11  </table>
12  `
    .trim();
```

`tmpl()` takes the following steps:

- The text inside the `<table>` is produced via a nested templating function for single addresses (line 4). Note how it uses the string method `.trim()` at the end, to remove unnecessary whitespace.
- The nested templating function is applied to each element of the Array `addrs` via the Array method `.map()` (line 3).
- The resulting Array (of strings) is converted into a string via the Array method `.join()` (line 10).
- The helper function `escapeHtml()` is used to escape special HTML characters (line 6 and line 7). Its implementation is shown in the next section.

This is how to call `tmpl()` with the addresses and log the result:

```
console.log(tmpl(addresses));
```

The output is:

```
<table>
  <tr>
```

```
    <td>&lt;Jane&gt;</td>
    <td>Bond</td>
  </tr><tr>
    <td>Lars</td>
    <td>&lt;Croft&gt;</td>
  </tr>
</table>
```

### 18.7.2 Simple HTML-escaping

```
function escapeHtml(str) {
  return str
    .replace(/&/g, '&amp;') // first!
    .replace(/>/g, '&gt;')
    .replace(/</g, '&lt;')
    .replace(/"/g, '&quot;')
    .replace(/'/g, '&#39;')
    .replace(/`/g, '&#96;')
  ;
}
```



#### Exercise: HTML templating

Exercise with bonus challenge: `exercises/template-literals/templating_test.js`

## 18.8 Further reading

- How to implement your own tag functions is described in “Exploring ES6<sup>6</sup>”.



#### Quiz

See quiz app.

---

<sup>6</sup>[http://exploringjs.com/es6/ch\\_template-literals.html](http://exploringjs.com/es6/ch_template-literals.html)

# Chapter 19

## Symbols

### Contents

<b>19.1 Use cases for symbols</b>	<b>147</b>
19.1.1 Symbols: enum-style values	148
19.1.2 Symbols: unique property keys	149
<b>19.2 Publicly known symbols</b>	<b>149</b>
<b>19.3 Converting symbols</b>	<b>150</b>
<b>19.4 Further reading</b>	<b>151</b>

Symbols are primitive values that are created via the factory function `Symbol()`:

```
const mySymbol = Symbol('mySymbol');
```

The parameter is optional and provides a description, which is mainly useful for debugging.

On one hand, symbols are like objects in that each value created by `Symbol()` is unique and not compared by value:

```
> Symbol() === Symbol()
false
```

On the other hand, they also behave like primitive values – they have to be categorized via `typeof` and they can be property keys in objects:

```
const sym = Symbol();
assert.equal(typeof sym, 'symbol');

const obj = {
  [sym]: 123,
};
```

### 19.1 Use cases for symbols

The main use cases for symbols are:

- Defining constants for the values of an enumerated type.
- Creating unique property keys.

### 19.1.1 Symbols: enum-style values

Let's assume you want to create constants representing the colors red, orange, yellow, green, blue and violet. One simple way of doing so would be to use strings:

```
const COLOR_BLUE = 'Blue';
```

On the plus side, logging that constant produces helpful output. On the minus side, there is a risk of mistaking an unrelated value for a color, because two strings with the same content are considered equal:

```
const MOOD_BLUE = 'Blue';
assert.equal(COLOR_BLUE, MOOD_BLUE);
```

We can fix that problem via a symbol:

```
const COLOR_BLUE = Symbol('Blue');

const MOOD_BLUE = 'Blue';
assert.notEqual(COLOR_BLUE, MOOD_BLUE);
```

Let's use symbol-valued constants to implement a function:

```
const COLOR_RED    = Symbol('Red');
const COLOR_ORANGE = Symbol('Orange');
const COLOR_YELLOW = Symbol('Yellow');
const COLOR_GREEN  = Symbol('Green');
const COLOR_BLUE   = Symbol('Blue');
const COLOR_VIOLET = Symbol('Violet');

function getComplement(color) {
  switch (color) {
    case COLOR_RED:
      return COLOR_GREEN;
    case COLOR_ORANGE:
      return COLOR_BLUE;
    case COLOR_YELLOW:
      return COLOR_VIOLET;
    case COLOR_GREEN:
      return COLOR_RED;
    case COLOR_BLUE:
      return COLOR_ORANGE;
    case COLOR_VIOLET:
      return COLOR_YELLOW;
    default:
      throw new Exception('Unknown color: '+color);
  }
}

assert.equal(getComplement(COLOR_YELLOW), COLOR_VIOLET);
```

Notably, this function throws an exception if you call it with 'Blue':

```
assert.throws(() => getComplement('Blue'));
```

### 19.1.2 Symbols: unique property keys

The keys of properties (fields) in objects are used at two levels:

- The program operates at a base level. Its keys reflect the problem domain.
- Libraries and ECMAScript operate at a meta-level. For example, `.toString` is a meta-level property key:

The following code demonstrates the difference:

```
const point = {
  x: 7,
  y: 4,
  toString() {
    return `${this.x}, ${this.y}`;
  },
};
assert.equal(String(point), '(7, 4)');
```

Properties `.x` and `.y` exist at the base level. They are the coordinates of the point encoded by `point` and reflect the problem domain. Method `.toString()` is a meta-level property. It tells JavaScript how to stringify this object.

The meta-level and the base level must never clash, which becomes harder when introducing new mechanisms later in the life of a programming language.

Symbols can be used as property keys and solve this problem: Each symbol is unique and never clashes with any string or any other symbol.

As an example, let's assume we are writing a library that treats objects differently if they implement a special method. This is what defining a property key for such a method and implementing it for an object would look like:

```
const specialMethod = Symbol('specialMethod');
const obj = {
  [specialMethod](x) {
    return x + x;
  }
};
assert.equal(obj[specialMethod]('abc'), 'abcabc');
```

This syntax is explained in more detail in [the chapter on objects](#).

## 19.2 Publicly known symbols

Symbols that play special roles within ECMAScript are called *publicly known symbols*. Examples include:

- `Symbol.iterator`: makes an object *iterable*. It's the key of a method that returns an iterator. Iteration is explained in [its own chapter](#).
- `Symbol.hasInstance`: customizes how `instanceof` works. It's the key of a method to be implemented by the right-hand side of that operator. For example:

```
class PrimitiveNull {
  static [Symbol.hasInstance](x) {
    return x === null;
  }
}
assert.equal(null instanceof PrimitiveNull, true);
```

- `Symbol.toStringTag`: influences the default `.toString()` method.

```
> String({})
'[object Object]'
> String({ [Symbol.toStringTag]: 'is no money' })
'[object is no money]'
```

Note: It's usually better to override `.toString()`.



#### Exercises: Publicly known symbols

- `Symbol.toStringTag`: `exercises/symbols/to_string_tag_test.js`
- `Symbol.hasInstance`: `exercises/symbols/has_instance_test.js`

## 19.3 Converting symbols

What happens if we convert a symbol `sym` to another primitive type? Tbl. 19.1 has the answers.

Table 19.1: The results of converting symbols to other primitive types.

Convert to	Explicit conversion	Coercion (implicit conv.)
boolean	<code>Boolean(sym) → OK</code>	<code>!sym → OK</code>
number	<code>Number(sym) → TypeError</code>	<code>sym*2 → TypeError</code>
string	<code>String(sym) → OK</code> <code>sym.toString() → OK</code>	<code>''+sym → TypeError</code> <code>`\${sym}` → TypeError</code>

One key pitfall with symbols is how often exceptions are thrown when converting them to something else. What is the thinking behind that? First, conversion to number never makes sense and should be warned about. Second, converting a symbol to a string is indeed useful for diagnostic output. But it also makes sense to warn about accidentally turning a symbol into a string property key:

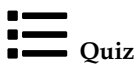
```
const obj = {};
const sym = Symbol();
assert.throws(
  () => { obj['__'+sym+'__'] = true },
  { message: 'Cannot convert a Symbol value to a string' });
```

The downside is that the exceptions make working with symbols more complicated. You have to explicitly convert symbols when assembling strings via the plus operator:

```
> const mySymbol = Symbol('mySymbol');  
> 'Symbol I used: ' + mySymbol  
TypeError: Cannot convert a Symbol value to a string  
> 'Symbol I used: ' + String(mySymbol)  
'Symbol I used: Symbol(mySymbol)'
```

## 19.4 Further reading

- In-depth coverage of symbols (cross-realm symbols, etc.): see “Exploring ES6<sup>1</sup>”



See quiz app.

---

<sup>1</sup>[http://exploringjs.com/es6/ch\\_symbols.html](http://exploringjs.com/es6/ch_symbols.html)





## **Part V**

# **Control flow and data flow**



# Chapter 20

## Control flow statements

### Contents

<b>20.1 Controlling loops: break and continue</b>	<b>156</b>
20.1.1 break	156
20.1.2 Additional use case for break: leaving blocks	156
20.1.3 continue	157
<b>20.2 if statements</b>	<b>157</b>
20.2.1 The syntax of if statements	158
<b>20.3 switch statements</b>	<b>158</b>
20.3.1 A first example	159
20.3.2 Don't forget to return or break!	159
20.3.3 Empty cases clauses	160
20.3.4 Checking for illegal values via a default clause	161
<b>20.4 while loops</b>	<b>162</b>
20.4.1 Examples	162
<b>20.5 do-while loops</b>	<b>162</b>
<b>20.6 for loops</b>	<b>162</b>
20.6.1 Examples	163
<b>20.7 for-of loops</b>	<b>164</b>
20.7.1 const: for-of vs. for	164
20.7.2 Iterating over iterables	164
20.7.3 Iterating over [index, element] pairs of Arrays	164
<b>20.8 for-await-of loops</b>	<b>165</b>
<b>20.9 for-in loops (avoid)</b>	<b>165</b>

This chapter covers the following control flow statements:

- if statements (ES1)
- switch statements (ES3)
- while loops (ES1)
- do-while loops (ES3)
- for loops (ES1)

- for-of loops (ES6)
- for-await-of loops (ES2018)
- for-in loops (ES1)

Before we get to the actual control flow statements, let's take a look at two operators for controlling loops.

## 20.1 Controlling loops: break and continue

The two operators break and continue can be used to control loops and other statements while you are inside them.

### 20.1.1 break

There are two versions of break: one with an operand and one without an operand. The latter version works inside the following statements: while, do-while, for, for-of, for-await-of, for-in and switch. It immediately leaves the current statement:

```
for (const x of ['a', 'b', 'c']) {
  console.log(x);
  if (x === 'b') break;
  console.log('---')
}
```

```
// Output:
// 'a'
// '---'
// 'b'
```

### 20.1.2 Additional use case for break: leaving blocks

break with an operand works everywhere. Its operand is a *label*. Labels can be put in front of any statement, including blocks. break foo leaves the statement whose label is foo:

```
foo: { // label
  if (condition) break foo; // labeled break
  // ...
}
```

Breaking from blocks is occasionally handy if you are using a loop and want to distinguish between finding what you were looking for and finishing the loop without success:

```
function search(stringArray, suffix) {
  let result;
  search_block: {
    for (const str of stringArray) {
      if (str.endsWith(suffix)) {
        // Success
        result = str;
      }
    }
  }
}
```

```

        break search_block;
    }
} // for
// Failure
result = '(Untitled)';
} // search_block

return { suffix, result };
// same as: {suffix: suffix, result: result}
}
assert.deepEqual(
  search(['foo.txt', 'bar.html'], '.html'),
  { suffix: '.html', result: 'bar.html' }
);
assert.deepEqual(
  search(['foo.txt', 'bar.html'], '.js'),
  { suffix: '.js', result: '(Untitled)' }
);

```

### 20.1.3 continue

continue only works inside while, do-while, for, for-of, for-await-of and for-in. It immediately leaves the current loop iteration and continues with the next one. For example:

```

const lines = [
  'Normal line',
  '# Comment',
  'Another normal line',
];
for (const line of lines) {
  if (line.startsWith('#')) continue;
  console.log(line);
}
// Output:
// 'Normal line'
// 'Another normal line'

```

## 20.2 if statements

These are two simple if statements: One with just a “then” branch and one with both a “then” branch and an “else” branch:

```

if (cond) {
  // then branch
}

if (cond) {
  // then branch
}

```

```
} else {  
    // else branch  
}
```

Instead of the block, `else` can also be followed by another `if` statement:

```
if (cond1) {  
    // ...  
} else if (cond2) {  
    // ...  
}
```

```
if (cond1) {  
    // ...  
} else if (cond2) {  
    // ...  
} else {  
    // ...  
}
```

You can continue this chain with more `else if`s.

### 20.2.1 The syntax of `if` statements

The general syntax of `if` statements is:

```
if (cond) «then_statement»  
else «else_statement»
```

So far, the `then_statement` has always been a block, but you can also use a statement. That statement must be terminated with a semicolon:

```
if (true) console.log('Yes'); else console.log('No');
```

That means that `else if` is not its own construct, it's simply an `if` statement whose `else_statement` is another `if` statement.

## 20.3 switch statements

The head of a `switch` statement looks as follows:

```
switch («switch_expression») {  
    «switch_body»  
}
```

Inside the body of `switch`, there are zero or more case clauses:

```
case «case_expression»:  
    «statements»
```

And, optionally, a default clause:

```
default:
  «statements»
```

A switch is executed as follows:

- Evaluate the switch expression.
- Jump to the first case clause whose expression has the same result as the switch expression.
- If there is no such case clause, jump to the default clause.
- If there is no default clause, nothing happens.

### 20.3.1 A first example

Let's look at an example: The following function converts a number from 1–7 to the name of a weekday.

```
function dayOfTheWeek(num) {
  switch (num) {
    case 1:
      return 'Monday';
    case 2:
      return 'Tuesday';
    case 3:
      return 'Wednesday';
    case 4:
      return 'Thursday';
    case 5:
      return 'Friday';
    case 6:
      return 'Saturday';
    case 7:
      return 'Sunday';
  }
}
assert.equal(dayOfTheWeek(5), 'Friday');
```

### 20.3.2 Don't forget to return or break!

At the end of a case clause, execution continues with the next case clause (unless you return or break). For example:

```
function dayOfTheWeek(num) {
  let name;
  switch (num) {
    case 1:
      name = 'Monday';
    case 2:
      name = 'Tuesday';
    case 3:
      name = 'Wednesday';
    case 4:
```

```

        name = 'Thursday';
    case 5:
        name = 'Friday';
    case 6:
        name = 'Saturday';
    case 7:
        name = 'Sunday';
    }
    return name;
}
assert.equal(dayOfTheWeek(5), 'Sunday'); // not 'Friday'!
```

That is, the previous implementation of `dayOfTheWeek()` only worked, because we used `return`. We can fix this implementation by using `break`:

```

function dayOfTheWeek(num) {
    let name;
    switch (num) {
        case 1:
            name = 'Monday';
            break;
        case 2:
            name = 'Tuesday';
            break;
        case 3:
            name = 'Wednesday';
            break;
        case 4:
            name = 'Thursday';
            break;
        case 5:
            name = 'Friday';
            break;
        case 6:
            name = 'Saturday';
            break;
        case 7:
            name = 'Sunday';
            break;
    }
    return name;
}
assert.equal(dayOfTheWeek(5), 'Friday');
```

### 20.3.3 Empty cases clauses

The statements of a case clause can be omitted, which effectively gives us multiple case expressions per case clause:



```
function isWeekDay(name) {
  switch (name) {
    case 'Monday':
    case 'Tuesday':
    case 'Wednesday':
    case 'Thursday':
    case 'Friday':
      return true;
    case 'Saturday':
    case 'Sunday':
      return false;
  }
}
assert.equal(isWeekDay('Wednesday'), true);
assert.equal(isWeekDay('Sunday'), false);
```

### 20.3.4 Checking for illegal values via a default clause

A default clause is jumped to if the switch expression has no other match. That makes it useful for error checking:

```
function isWeekDay(name) {
  switch (name) {
    case 'Monday':
    case 'Tuesday':
    case 'Wednesday':
    case 'Thursday':
    case 'Friday':
      return true;
    case 'Saturday':
    case 'Sunday':
      return false;
    default:
      throw new Error('Illegal value: '+name);
  }
}
assert.throws(
  () => isWeekDay('January'),
  {message: 'Illegal value: January'});
```



#### Exercises: switch

- `exercises/control-flow/month_to_number_test.js`
- Bonus: `exercises/control-flow/is_object_via_switch_test.js`

## 20.4 while loops

A while loop has the following syntax:

```
while («condition») {  
  «statements»  
}
```

Before each loop iteration, while evaluates condition:

- If the result is falsy, the loop is finished.
- If the result is truthy, the while body is executed one more time.

### 20.4.1 Examples

The following code uses a while loop. In each loop iteration, it removes the first element of `arr` via `.shift()` and logs it.

```
const arr = ['a', 'b', 'c'];  
while (arr.length > 0) {  
  const elem = arr.shift(); // remove first element  
  console.log(elem);  
}  
// Output:  
// 'a'  
// 'b'  
// 'c'
```

If the condition is `true` then while is an infinite loop:

```
while (true) {  
  if (Math.random() === 0) break;  
}
```

## 20.5 do-while loops

The do-while loop works much like while, but it checks its condition *after* each loop iteration (not before).

```
let input;  
do {  
  input = prompt('Enter text:');  
} while (input !== 'q');
```

## 20.6 for loops

With a for loop, you use the head to control how its body is executed. The head has three parts and each of them is optional:

```
for («initialization»; «condition»; «post_iteration») {  
  «statements»  
}
```

- **initialization**: sets up variables etc. for the loop. Variables declared here via `let` or `const` only exist inside the loop.
- **condition**: This condition is checked before each loop iteration. If it is falsy, the loop stops.
- **post\_iteration**: This code is executed after each loop iteration.

A for loop is therefore roughly equivalent to the following while loop:

```
«initialization»  
while («condition») {  
  «statements»  
  «post_iteration»  
}
```

### 20.6.1 Examples

As an example, this is how to count from zero to two via a for loop:

```
for (let i=0; i<3; i++) {  
  console.log(i);  
}
```

```
// Output:  
// 0  
// 1  
// 2
```

This is how to log the contents of an Array via a for loop:

```
const arr = ['a', 'b', 'c'];  
for (let i=0; i<3; i++) {  
  console.log(arr[i]);  
}
```

```
// Output:  
// 'a'  
// 'b'  
// 'c'
```

If you omit all three parts of the head, you get an infinite loop:

```
for (;;) {  
  if (Math.random() === 0) break;  
}
```

## 20.7 for-of loops

A for-of loop iterates over an *iterable* – a data container that supports [the iteration protocol](#). Each iterated value is stored in a variable, as specified in the head:

```
for («iteration_variable» of «iterable») {  
  «statements»  
}
```

The iteration variable is usually created via a variable declaration:

```
const iterable = ['hello', 'world'];  
for (const elem of iterable) {  
  console.log(elem);  
}  
// Output:  
// 'hello'  
// 'world'
```

But you can also use a (mutable) variable that already exists:

```
const iterable = ['hello', 'world'];  
let elem;  
for (elem of iterable) {  
  console.log(elem);  
}
```

### 20.7.1 const: for-of vs. for

Note that, in for-of loops, you can use `const`. The iteration variable can still be different for each iteration (it just can't change during the iteration). Think of it as a new `const` declaration being executed each time, in a fresh scope.

In contrast, in for loops, you must declare variables via `let` or `var` if their values change.

### 20.7.2 Iterating over iterables

As mentioned before, for-of works with any iterable object, not just with Arrays. For example, with Sets:

```
const set = new Set(['hello', 'world']);  
for (const elem of set) {  
  console.log(elem);  
}
```

### 20.7.3 Iterating over [index, element] pairs of Arrays

Lastly, you can also use for-of to iterate over the [index, element] entries of Arrays:

```
const arr = ['a', 'b', 'c'];
for (const [index, elem] of arr.entries()) {
  console.log(`${index} -> ${elem}`);
}
// Output:
// '0 -> a'
// '1 -> b'
// '2 -> c'
```



### Exercise: for-of

exercises/control-flow/array\_to\_string\_test.js

## 20.8 for-await-of loops

for-await-of is like for-of, but it works with asynchronous iterables instead of synchronous ones. And it can only be used inside async functions and async generators.

```
for await (const item of asyncIterable) {
  // ...
}
```

for-await-of is described in detail in a later chapter.

## 20.9 for-in loops (avoid)

for-in has several pitfalls. Therefore, it is usually best to avoid it.

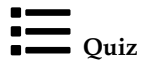
This is an example of using for-in:

```
function getOwnPropertyNames(obj) {
  const result = [];
  for (const key in obj) {
    if ({}.hasOwnProperty.call(obj, key)) {
      result.push(key);
    }
  }
  return result;
}
assert.deepEqual(
  getOwnPropertyNames({ a: 1, b: 2 }),
  ['a', 'b']);
assert.deepEqual(
  getOwnPropertyNames(['a', 'b']),
  ['0', '1']); // strings!
```

This is a better alternative:

```
function getOwnPropertyNames(obj) {  
  const result = [];  
  for (const key of Object.keys(obj)) {  
    result.push(key);  
  }  
  return result;  
}
```

For more information on `for-in`, consult “Speaking JavaScript<sup>1</sup>”.



Quiz

See quiz app.

---

<sup>1</sup><http://speakingjs.com/es5/ch13.html#for-in>

# Chapter 21

## Exception handling

### Contents

<b>21.1 Motivation: throwing and catching exceptions</b>	<b>167</b>
<b>21.2 throw</b>	<b>168</b>
21.2.1 Options for creating error objects	168
<b>21.3 try-catch-finally</b>	<b>169</b>
21.3.1 The catch clause	169
21.3.2 The finally clause	170
<b>21.4 Error classes and their properties</b>	<b>170</b>
21.4.1 Properties of error classes	171

This chapter covers how JavaScript handles exceptions.

As an aside: JavaScript didn't support exceptions until ES3. That explains why they are used sparingly by the language and its standard library.

### 21.1 Motivation: throwing and catching exceptions

Consider the following code. It reads profiles stored in files into an Array with instances of class Profile:

```
function readProfiles(filePaths) {
  const profiles = [];
  for (const filePath of filePaths) {
    try {
      const profile = readOneProfile(filePath);
      profiles.push(profile);
    } catch (err) { // (A)
      console.log('Error in: ' + filePath, err);
    }
  }
}

function readOneProfile(filePath) {
```

```

const profile = new Profile();
const file = openFile(filePath);
// ... (Read the data in `file` into `profile`)
return profile;
}
function openFile(filePath) {
  if (!fs.existsSync(filePath)) {
    throw new Error('Could not find file '+filePath); // (B)
  }
  // ... (Open the file whose path is `filePath`)
}

```

Let's examine what happens in line B: An error occurred, but the best place to handle the problem is not the current location, it's line A. There, we can skip the current file and move on to the next one.

Therefore:

- In line B, we use a throw statement to indicate that there was a problem.
- In line A, we use a try-catch statement to handle the problem.

When we throw, the following constructs are active:

```

readProfiles(...)
  for (const filePath of filePaths)
    try
      readOneProfile(...)
      openFile(...)
      if (!fs.existsSync(filePath))
        throw

```

throw walks up this chain of constructs, until it finds a try statement. Execution continues in the catch clause of that try statement.

## 21.2 throw

```
throw «value»;
```

Any value can be thrown, but it's best to throw instances of Error:

```
throw new Error('Problem!');
```

### 21.2.1 Options for creating error objects

- Use class Error. That is less limiting in JavaScript than in a more static language, because you can add your own properties to instances:

```

const err = new Error('Could not find the file');
err.filePath = filePath;
throw err;

```

- Use one of JavaScript's subclasses of Error (which are listed [later](#)).



- Subclass Error yourself.

```
class MyError extends Error {
}
function func() {
  throw new MyError;
}
assert.throws(
  () => func(),
  MyError);
```

## 21.3 try-catch-finally

The maximal version of the try statement looks as follows:

```
try {
  // try_statements
} catch (error) {
  // catch_statements
} finally {
  // finally_statements
}
```

The try clause is mandatory, but you can omit either catch or finally (but not both).

### 21.3.1 The catch clause

If an exception is thrown in the try block (and not caught earlier) then it is assigned to the parameter of the catch clause and the code in that clause is executed. Unless it is directed elsewhere (via return or similar), execution continues after the catch clause: with the finally clause – if it exists – or after the try statement.

The following code demonstrates that the value that is thrown in line A is indeed caught in line B.

```
const errorObject = new Error();
function func() {
  throw errorObject; // (A)
}

try {
  func();
} catch (err) { // (B)
  assert.equal(err, errorObject);
}
```

### 21.3.2 The finally clause

Let's look at a common use case for `finally`: You have created a resource and want to always destroy it when you are done with it – no matter what happens while working with it. You'd implement that as follows:

```
const resource = createResource();
try {
  // Work with `resource`: errors may be thrown.
} finally {
  resource.destroy();
}
```

The `finally` is always executed – even if an error is thrown (line A):

```
let finallyWasExecuted = false;
assert.throws(
  () => {
    try {
      throw new Error(); // (A)
    } finally {
      finallyWasExecuted = true;
    }
  },
  Error
);
assert.equal(finallyWasExecuted, true);
```

The `finally` is always executed – even if there is a `return` statement (line A):

```
let finallyWasExecuted = false;
function func() {
  try {
    return; // (A)
  } finally {
    finallyWasExecuted = true;
  }
}
func();
assert.equal(finallyWasExecuted, true);
```

## 21.4 Error classes and their properties

Quoting the ECMAScript specification<sup>1</sup>:

- Error [root class]
  - RangeError: Indicates a value that is not in the set or range of allowable values.
  - ReferenceError: Indicate that an invalid reference value has been detected.
  - SyntaxError: Indicates that a parsing error has occurred.

---

<sup>1</sup><https://tc39.github.io/ecma262/#sec-native-error-types-used-in-this-standard>

- `TypeError`: is used to indicate an unsuccessful operation when none of the other *NativeError* objects are an appropriate indication of the failure cause.
- `URIError`: Indicates that one of the global URI handling functions was used in a way that is incompatible with its definition.

### 21.4.1 Properties of error classes

Consider `err`, an instance of `Error`:

```
const err = new Error('Hello!');
assert.equal(String(err), 'Error: Hello!');
```

Two properties of `err` are especially useful:

- `.message`: contains just the error message.  
`assert.equal(err.message, 'Hello!');`
- `.stack`: contains a stack trace. It is supported by all mainstream browsers.

```
console.log(err.stack);
// Error
//   at repl:1:11
//   at ContextifyScript.Script.runInThisContext (vm.js:44:33)
//   ...
```

Note that logging an error usually also displays a stack trace:

```
console.log(err);
// Error: Hello!
//   at repl:1:11
//   at ContextifyScript.Script.runInThisContext (vm.js:44:33)
//   ...
```



#### Exercise: Exception handling

`exercises/exception-handling/call_function_test.js`



## Chapter 22

# Callable values

### Contents

---

<b>22.1 Kinds of functions</b>	<b>173</b>
22.1.1 Ordinary functions	174
22.1.2 Specialized functions	175
22.1.3 More kinds of real functions and methods	176
<b>22.2 Named function expressions</b>	<b>177</b>
<b>22.3 Arrow functions</b>	<b>178</b>
22.3.1 The syntax of arrow functions	178
22.3.2 Arrow functions: lexical <code>this</code>	179
22.3.3 Syntax pitfall: returning an object literal from an arrow function	179
<b>22.4 Hoisting</b>	<b>180</b>
<b>22.5 Returning values from functions</b>	<b>180</b>
<b>22.6 Parameter handling</b>	<b>181</b>
22.6.1 Terminology: parameters vs. arguments	181
22.6.2 Too many or not enough arguments	181
22.6.3 Parameter default values	182
22.6.4 Rest parameters	182
22.6.5 Named parameters	183
22.6.6 Simulating named parameters	183
22.6.7 Spreading (...) into function calls	184
<b>22.7 Understanding JavaScript's callable values (advanced)</b>	<b>185</b>

---

## 22.1 Kinds of functions

JavaScript has two categories of functions:

- An *ordinary function* can play several roles:
  - Real function (in other languages, you'd simply use the term "function"; in JavaScript, we need to distinguish between the role "real function" and the entity "ordinary function" that can play that role)

- Method
- Constructor function
- A *specialized function* can only play one of those roles. For example:
  - An *arrow function* can only be a real function.
  - A *method* can only be a method.
  - A *class* can only be a constructor function.

The next sections explain what all of those things mean.

### 22.1.1 Ordinary functions

Ordinary functions are created via:

```
// Function declaration (a statement)
function ordinary1(a, b, c) {
  // ...
}

// Anonymous function expression
const ordinary2 = function (a, b, c) {
  // ...
};

// Named function expression
const ordinary3 = function myName(a, b, c) {
  // `myName` is only accessible in here
};
```

Let's examine the parts of a function declaration via an example:

```
function add(x, y) {
  return x + y;
}
```

- `add` is the *name* of the function declaration.
- `add(x, y)` is the *head* of the function declaration.
- `x` and `y` are the *parameters*.
- The curly braces (`{` and `}`) and everything between them are the *body* of the function declaration.
- The return operator explicitly returns a value from the function.

#### 22.1.1.1 Roles played by ordinary functions

Consider the following function declaration from the previous section:

```
function add(x, y) {
  return x + y;
}
```

This function declaration creates an ordinary function whose name is `add`. As an ordinary function, `add()` can play three roles:

- Real function: invoked via a function call. It's what most programming languages consider to be simply *a function*.

```
assert.equal(add(2, 1), 3);
```

- Method: stored in property, invoked via a method call.

```
const obj = { addAsMethod: add };
assert.equal(obj.addAsMethod(2, 4), 6);
```

- Constructor function/class: invoked via new.

```
const inst = new add();
assert.equal(inst instanceof add, true);
```

(As an aside, the names of classes normally start with capital letters.)

### 22.1.2 Specialized functions

Specialized functions are specialized versions of ordinary functions. Each one of them only plays a single role:

- An *arrow function* can only be a real function:

```
const arrow = () => { return 123 };
assert.equal(arrow(), 123);
```

- A *method* can only be a method:

```
const obj = { method() { return 'abc' } };
assert.equal(obj.method(), 'abc');
```

- A *class* can only be a constructor function:

```
class MyClass { /* ... */ }
const inst = new MyClass();
```

Apart from nicer syntax, each kind of specialized function also supports new features, making them better at their job than ordinary functions.

- Arrow functions are explained [later in this chapter](#).
- Methods are explained in [the chapter on single objects](#).
- Classes are explained in [the chapter on prototype chains and classes](#).

Tbl. 22.1 lists the capabilities of ordinary and specialized functions.

Table 22.1: Capabilities of four kinds of functions.

	Ordinary function	Arrow function	Method	Class
Function call	✓	✓	✓	✗
Method call	✓	lexical this	✓	✗
Constructor call	✓	✗	✗	✓

### 22.1.2.1 Specialized functions are still functions

It's important to note that arrow functions, methods and classes are still categorized as functions:

```
> (() => {}) instanceof Function
true
> ({ method() {} }.method) instanceof Function
true
> (class SomeClass {}) instanceof Function
true
```

### 22.1.3 More kinds of real functions and methods



#### Next: a preview of things to come

Warning: You are about to be faced with a long list of things and few explanations. The idea is to give you a brief overview. This chapter focuses on synchronous real functions (which we have already seen). Later chapters will explain everything new we are about to see.

So far, we have always written simple, synchronous code. Upcoming chapters will cover two more modes of programming in JavaScript:

- *Iteration* treats objects as containers of data (so-called *iterables*) and provides a standardized way for retrieving what is inside them.
- *Asynchronous programming* deals with handling a long-running computation. You are notified, when the computation is finished and can do something else in between.

Two variations of real functions and methods help with these modes of programming:

- *Generator functions* and *generator methods* help with iteration.
- *Async functions* and *async methods* help with asynchronous programming.

Both variations can be combined, leading to a total of eight different constructs (tbl. 22.2).

Table 22.2: Different kinds of real functions and methods.

sync vs. async	generator vs. not	real function vs. method
sync		function
sync		method
async		function
async		method
sync	generator	function
sync	generator	method
async	generator	function
async	generator	method

Tbl. 22.3 gives an overview of the syntax for creating these constructs (a Promise is a mechanism for delivering asynchronous results).



Table 22.3: Syntax for creating real functions and methods.

Sync function	Sync method	Result	Values
function f() {} f = function () {} f = () => {}	{ m() {} }	value	1
<b>Sync generator function</b> function* f() {} f = function* () {}	<b>Sync gen. method</b> { * m() {} }	iterable	0+
<b>Async function</b> async function f() {} f = async function () {} f = async () => {}	<b>Async method</b> { async m() {} }	Promise	1
<b>Async generator function</b> async function* f() {} f = async function* () {}	<b>Async gen. method</b> { async * m() {} }	async iterable	0+

For the remainder of this chapter, we'll explore real functions and their foundations.

## 22.2 Named function expressions

Function *declarations* (statements) always have names. With function *expressions*, you can choose whether to provide a name or not.

The following is an example of a function expression without a name – an *anonymous function expression*. Its result is assigned to the variable f1:

```
const f1 = function () {};
```

Next, we see a named function expression (assigned to f2):

```
const f2 = function myName() {};
```

As mentioned before, named function expressions look exactly like function declarations, but they are expressions and thus used in different contexts.

Named function expressions have two benefits.

First, their names show up in stack traces:

```
const func = function problem() { throw new Error() };
setTimeout(func, 0);
// Error
//   at Timeout.problem [as _onTimeout] (repl:1:37)
//   at ontimeout (timers.js:488:11)
//   at tryOnTimeout (timers.js:323:5)
//   at Timer.listOnTimeout (timers.js:283:5)
```

You can see the name `problem` in the first line of the stack trace.

Second, the name of a named function expression provides a convenient way for the function to refer to itself. For example:

```
const fac = function me(n) {
  if (n <= 1) return 1;
  return n * me(n-1);
};
// `me` only exists inside the function:
assert.throws(() => me, ReferenceError);

assert.equal(fac(3), 6);
```

You are free to assign the value of `fac` to another value. It will continue to work, because it refers to itself via its internal name, not via `fac`.

## 22.3 Arrow functions

Arrow functions were added to JavaScript for two reasons:

1. To provide a more concise way for creating functions.
2. To make working with real functions easier: You can't refer to the `this` of the surrounding scope inside an ordinary function (details [soon](#)).

### 22.3.1 The syntax of arrow functions

Let's review the syntax of an anonymous function expression:

```
const f = function (x, y, z) { return 123 };
```

The (roughly) equivalent arrow function looks as follows. Arrow functions are expressions.

```
const f = (x, y, z) => { return 123 };
```

Here, the body of the arrow function is a block. But it can also be an expression. The following arrow function works exactly like the previous one.

```
const f = (x, y, z) => 123;
```

If an arrow function has only a single parameter and that parameter is an identifier (not a destructuring pattern) then you can omit the parentheses around the parameter:

```
const id = x => x;
```

That is convenient when passing arrow functions as parameters to other functions or methods:

```
> [1,2,3].map(x => x+1)
[ 2, 3, 4 ]
```

This last example demonstrates the first benefit of arrow functions – conciseness. In contrast, this is the same method call, but with a function expression:

```
[1,2,3].map(function (x) { return x+1 });
```

### 22.3.2 Arrow functions: lexical `this`

Ordinary functions can be both methods and real functions. Alas, the two roles are in conflict:

- As each ordinary function can be a method, it has its own `this`.
- That own `this` makes it impossible to access the `this` of the surrounding scope from inside an ordinary function. And that is inconvenient for real functions.

The following code demonstrates a common work-around:

```
const prefixer = {
  prefix: '==> ',
  prefixStringArray(stringArray) {
    const that = this; // (A)
    return stringArray.map(
      function (x) {
        return that.prefix + x; // (B)
      }
    );
  },
};
assert.deepEqual(
  prefixer.prefixStringArray(['a', 'b']),
  ['==> a', '==> b']);
```

In line B, we want to access the `this` of `.prefixStringArray()`. But we can't, since the surrounding ordinary function has its own `this` that *shadows* (blocks access to) the `this` of the method. Therefore, we save the method's `this` in the extra variable `that` (line A) and use that variable in line B.

An arrow function doesn't have `this` as an implicit parameter, it picks up its value from the surroundings. That is, `this` behaves just like any other variable.

```
const prefixer = {
  prefix: '==> ',
  prefixStringArray(stringArray) {
    return stringArray.map(
      x => this.prefix + x);
  },
};
```

To summarize:

- In ordinary functions, `this` is an implicit (*dynamic*) parameter (details in [the chapter on single objects](#)).
- Arrow functions get `this` from their surrounding scopes (*lexically*).

### 22.3.3 Syntax pitfall: returning an object literal from an arrow function

If you want the expression body of an arrow function to be an object literal, you must put the literal in parentheses:

```
const func1 = () => ({a: 1});
assert.deepEqual(func1(), { a: 1 });
```

If you don't, JavaScript thinks, the arrow function has a block body (that doesn't return anything):

```
const func2 = () => {a: 1};  
assert.deepEqual(func2(), undefined);
```

{a: 1} is interpreted as a block with the label a: and the expression statement 1.

## 22.4 Hoisting

Function declarations are *hoisted* (internally moved to the top):

```
assert.equal(foo(), 123); // OK
```

```
function foo() { return 123; }
```

Hoisting lets you call a function before it is declared.

Variable declarations are not hoisted – you can only use their variables *after* they were declared:

```
assert.throws( // before  
  () => foo(),  
  ReferenceError);
```

```
const foo = function () { return 123; };
```

```
assert.equal(foo(), 123); // after
```

Class declarations are not hoisted, either:

```
assert.throws(  
  () => new MyClass(),  
  ReferenceError);
```

```
class MyClass {}
```

```
assert.ok(new MyClass() instanceof MyClass);
```

## 22.5 Returning values from functions

You use the return operator to return values from a function:

```
function func() {  
  return 123;  
}  
assert.equal(func(), 123);
```

Another example:

```
function boolToYesNo(bool) {  
  if (bool) {  
    return 'Yes';  
  }
```

```

    } else {
        return 'No';
    }
}
assert.equal(boolToYesNo(true), 'Yes');
assert.equal(boolToYesNo(false), 'No');

```

If, at the end of a function, you haven't returned anything explicitly, JavaScript returns undefined for you:

```

function noReturn() {
    // No explicit return
}
assert.equal(noReturn(), undefined);

```

## 22.6 Parameter handling

### 22.6.1 Terminology: parameters vs. arguments

The term *parameter* and the term *argument* basically mean the same thing. If you want to, you can make the following distinction:

- *Parameters* are part of a function definition. They are also called *formal parameters* and *formal arguments*.
- *Arguments* are part of a function call. They are also called *actual parameters* and *actual arguments*.

### 22.6.2 Too many or not enough arguments

JavaScript does not complain if a function call provides a different number of arguments than expected by the function definition:

- Extra arguments are ignored.
- Missing parameters are set to undefined.

For example:

```

function foo(x, y) {
    return [x, y];
}

// Too many arguments:
assert.deepEqual(foo('a', 'b', 'c'), ['a', 'b']);

// The expected number of arguments:
assert.deepEqual(foo('a', 'b'), ['a', 'b']);

// Not enough arguments:
assert.deepEqual(foo('a'), ['a', undefined]);

```

### 22.6.3 Parameter default values

Parameter default values specify the value to use if a parameter has not been provided. For example:

```
function f(x, y=0) {
  return [x, y];
}

assert.deepEqual(f(1), [1, 0]);
assert.deepEqual(f(), [undefined, 0]);
```

undefined also triggers the default value:

```
assert.deepEqual(
  f(undefined, undefined),
  [undefined, 0]);
```

### 22.6.4 Rest parameters

A rest parameter is declared by prefixing an identifier with three dots (...). During a function or method call, it receives an Array with all remaining arguments. If there are no extra arguments at the end, it is an empty Array. For example:

```
function f(x, ...y) {
  return [x, y];
}

assert.deepEqual(
  f('a', 'b', 'c'),
  ['a', ['b', 'c']]);
assert.deepEqual(
  f(),
  [undefined, []]);
```

#### 22.6.4.1 Enforcing an arity via a rest parameter

You can use rest parameters to enforce arities. Take, for example, the following function.

```
function bar(a, b) {
  // ...
}
```

This is how we force callers to always provide two arguments:

```
function bar(...args) {
  if (args.length !== 2) {
    throw new Error('Please provide exactly 2 arguments!');
  }
  const [a, b] = args;
  // ...
}
```

### 22.6.5 Named parameters

When someone calls a function, the arguments provided by the caller are assigned to the parameters received by the callee. Two common ways of performing the mapping are:

1. Positional parameters: An argument is assigned to a parameter if they have the same position. A function call with only positional arguments looks as follows.

```
selectEntries(3, 20, 2)
```

2. Named parameters: An argument is assigned to a parameter if they have the same name. JavaScript doesn't have named parameters, but you can simulate them. For example, this is a function call with only (simulated) named arguments:

```
selectEntries({start: 3, end: 20, step: 2})
```

Named parameters have several benefits:

- They lead to more self-explanatory code, because each argument has a descriptive label. Just compare the two versions of `selectEntries()`: With the second one, it is much easier to see what happens.
- Order of parameters doesn't matter (as long as the names are correct).
- Handling more than one optional parameter is more convenient: Callers can easily provide any subset of all optional parameters and don't have to be aware of the ones they omitted (with positional parameters, you have to fill in preceding optional parameters, with `undefined`).

### 22.6.6 Simulating named parameters

JavaScript doesn't have real named parameters. The official way of simulating them is via object literals:

```
function selectEntries({start=0, end=-1, step=1}) {
  return {start, end, step};
}
```

This function uses *destructuring* to access the properties of its single parameter. The pattern it uses is an abbreviation for the following pattern:

```
{start: start=0, end: end=-1, step: step=1}
```

This destructuring pattern works for empty object literals:

```
> selectEntries({})
{ start: 0, end: -1, step: 1 }
```

But it does not work if you call the function without any parameters:

```
> selectEntries()
TypeError: Cannot destructure property `start` of 'undefined' or 'null'.
```

You can fix this by providing a default value for the whole pattern. This default value works the same as default values for simpler parameter definitions: If the parameter is missing, the default is used.

```
function selectEntries({start=0, end=-1, step=1} = {}) {
  return {start, end, step};
}
```

```
assert.deepEqual(  
  selectEntries(),  
  { start: 0, end: -1, step: 1 });
```

### 22.6.7 Spreading (...) into function calls

The prefix (...) of a spread argument is the same as the prefix of a rest parameter. The former is used when calling functions or methods. Its operand must be an iterable object. The iterated values are turned into positional arguments. For example:

```
function func(x, y) {  
  console.log(x);  
  console.log(y);  
}  
const someIterable = ['a', 'b'];  
func(...someIterable);  
  
// Output:  
// 'a'  
// 'b'
```

Therefore, spread arguments and rest parameters serve opposite purposes:

- Rest parameters are used when defining functions or methods. They collect arguments in Arrays.
- Spread arguments are used when calling functions or methods. They turn iterable objects into arguments.

#### 22.6.7.1 Example: spreading into Math.max()

`Math.max()` returns the largest one of its zero or more arguments. Alas, it can't be used for Arrays, but spreading gives us a way out:

```
> Math.max(-1, 5, 11, 3)  
11  
> Math.max(...[-1, 5, 11, 3])  
11  
> Math.max(-1, ...[-5, 11], 3)  
11
```

#### 22.6.7.2 Example: spreading into Array.prototype.push()

Similarly, the Array method `.push()` destructively adds its zero or more parameters to the end of its Array. JavaScript has no method for destructively appending an Array to another one, but once again we are saved by spreading:

```
const arr1 = ['a', 'b'];  
const arr2 = ['c', 'd'];
```



```
arr1.push(...arr2);
assert.deepEqual(arr1, ['a', 'b', 'c', 'd']);
```



### Exercises: Parameter handling

- Positional parameters: `exercises/callables/positional_parameters_test.js`
- Named parameters: `exercises/callables/named_parameters_test.js`

## 22.7 Understanding JavaScript's callable values (advanced)

In order to better understand JavaScript's many callable values, it is helpful to distinguish between:

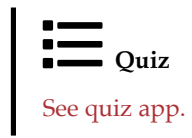
- Syntax: what is written in source code?
- Semantics: what is the result of executing the source code?

The following two concepts help with categorizing callable values:

- Role: What role does a callable value play? The roles are:
  - Real function
  - Method
  - Class (constructor function)
- Mode: What mode does a callable value operate in?
  - Is the real function or method synchronous or asynchronous?
  - Is the real function or method a generator or not?

Let's look at a few examples:

- `function foo() {}`
  - Syntax: sync function declaration
  - Semantics: ordinary function
  - Potential roles: real function, method, class
  - Mode: synchronous, not a generator
- `* m() {}`
  - Syntax: generator method definition
  - Semantics: generator method
  - Potential roles: method
  - Mode: synchronous generator
- `() => {}`
  - Syntax: arrow function expression
  - Semantics: arrow function
  - Potential roles: real function
  - Mode: synchronous, not a generator
- `async () => {}`
  - Syntax: async arrow function expression
  - Semantics: async arrow function
  - Potential roles: real function
  - Mode: asynchronous, not a generator



## Chapter 23

# Variable scopes and closures (advanced)

### Contents

---

<b>23.1 The scope of a variable</b>	<b>188</b>
<b>23.2 Terminology: static versus dynamic</b>	<b>188</b>
23.2.1 Static phenomenon: scopes of variables	188
23.2.2 Dynamic phenomenon: function calls	189
<b>23.3 Temporal dead zone (TDZ)</b>	<b>189</b>
<b>23.4 Hoisting</b>	<b>190</b>
<b>23.5 Global variables</b>	<b>190</b>
23.5.1 The global object	191
23.5.2 Avoid the global object!	192
<b>23.6 Ways of declaring variables</b>	<b>192</b>
<b>23.7 Variables in detail: environments</b>	<b>192</b>
<b>23.8 Recursion via environments</b>	<b>193</b>
23.8.1 Executing the code	193
<b>23.9 Nested scopes via environments</b>	<b>194</b>
23.9.1 Executing the code	195
<b>23.10 Closures</b>	<b>197</b>
23.10.1 Bound variables versus free variables	197
23.10.2 What is a closure?	197
23.10.3 Example: Currying <code>add()</code>	197
23.10.4 Example: A factory for incrementors	198
23.10.5 Use cases for closures	200

---

In this chapter, we cover in more depth, how JavaScript handles variables. There are six ways of declaring variables: `var`, `let`, `const`, `function`, `class` and `import`. We initially explore `let` and `const` and later apply what we have learned to the remaining constructs.

## 23.1 The scope of a variable

The *scope* of a variable is the part of the program in which the variable exists. Consider the following code.

```
{ // (A)
  let x = 0;
  // This scope: can access x
  { // (B)
    let y = 1;
    // This scope: can access x, y
    { // (C)
      let z = 2;
      // This scope: can access x, y, z
    }
  }
}
```

- Scope A is the (*direct*) *scope* of `x`.
- Scopes B and C are *inner scopes* of scope A.
  - The variables of a scope can be accessed from all of its inner scopes.
- Scope A is an *outer scope* of scope B and scope C.

The scope of a variable declared via `const` or `let` is always the directly surrounding block. That's why these declarations are called *block-scoped*.

## 23.2 Terminology: static versus dynamic

These two adjectives describe phenomena in programming languages:

- *Static* means that something is related to source code and can be determined without executing code.
- *Dynamic* means at runtime.

Let's look at examples for these two terms.

### 23.2.1 Static phenomenon: scopes of variables

Variable scopes are a static phenomenon. Consider the following code:

```
function f() {
  const x = 3;
  // ...
}
```

`x` is *statically* (or *lexically*) *scoped*. That is, its scope is fixed and doesn't change at runtime.

Variable scopes form a static tree (via static nesting).

### 23.2.2 Dynamic phenomenon: function calls

Function calls are a dynamic phenomenon. Consider the following code:

```
function g(x) {}
function h(y) {
  if (Math.random()) g(y); // (A)
}
```

Whether or not the function call in line A happens, can only be decided at runtime.

Function calls form a dynamic tree (via dynamic calls).

## 23.3 Temporal dead zone (TDZ)

Declaring a variable, influences its whole scope, because you can't declare the same variable twice (directly) inside the same scope.

Thus, for JavaScript, TC39 needed to decide what happens if you enter a new scope and mention the name of a variable before its declaration. Two possible approaches are:

- The name is resolved in the scope surrounding the current scope.
- There is an error.

TC39 chose the latter for `const` and `let`, because you likely made a mistake, if you use a name that is declared later in the same scope.

The time between entering the scope of a variable and executing its declaration is called the *temporal dead zone* of that variable:

- During this time, the variable is considered to be uninitialized (as if that were a special value it has).
- If you access an uninitialized variable, you get a `ReferenceError`.
- Once you reach a variable declaration, the variable is set to either the value of the initializer (specified via the assignment symbol) or `undefined` – if there is no initializer.

The following code illustrates the temporal dead zone:

```
if (true) { // enter scope of `tmp`, TDZ starts
  // `tmp` is uninitialized:
  assert.throws(() => (tmp = 'abc'), ReferenceError);
  assert.throws(() => console.log(tmp), ReferenceError);

  let tmp; // TDZ ends
  assert.equal(tmp, undefined);

  tmp = 123;
  assert.equal(tmp, 123);
}
```

The next example shows that the temporal dead zone is truly *temporal* (related to time):

```

if (true) { // enter scope of `myVar`, TDZ starts
  const func = () => {
    console.log(myVar); // executed later
  };

  // We are within the TDZ:
  // Accessing `myVar` causes `ReferenceError`

  let myVar = 3; // TDZ ends
  func(); // OK, called outside TDZ
}

```

Even though `func()` is located before the declaration of `myVar` and uses that variable, we can call `func()`. But we have to wait until the temporal dead zone of `myVar` is over.

## 23.4 Hoisting

Hoisting means that a construct is moved to the beginning of its scope, regardless of where it is located in that scope. We have already seen hoisting in [the chapter on callable entities](#):

```

assert.equal(func(), 123); // Works!

function func() {
  return 123;
}

```

You can use `func()` before its declaration, because, internally, it is hoisted. That is, the previous code is actually executed like this:

```

function func() {
  return 123;
}

assert.equal(func(), 123);

```

It's nice that we can use a function before declaring it. That gives us more freedom in structuring our code.

The temporal dead zone can also be viewed as a form of hoisting, because the declaration affects what happens at the beginning of its scope.

## 23.5 Global variables

A variable is global if it is declared in the top-level scope. Every nested scope can access such a variable. In JavaScript, there are multiple layers of global scopes (Fig. 23.1):

- The outermost global scope is special: its variables can be accessed via the properties of an object, the so-called *global object*. The global object is referred to by `window` and `self` in browsers. Variables in this scope are created via:

- Properties of the global object
- `var` and function at the top level of a script
- Nested in that scope is the global scope of scripts. Variables in this scope are created by `let`, `const` and `class` at the top level of a script.
- Nested in that scope are the scopes of modules. Each module has its own global scope. Variables in that scope are created by declarations at the top level of the module.

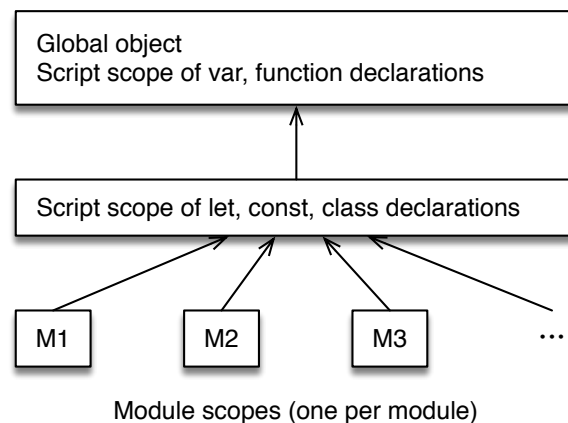


Figure 23.1: JavaScript has multiple global scopes.

### 23.5.1 The global object

The global object lets you access the outermost global scope via an object. The two are always in sync:

- If you create a variable in the outermost global scope, the global object gets a new property. If you change such a global variable, the property changes.
- If you create or delete a property of the global object, the corresponding global variable is created or deleted. If you change a property of the global object, the corresponding global variable changes.

The global object is available via special variables:

- `window`: is the classic way of referring to the global object. But it only works in normal browser code, not in Web Workers (processes running concurrently to normal browser code) and not in Node.js
- `self`: is available everywhere in browsers, including in Web Workers. But it isn't supported by Node.js.
- `global`: is only available in Node.js.

Let's examine how `self` works:

```

// At the top level of a script
var myGlobalVariable = 123;
assert.ok('myGlobalVariable' in self);

delete self.myGlobalVariable;
assert.throws(() => console.log(myGlobalVariable), ReferenceError);
  
```

```
// Create a global variable anywhere:
if (true) {
  self.anotherGlobalVariable = 'abc';
}
assert.equal(anotherGlobalVariable, 'abc');
```

### 23.5.2 Avoid the global object!

Brendan Eich called the global object one of his biggest regrets about JavaScript. It is best not to put variables into its scope:

- In general, variables that are global to all scripts on a web page, risk name clashes.
- Via the global object, you can create and delete global variables anywhere. Doing so makes code unpredictable, because it's normally not possible to make this kind of change in nested scopes.

You occasionally see `window.globalVariable` in tutorials on the web, but the prefix “`window.`” is not necessary. I prefer to omit it:

```
window.encodeURIComponent(str); // no
encodeURIComponent(str); // yes
```

## 23.6 Ways of declaring variables

Table 23.1: These are all the ways in which you declare variables in JavaScript.

	Hoisting	Scope	Script scope is global object?
<code>var</code>	Declaration only	Function	✓
<code>let</code>	Temporal dead zone	Block	✗
<code>const</code>	Temporal dead zone	Block	✗
<code>function</code>	Everything	Block	✓
<code>class</code>	No	Block	✗
<code>import</code>	Everything	Module	✗

Tbl. 23.1 lists all ways in which you can declare variables in JavaScript: `var`, `let`, `const`, `function`, `class` and `import`.

## 23.7 Variables in detail: environments

Environments are the data structure that JavaScript engines use to manage variables. You need to know how they work if you want to understand an important advanced JavaScript concept: *closures*.

An environment is a dictionary whose keys are variable names and whose values are the values of those



variables. Each scope has its associated environment. Environments must be able to support the following phenomena related to variables:

- Recursion
- Nested scopes
- Closures

We'll use examples to illustrate how that is done for each phenomenon.

## 23.8 Recursion via environments

We'll tackle recursion first. Consider the following code:

```
function f(x) {  
    return x * 2;  
}  
function g(y) {  
    const tmp = y + 1;  
    return f(tmp);  
}  
assert.equal(g(3), 8);
```

For each function call, you need fresh storage space for the variables (parameters and local variables) of the called function. This is managed via a stack of so-called *execution contexts*, which – for the purpose of this chapter, are references to environments. Environments themselves are stored on the heap and can therefore be *shared* via references (more than one storage location can point to them).

### 23.8.1 Executing the code

While executing the code, we make the following pauses:

```
function f(x) {  
    // Pause 3  
    return x * 2;  
}  
function g(y) {  
    const tmp = y + 1;  
    // Pause 2  
    return f(tmp);  
}  
// Pause 1  
assert.equal(g(3), 8);
```

This is what happens:

- Pause 1 – before calling `g()` (fig. 23.2).
- Pause 2 – while executing `g()` (fig. 23.3).
- Pause 3 – while executing `f()` (fig. 23.4).
- Remaining steps: Every time there is a `return`, one execution context is removed from the stack.

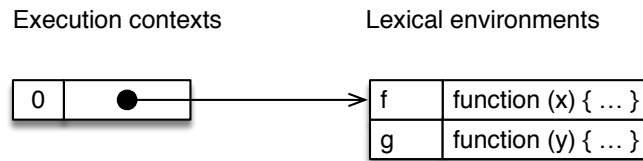


Figure 23.2: Recursion, pause 1 – before calling `g()`: The execution context stack has one entry, which points to the global environment. In that environment, there are two entries; one for `f()` and one for `g()`.

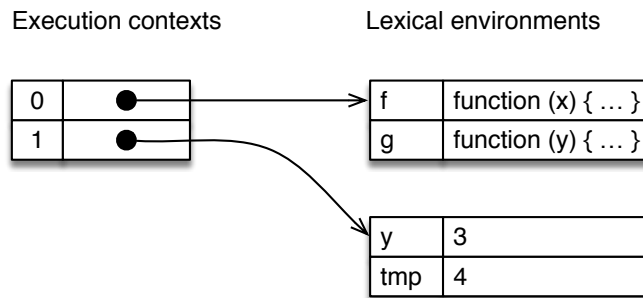


Figure 23.3: Recursion, pause 2 – while executing `g()`: The top of the execution context stack points to the environment that was created for `g()`. That environment contains entries for the argument `y` and for the local variable `tmp`.

## 23.9 Nested scopes via environments

We use the following code to explore how nested scopes are implemented via environments.

```
function f(x) {
  function square() {
    const result = x * x;
    return result;
  }
  return square();
}
assert.equal(f(6), 36);
```

The idea is that the environment of each scope points to the environment of the surrounding scope, via a field called `outer`. That is, the latter environment is the outer environment of the former environment. When we are looking up the value of a variable, we first search for its name in the current environment, then in the outer environment, then in the outer environment's outer environment, etc. The whole chain of outer environments contains all variables that can currently be accessed (if we are ignoring shadowed variables).

When you make a function call, you create a new environment. The outer environment of that environment is the environment in which the function is stored – that's how static scoping works. To help set up the field `outer` of environments created via function calls, each function has an internal property named `[[Scope]]` that points to its "birth environment".

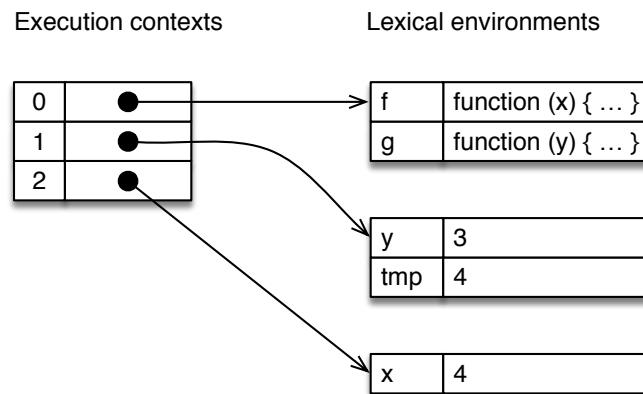


Figure 23.4: Recursion, pause 3 – while executing `f()`: The top execution context now points to the environment for `f()`.

### 23.9.1 Executing the code

These are the pauses we are making while executing the code:

```
function f(x) {
  function square() {
    const result = x * x;
    // Pause 3
    return result;
  }
  // Pause 2
  return square();
}
// Pause 1
assert.equal(f(6), 36);
```

This is what happens:

- Pause 1 – before calling `f()` (fig. 23.5).
- Pause 2 – while executing `f()` (fig. 23.6).
- Pause 3 – while executing `square()` (fig. 23.7).
- After that, return statements once again pop execution entries off the stack.

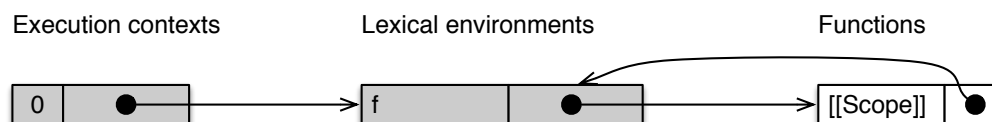


Figure 23.5: Nested scopes, pause 1 – before calling `f()`: The global environment has a single entry, for `f()`. The birth environment of `f()` is the global environment. Therefore, `f`'s `[[Scope]]` points to it.

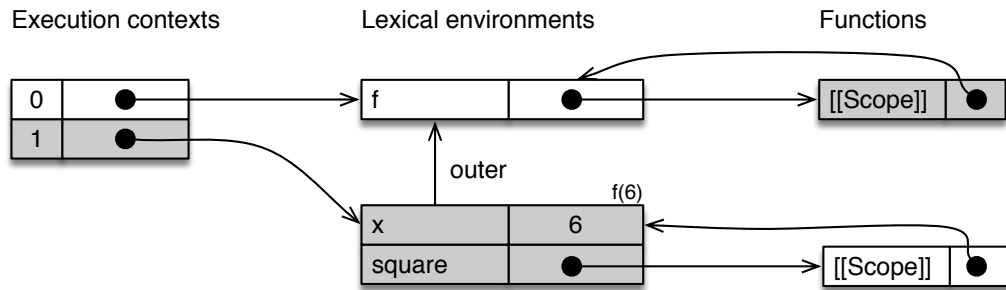


Figure 23.6: Nested scopes, pause 2 – while executing `f()`: There is now an environment for the function call `f(6)`. The outer environment of that environment is the birth environment of `f()` (the global environment at index 0). We can see that the field `outer` was set to the value of `f`'s `[[Scope]]`. Furthermore, the `[[Scope]]` of the new function `square()` is the environment that was just created.

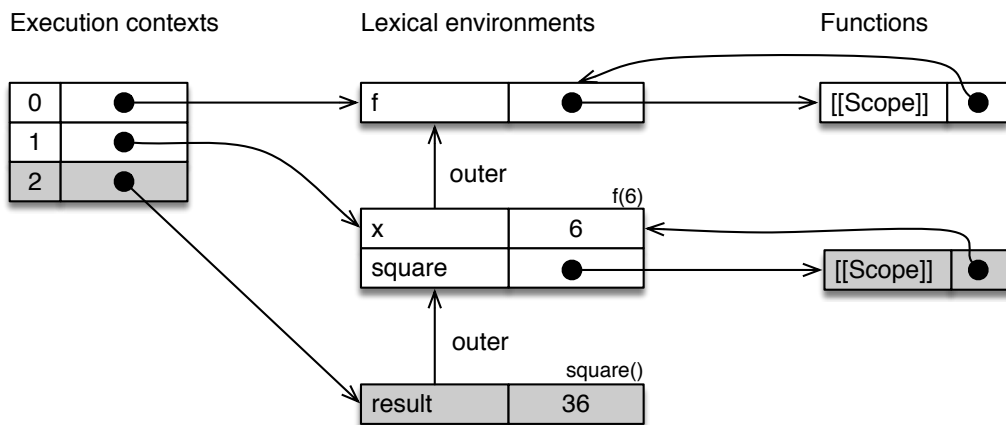


Figure 23.7: Nested scopes, pause 3 – while executing `square()`: The previous pattern was repeated: the `outer` of the most recent environment was set up via the `[[Scope]]` of the function that we just called. The chain of scopes created via `outer`, contains all variables that are active right now. For example, we can access `result`, `square` and `f`, if we want to. Environments reflect two aspects of variables. First, the chain of outer environments reflects the nested static scopes. Second, the stack of execution contexts reflects what function calls were made, dynamically.

## 23.10 Closures

Before we can explore closures, we need to learn about bound variables and free variables.

### 23.10.1 Bound variables versus free variables

Per scope, there is a set of variables that are mentioned. Among these variables we distinguish:

- *Bound variables* are declared within the scope. They are parameters and local variables.
- *Free variables* are declared externally. They are also called *non-local variables*.

Consider the following code:

```
function func(x) {  
  const y = 123;  
  console.log(z);  
}
```

Within (the scope of) `func()`, `x` and `y` are bound variables. `z` is a free variable.

### 23.10.2 What is a closure?

So what is a closure?

A *closure* is a function plus its “birth environment” (the environment that was active when the function was created).

What is the point of packaging the two? The birth environment provides the values for the free variables of the function. For example:

```
function funcFactory(value) {  
  return function () {  
    return value;  
  };  
}
```

```
const func = funcFactory('abc');  
assert.equal(func(), 'abc'); // (A)
```

`funcFactory` returns a closure that is assigned to `func`. Because `func` has the connection to the variables at its birth place, it can still access the free variable `value` when it is called in line A (even though it “escaped” its scope).

All functions in JavaScript are closures. We saw that when we were looking at how environments support nested scopes: Each function has the internal property `[[Scope]]` that points to its birth environment.

### 23.10.3 Example: Currying `add()`

The following code demonstrates another closure:

```
function add(x) {
  return function (y) { // (A)
    return x + y;
  };
}
assert.equal(add(3)(1), 4); // (B)
```

What is going on here? `add()` is a function that returns a function. When we make the nested function call `add(3)(1)` in line B, the first parameter is for `add()`, the second parameter is for the function it returns. This trick works, because the function created in line A does not lose the connection to its birth scope when it leaves that scope. The associated environment is kept alive by that connection and the function still has access to variable `x` in that environment (`x` is free inside the function).

This nested way of calling `add()` has an advantage: If you only make the first function call, you get a version of `add()` whose parameter `x` is already filled in:

```
const plus2 = add(2);
assert.equal(plus2(5), 7);
```

Converting a function with two parameters into two nested functions with one parameter each, is called *currying*. `add()` is a *curried function*.

Only filling in some of the parameters of a function is called *partial application* (the function has not been fully applied, yet). Method `.bind()` of functions performs partial application. In the previous example, we can see that partial application is simple if a function is curried.

### 23.10.3.1 Executing the code

As we are executing the following code, we are making three pauses:

```
function add(x) {
  return function (y) {
    // Pause 3: plus2(5)
    return x + y;
  }; // Pause 1: add(2)
}
const plus2 = add(2);
// Pause 2
assert.equal(plus2(5), 7);
```

This is what happens:

- Pause 1 – during the execution of `add(2)` (fig. 23.8).
- Pause 2 – after the execution of `add(2)` (fig. 23.9).
- Pause 3 – while executing `plus2(5)` (fig. 23.10).

### 23.10.4 Example: A factory for incrementors

The following function returns *incrementors* (a name that I just invented). An incrementor is a function that internally stores a number. When it is called, it updates that number by adding the argument to it and returns the new value.

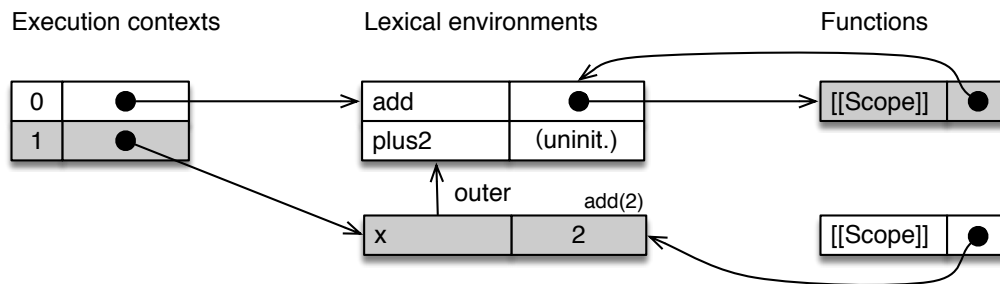


Figure 23.8: Closures, pause 1 – during the execution of `add(2)`: We can see that the function returned by `add()` already exists (see bottom right corner) and that it points to its birth environment via its internal property `[[Scope]]`. Note that `plus2` is still in its temporal dead zone and uninitialized.

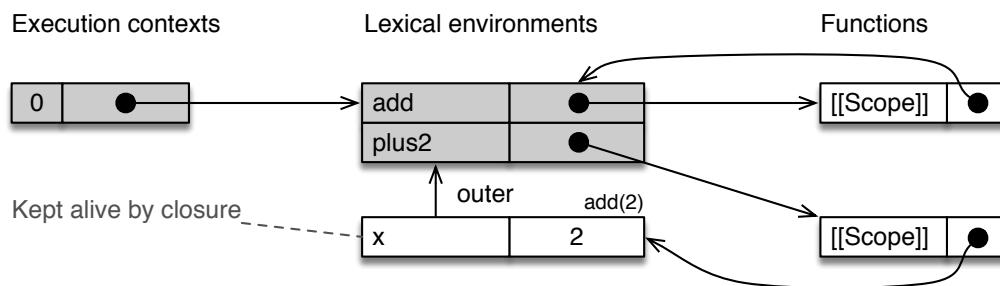


Figure 23.9: Closures, pause 2 – after the execution of `add(2)`: `plus2` now points to the function returned by `add(2)`. That function keeps its birth environment (the environment of `add(2)`) alive via its `[[Scope]]`.

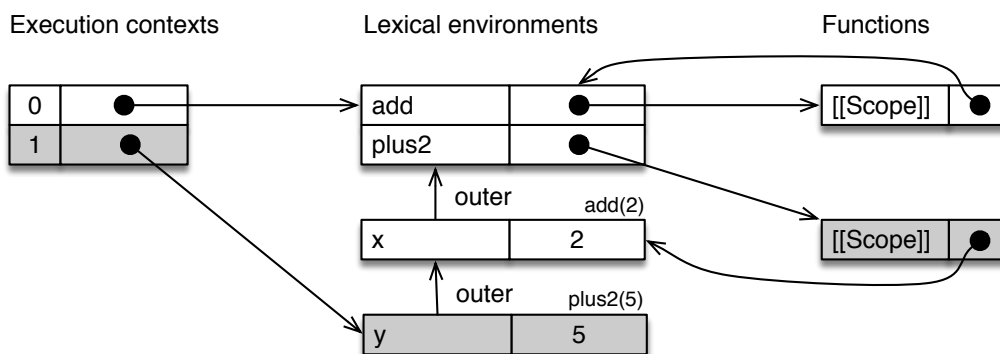


Figure 23.10: Closures, pause 3 – while executing `plus2(5)`: The `[[Scope]]` of `plus2(5)` is used to set up the `outer` of the new environment. That's how the current function gets access to `x`.

```
function createInc(startValue) {
  return function (step) { // (A)
    startValue += step;
    return startValue;
  };
}
const inc = createInc(5);
assert.equal(inc(2), 7);
```

We can see that the function created in line A keeps its internal number in the free variable `startValue`. This time, we don't just read from the birth environment, we use it to store data that we change and that persists across function calls.

We can create more storage slots in the birth environment, via local variables:

```
function createInc(startValue) {
  let index = -1;
  return function (step) {
    startValue += step;
    index++;
    return [index, startValue];
  };
}
const inc = createInc(5);
assert.deepEqual(inc(2), [0, 7]);
assert.deepEqual(inc(2), [1, 9]);
assert.deepEqual(inc(2), [2, 11]);
```

### 23.10.5 Use cases for closures

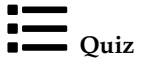
So what are closures good for?

- For starters, they are simply an implementation of static scoping. As such, they provide context data for callbacks.
- They can also be used by functions, to store state that persists across function calls. `createInc()` is an example of that.
- And they can provide private data for objects. For example:

```
function createStringBuilder() {
  let data = ''; // private data
  return {
    add(str) {
      data += str;
    },
    toString() {
      return data;
    },
  };
}
const sb = createStringBuilder();
```



```
sb.add('Hello');  
sb.add(' world');  
sb.add('!');  
assert.equal(sb.toString(), 'Hello world!');
```

**Quiz**

See quiz app.



**Part VI**

**Modularity**



# Chapter 24

## Modules

### Contents

<b>24.1 Before modules: scripts</b>	<b>205</b>
<b>24.2 Module systems created prior to ES6</b>	<b>206</b>
24.2.1 Server side: CommonJS modules	207
24.2.2 Client side: AMD (Asynchronous Module Definition) modules	207
24.2.3 Characteristics of JavaScript modules	208
<b>24.3 ECMAScript modules</b>	<b>208</b>
24.3.1 ECMAScript modules: three parts	209
<b>24.4 Named exports</b>	<b>209</b>
<b>24.5 Default exports</b>	<b>210</b>
24.5.1 The two styles of default-exporting	210
<b>24.6 Naming modules</b>	<b>211</b>
<b>24.7 Imports are read-only views on exports</b>	<b>212</b>
<b>24.8 Module specifiers</b>	<b>212</b>
<b>24.9 Syntactic pitfall: importing is not destructuring</b>	<b>213</b>
<b>24.10 Preview: loading modules dynamically</b>	<b>214</b>
<b>24.11 Further reading</b>	<b>215</b>

The current landscape of JavaScript modules is quite diverse: ES6 brought built-in modules, but the module systems that came before them, are still around, too. Understanding the latter helps understand the former, so let's investigate.

### 24.1 Before modules: scripts

Initially, browsers only had *scripts* – pieces of code that were executed in global scope. As an example, consider an HTML file that loads a *script file* via the following HTML element:

```
<script src="my-library.js"></script>
```

In the script file, we simulate a module:

```

var myLibrary = function () { // Open IIFE
  // Imports (via global variables)
  var importedFunc1 = otherLibrary1.importedFunc1;
  var importedFunc2 = otherLibrary2.importedFunc2;

  function internalFunc() {
    // ...
  }

  function exportedFunc() {
    importedFunc1();
    importedFunc2();
    internalFunc();
    // ...
  }

  // Exports (assigned to global variable `myLibrary`)
  return {
    exportedFunc: exportedFunc,
  };
}(); // Close IIFE

```

Before we get to real modules (which were introduced with ES6), all code is written in ES5 (which didn't have `const` and `let`, only `var`).

`myModule` is a global variable. The code that defines the module is wrapped in an **immediately invoked function expression**. Creating a function and calling it right away, only has one benefit compared to executing the code directly (without wrapping it): All variables defined inside the IIFE, remain local to its scope and don't become global. At the end, we pick what we want to export and return it via an object literal. This pattern is called the *revealing module pattern* (coined by Christian Heilmann).

This way of simulating modules has several problems:

- Libraries in script files export and import functionality via global variables, which risks name clashes.
- Dependencies are not stated explicitly and there is no built-in way for a script to load the scripts it depends on. Therefore, the web page has to load not just the scripts that are needed by the page, but also the dependencies of those scripts, the dependencies' dependencies, etc. And it has to do so in the right order!

## 24.2 Module systems created prior to ES6

Prior to ECMAScript 6, JavaScript did not have built-in modules. Therefore, the flexible syntax of the language was used to implement custom module systems *within* the language. Two popular ones are CommonJS (targeted at the server side) and AMD (Asynchronous Module Definition, targeted at the client side).

### 24.2.1 Server side: CommonJS modules

The original CommonJS standard for modules was mainly created for server and desktop platforms. It was the foundation of the module system of Node.js where it achieved incredible popularity. Contributing to that popularity were Node's package manager, npm, and tools that enabled using Node modules on the client side (browserify and webpack).

From now on, I use the terms *CommonJS module* and *Node.js module* interchangeably, even though Node.js has a few additional features. The following is an example of a Node.js module.

```
// Imports
var importedFunc1 = require('other-module1').importedFunc1;
var importedFunc2 = require('other-module2').importedFunc2;

function internalFunc() {
  // ...
}

function exportedFunc() {
  importedFunc1();
  importedFunc2();
  internalFunc();
  // ...
}

// Exports
module.exports = {
  exportedFunc: exportedFunc,
};
```

CommonJS can be characterized as follows:

- Designed for servers.
- Modules are meant to be loaded synchronously.
- Compact syntax.

### 24.2.2 Client side: AMD (Asynchronous Module Definition) modules

The AMD module format was created to be easier to use in browsers than the CommonJS format. Its most popular implementation is RequireJS. The following is an example of a RequireJS module.

```
define(['other-module1', 'other-module2'],
function (otherModule1, otherModule2) {
  var importedFunc1 = otherModule1.importedFunc1;
  var importedFunc2 = otherModule2.importedFunc2;

  function internalFunc() {
    // ...
  }

  function exportedFunc() {
```

```

    importedFunc1();
    importedFunc2();
    internalFunc();
    // ...
}
return {
    exportedFunc: exportedFunc,
};
});

```

AMD can be characterized as follows:

- Designed for browsers.
- Modules are meant to be loaded asynchronously. That's a crucial requirement for browsers, where code can't wait until a module has finished downloading. It has to be notified once the module is available.
- The syntax is slightly more complicated. On the plus side, AMD modules can be executed directly, without customized creation and execution of source code (think `eval()`). That is not always permitted on the web.

### 24.2.3 Characteristics of JavaScript modules

Looking at CommonJS and AMD, similarities between JavaScript module systems emerge:

- There is one module per file (AMD also supports more than one module per file).
- Such a file is basically a piece of code that is executed:
  - Exports: That code contains declarations (variables, functions, etc.). By default, those declarations remain local to the module, but you can mark some of them as exports.
  - Imports: The module can import entities from other modules. Those other modules are identified via *module specifiers* (usually paths, occasionally URLs).
- Modules are *singletons*: Even if a module is imported multiple times, only a single instance of it exists.
- No global variables are used. Instead, module specifiers serve as global IDs.

## 24.3 ECMAScript modules

ECMAScript modules were introduced with ES6: They stand firmly in the tradition of JavaScript modules and share many of the characteristics of existing module systems:

- With CommonJS, ES modules share the compact syntax, better syntax for single exports than for *named exports* (so far, we have only seen named exports) and support for cyclic dependencies.
- With AMD, ES modules share a design for asynchronous loading and configurable module loading (e.g. how specifiers are resolved).

ES modules also have new benefits:

- Their syntax is even more compact than CommonJS's.
- Their modules have a static structure (that can't be changed at runtime). That enables static checking, optimized access of imports, better bundling (delivery of less code) and more.



- Their support for cyclic imports is completely transparent.

This is an example of ES module syntax:

```
import {importedFunc1} from 'other-module1';
import {importedFunc2} from 'other-module2';

function internalFunc() {
  ...
}

export function exportedFunc() {
  importedFunc1();
  importedFunc2();
  internalFunc();
  ...
}
```

From now on, “module” means “ECMAScript module”.

### 24.3.1 ECMAScript modules: three parts

ECMAScript modules comprise three parts:

1. Declarative module syntax: What is a module? How are imports and exports declared?
2. The semantics of the syntax: How are the variable bindings handled that are created by imports? How are exported variable bindings handled?
3. A programmatic loader API for configuring module loading.

Parts 1 and 2 were introduced with ES6. Work on Part 3 is ongoing.

## 24.4 Named exports

Each module can have zero or more named exports.

As an example, consider the following three files:

```
lib/my-math.js
main1.js
main2.js
```

Module `my-math.js` has two named exports: `square` and `MY_CONSTANT`.

```
let notExported = 'abc';
export function square(x) {
  return x * x;
}
export const MY_CONSTANT = 123;
```

Module `main1.js` has a single named import, `square`:

```
import {square} from './lib/my-math.js';
assert.equal(square(3), 9);
```

Module `main2.js` has a so-called *namespace import* – all named exports of `my-math.js` can be accessed as properties of the object `myMath`:

```
import * as myMath from './lib/my-math.js';
assert.equal(myMath.square(3), 9);
```



### Exercise: Named exports

`exercises/modules/export_named_test.js`

## 24.5 Default exports

Each module can have at most one default export. The idea is that the module *is* the default-exported value. A module can have both named exports and a default export, but it's usually better to stick to one export style per module.

As an example for default exports, consider the following two files:

`my-func.js`  
`main.js`

Module `my-func.js` has a default export:

```
export default function () {
  return 'Hello!';
}
```

Module `main.js` default-imports the exported function:

```
import myFunc from './my-func.js';
assert.equal(myFunc(), 'Hello!');
```

Note the syntactic difference: The curly braces around named imports indicate that we are reaching *into* the module, while a default import *is* the module.

The most common use case for a default export is a module that contains a single function or a single class.

### 24.5.1 The two styles of default-exporting

There are two styles of doing default exports.

First, you can label existing declarations with `export default`:

```
export default function foo() {} // no semicolon!
export default class Bar {} // no semicolon!
```

Second, you can directly default-export values. In that style, `export default` is itself much like a declaration.

```
export default 'abc';
export default foo();
export default /^xyz$/;
export default 5 * 7;
export default { no: false, yes: true };
```

Why are there two default export styles? The reason is that `export default` can't be used to label `const`: `const` may define multiple values, but `export default` needs exactly one value.

```
// Not legal JavaScript!
export default const foo = 1, bar = 2, baz = 3;
```

With this hypothetical code, you don't know which one of the three values is the default export.



### Exercise: Default exports

`exercises/modules/export_default_test.js`

## 24.6 Naming modules

There are no established best practices for naming module files and the variables they are imported into.

In this chapter, I've used the following naming style:

- The names of module files are dash-cased and start with lowercase letters:

```
./my-module.js
./some-func.js
```

- The names of namespace imports are lowercased and camel-cased:

```
import * as myModule from './my-module.js';
```

- The names of default imports are lowercased and camel-cased:

```
import someFunc from './some-func.js';
```

What are the rationales behind this style?

- npm doesn't allow uppercase letters in package names (source<sup>1</sup>). Thus, we avoid camel-case, so that "local" files have names that are consistent with those of npm packages.
- There are clear rules for translating dashed-cased file names to camel-cased JavaScript variable names. Due to how we name namespace imports, these rules work for both namespace imports and default imports.

I also like underscore-cased module file names, because you can directly use these names for namespace imports (without any translation):

```
import * as my_module from './my_module.js';
```

But that style does not work for default imports: I like underscore-casing for namespace objects, but it is not a good choice for functions etc.

<sup>1</sup>[npm%20doesn't%20allow%20uppercase%20letters](#)

## 24.7 Imports are read-only views on exports

So far, we have used imports and exports intuitively and everything seems to have worked as expected. But now it is time to take a closer look at how imports and exports are really related.

Consider the following two modules:

```
counter.js
main.js
```

counter.js exports a (mutable!) variable and a function:

```
export let counter = 3;
export function incCounter() {
  counter++;
}
```

main.js name-imports both exports. When we use incCounter(), we discover that the connection to counter is live – we can always access the live state of that variable:

```
import { counter, incCounter } from './counter.js';
```

```
// The imported value `counter` is live
assert.equal(counter, 3);
incCounter();
assert.equal(counter, 4);
```

Note that, while the connection is live and we can read counter, we cannot change this variable (e.g. via counter++).

Why do ES modules behave this way?

First, it is easier to split modules, because previously shared variables can become exports and imports.

Second, this behavior is crucial for cyclic imports. The exports of a module are known before executing it. Therefore, if a module L and a module M import each other, cyclically, the following steps happen:

- The execution of L starts.
  - L imports M. L's imports point to uninitialized slots inside M.
  - L's body is not executed, yet.
- The execution of M starts (triggered by the import).
  - M imports L.
  - The body of M is executed. Now L's imports have values (due to the live connection).
- The body of L is executed. Now M's imports have values.

Cyclic imports are something that you should avoid as much as possible, but they can arise in complex systems or when refactoring systems. It is important that things don't break when that happens.

## 24.8 Module specifiers

One key rule is:

All ES modules specifiers must be valid URLs.

That means that paths to sibling files must include file name extensions. Beyond that, everything is still in flux. CommonJS distinguishes several kinds of module specifiers:

- Relative paths: start with a dot. Examples:  
 `'./some/other/module.js'`  
 `'../..../lib/counter.js'`
- Absolute paths: start with a slash. Example:  
 `'/home/jane/file-tools.js'`
- URLs: include a protocol (technically, paths are URLs, too). Example:  
 `'https://example.com/some-module.js'`
- Bare paths: do not start with a dot, a slash or a protocol. Additionally, Node.js set the precedence of them never having file name extensions.  
 `'lodash'`  
 `'mylib/string-tools'`

In Node.js, module specifiers are handled as follows:

- Relative and absolute paths are interpreted as expected. In Node.js, module specifiers often have no file name extension.
- Only file: URLs are supported, at the moment.
- Bare paths are looked up in `node_modules`, where npm installs packages.

Browsers handle module specifiers as follows:

- Relative paths, absolute paths and URLs are straightforward and handled as expected. You must include a file name extension.
- How bare paths will end up being handled is not yet clear. You may eventually be able to map them to other specifiers via lookup tables.

Note that bundling tools such as browserify and webpack that compile multiple modules into single files are less restrictive w.r.t. module specifiers than browsers, because they operate at compile time, not at runtime.

## 24.9 Syntactic pitfall: importing is not destructuring

Both importing and destructuring look similar:

```
import {foo} from './bar.js'; // import
const {foo} = require('./bar.js'); // destructuring
```

But they are quite different:

- You can destructure again inside a destructuring pattern, but the `{}` in an import statement can't be nested.
- Imports don't lose the connection to their exports.
- The syntax for renaming is different:

```
import {foo as f} from './bar.js'; // importing
const {foo: f} = require('./bar.js'); // destructuring
```

Rationale: Destructuring is reminiscent of an object literal (incl. nesting), while importing evokes the idea of renaming.

## 24.10 Preview: loading modules dynamically

So far, the only way to import a module has been via an `import` statement. And with those statements, the module specifier is always fixed. That is, you can't change what you import depending on a condition, you can't assemble a specifier from parts and you can't read it from a file.

An upcoming JavaScript feature changes that: The `import()` operator is used like an asynchronous function.

Consider the following files:

```
lib/my-math.js
main1.js
main2.js
```

We have already seen module `my-math.js`:

```
let notExported = 'abc';
export function square(x) {
  return x * x;
}
export const MY_CONSTANT = 123;
```

This is what using `import()` looks like:

```
const dir = './lib/';
const moduleSpecifier = dir + 'my-math.js';

function loadConstant() {
  return import(moduleSpecifier)
    .then(myMath => {
      const result = myMath.MY_CONSTANT;
      assert.equal(result, 123);
      return result;
    });
}
```

Method `.then()` is part of *Promises*, a mechanism for handling asynchronous results, which is covered later in this book.

Two things in this code weren't possible before:

- We are importing inside a function. `import` statements can only appear at the top level of a module.
- The module specifier comes from a variable.

Next, we'll implement the exact same functionality, but via a so-called *async function*, which provides nicer syntax for Promises.

```
const dir = './lib/';
const moduleSpecifier = dir + 'my-math.js';

async function loadConstant() {
  const myMath = await import(moduleSpecifier);
  const result = myMath.MY_CONSTANT;
  assert.equal(result, 123);
  return result;
}
```

Alas, `import()` isn't part of JavaScript, but probably will be, relatively soon. That means that support is mixed and may be inconsistent.

## 24.11 Further reading

- More on `import()`: “ES proposal: `import()` – dynamically importing ES modules<sup>2</sup>” on 2ality.
- For an in-depth look at ECMAScript modules, consult “Exploring ES6<sup>3</sup>”.



Quiz

See quiz app.

---

<sup>2</sup><http://2ality.com/2017/01/import-operator.html>

<sup>3</sup>[http://exploringjs.com/es6/ch\\_modules.html](http://exploringjs.com/es6/ch_modules.html)





## Chapter 25

# Single objects

### Contents

---

<b>25.1 The two roles of objects in JavaScript</b>	<b>218</b>
<b>25.2 Objects as records</b>	<b>219</b>
25.2.1 Object literals: properties	219
25.2.2 Object literals: property value shorthands	219
25.2.3 Terminology: property keys, property names, property symbols	219
25.2.4 Getting properties	220
25.2.5 Setting properties	220
25.2.6 Object literals: methods	220
25.2.7 Object literals: accessors	220
<b>25.3 Spreading into object literals (...)</b>	<b>221</b>
25.3.1 Use case for spreading: copying objects	222
25.3.2 Use case for spreading: default values for missing properties	222
25.3.3 Use case for spreading: non-destructively changing properties	223
<b>25.4 Methods</b>	<b>223</b>
25.4.1 Methods are properties whose values are functions	223
25.4.2 <code>.call()</code> : explicit parameter <code>this</code>	224
25.4.3 <code>.bind()</code> : pre-filling <code>this</code> and parameters of functions	225
25.4.4 <code>this</code> pitfall: extracting methods	226
25.4.5 <code>this</code> pitfall: accidentally shadowing <code>this</code>	227
25.4.6 The value of <code>this</code> in various contexts	227
25.4.7 Tips for using <code>this</code>	228
<b>25.5 Objects as dictionaries</b>	<b>228</b>
25.5.1 Arbitrary fixed strings as property keys	229
25.5.2 Computed property keys	230
25.5.3 The <code>in</code> operator: is there a property with a given key?	230
25.5.4 Deleting properties	231
25.5.5 Dictionary pitfalls	231
25.5.6 Listing property keys	232
25.5.7 Listing property values	233

25.5.8

Listing property entries

233

25.5.9

Properties are listed deterministically

234

25.6

Standard methods

234

25.7

Advanced topics

235

25.7.1

Object.assign()

235

25.7.2

Freezing objects

235

25.7.3

Property attributes and property descriptors

236

In this book, JavaScript’s style of object-oriented programming (OOP) is introduced in four steps. This chapter covers step 1, the next chapter covers steps 2–4. The steps are (fig. 25.1):

1. **Single objects:** How do *objects*, JavaScript’s basic OOP building blocks, work in isolation?

2. **Prototype chains:** Each object has a chain of zero or more *prototype objects*. Prototypes are JavaScript’s core inheritance mechanism.

3. **Classes:** JavaScript’s *classes* are factories for objects. The relationship between a class and its instances is based on prototypal inheritance.

4. **Subclassing:** The relationship between a *subclass* and its *superclass* is also based on prototypal inheritance.

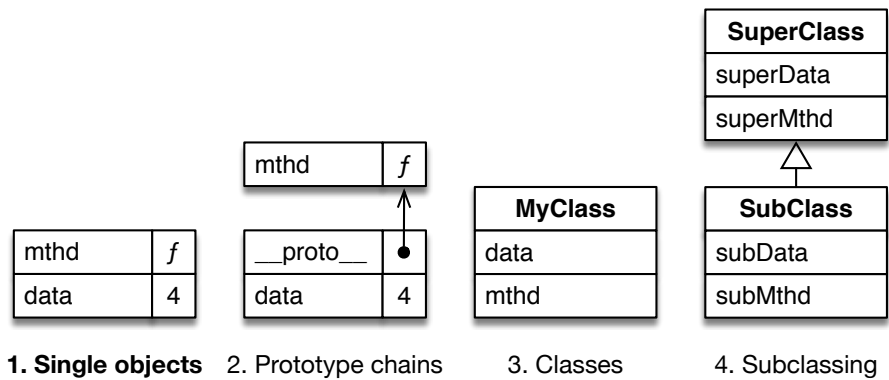


Figure 25.1: This book introduces object-oriented programming in JavaScript in four steps.

## 25.1 The two roles of objects in JavaScript

Objects play two roles in JavaScript:

- **Records:** Objects-as-records have a fixed number of properties, whose keys are known at development time. Their values can have different types. This way of using objects is covered first in this chapter.
- **Dictionaries:** Objects-as-dictionaries have a variable number of properties, whose keys are not known at development time. All of their values have the same type. It is usually better to use Maps as dictionaries than objects (which is covered later in this chapter).

## 25.2 Objects as records

### 25.2.1 Object literals: properties

Objects as records are created via so-called *object literals*. Object literals are a stand-out feature of JavaScript: they allow you to directly create objects. No need for classes! This is an example:

```
const jane = {  
  first: 'Jane',  
  last: 'Doe', // optional trailing comma  
};
```

In this code, we create an object via an object literal, which starts with a curly brace and ends with a curly brace: { ... }. Inside it, we define two properties:

- The first property has the key `first` and the value `'Jane'`.
- The second property has the key `last` and the value `'Doe'`.

If they are written this way, property keys must follow the rules of JavaScript variable names, with the exception that reserved words are allowed.

The properties are accessed as follows:

```
assert.equal(jane.first, 'Jane'); // get property .first  
jane.first = 'John'; // set property .first  
assert.equal(jane.first, 'John');
```

### 25.2.2 Object literals: property value shorthands

Whenever the value of a property is defined via a variable name and that name is the same as the key, you can omit the key.

```
const x = 4;  
const y = 1;  
  
assert.deepEqual(  
  { x, y },  
  { x: x, y: y }  
);
```

### 25.2.3 Terminology: property keys, property names, property symbols

Given that property keys can be strings and symbols, the following distinction is made:

- If a property key is a string, it is also called a *property name*.
- If a property key is a symbol, it is also called a *property symbol*.

This terminology is used in the JavaScript standard library (“own” means “not inherited” and is explained in the next chapter):

- `Object.keys(obj)`: returns all property keys of `obj`
- `Object.getOwnPropertyNames(obj)`

- `Object.getOwnPropertySymbols(obj)`

### 25.2.4 Getting properties

This is how you *get* (read) a property:

```
obj.propKey
```

If `obj` does not have a property whose key is `propKey`, this expression evaluates to `undefined`:

```
const obj = {};
assert.equal(obj.propKey, undefined);
```

### 25.2.5 Setting properties

This is how you *set* (write to) a property:

```
obj.propKey = value;
```

If `obj` already has a property whose key is `propKey`, this statement changes that property. Otherwise, it creates a new property:

```
const obj = {};
assert.deepEqual(
  Object.keys(obj), []);

obj.propKey = 123;
assert.deepEqual(
  Object.keys(obj), ['propKey']);
```

### 25.2.6 Object literals: methods

The following code shows how to create the method `.describe()` via an object literal:

```
const jane = {
  first: 'Jane', // data property
  says(text) { // method
    return `${this.first} says "${text}"`; // (A)
  }, // comma as separator (optional at end)
};
assert.equal(jane.says('hello'), 'Jane says "hello"');
```

During the method call `jane.says('hello')`, `jane` is called the *receiver* of the method call and assigned to the special variable `this`. That enables method `.says()` to access the sibling property `.first` in line A.

### 25.2.7 Object literals: accessors

There are two kinds of accessors in JavaScript:

- A *getter* is a method that is invoked by *getting* (reading) a property.
- A *setter* is a method that is invoked by *setting* (writing) a property.

#### 25.2.7.1 Getters

A getter is created by prefixing a method definition with the keyword `get`:

```
const jane = {
  first: 'Jane',
  last: 'Doe',
  get full() {
    return `${this.first} ${this.last}`;
  },
};
```

```
assert.equal(jane.full, 'Jane Doe');
jane.first = 'John';
assert.equal(jane.full, 'John Doe');
```

#### 25.2.7.2 Setters

A setter is created by prefixing a method definition with the keyword `set`:

```
const jane = {
  first: 'Jane',
  last: 'Doe',
  set full(fullName) {
    const parts = fullName.split(' ');
    this.first = parts[0];
    this.last = parts[1];
  },
};
```

```
jane.full = 'Richard Roe';
assert.equal(jane.first, 'Richard');
assert.equal(jane.last, 'Roe');
```



Exercise: Creating an object via an object literal

`exercises/single-objects/color_point_object_test.js`

## 25.3 Spreading into object literals (...)

We have already seen spreading (...) being used in function calls, where it turns the contents of an iterable into arguments.

Inside an object literal, a *spread property* adds the properties of another object to the current one:

```
> const obj = {foo: 1, bar: 2};
> {...obj, baz: 3}
{ foo: 1, bar: 2, baz: 3 }
```

If property keys clash, the property that is mentioned last “wins”:

```
> const obj = {foo: 1, bar: 2, baz: 3};
> {...obj, foo: true}
{ foo: true, bar: 2, baz: 3 }
> {foo: true, ...obj}
{ foo: 1, bar: 2, baz: 3 }
```

### 25.3.1 Use case for spreading: copying objects

You can use spread to create a copy of an object:

```
const copy = {...obj};
```

Caveat – the copy is shallow:

```
const original = { a: 1, b: {foo: true} };
const copy = {...original};

// The first level is a true copy:
assert.deepEqual(
  copy, { a: 1, b: {foo: true} });
original.a = 2;
assert.deepEqual(
  copy, { a: 1, b: {foo: true} }); // no change

// Deeper levels are not copied:
original.b.foo = false;
// The value of property `b` is shared
// between original and copy.
assert.deepEqual(
  copy, { a: 1, b: {foo: false} });
```

### 25.3.2 Use case for spreading: default values for missing properties

If one of the inputs of your code is an object with data, you can make properties optional if you specify default values for them. One technique for doing so is via an object whose properties contain the default values. In the following example, that object is `DEFAULTS`:

```
const DEFAULTS = {foo: 'a', bar: 'b'};
const providedData = {foo: 1};

const allData = {...DEFAULTS, ...providedData};
assert.deepEqual(allData, {foo: 1, bar: 'b'});
```

The result, the object `allData`, is created by creating a copy of `DEFAULTS` and overriding its properties with those of `providedData`.

But you don't need an object to specify the default values, you can also specify them inside the object literal, individually:

```
const providedData = {foo: 1};

const allData = {foo: 'a', bar: 'b', ...providedData};
assert.deepEqual(allData, {foo: 1, bar: 'b'});
```

### 25.3.3 Use case for spreading: non-destructively changing properties

So far, we have encountered one way of changing a property of an object: We *set* it and mutate the object. That is, this way of changing a property is *destructive*

With spreading, you can change a property *non-destructively*: You make a copy of the object where the property has a different value.

For example, this code non-destructively updates property `.foo`:

```
const obj = {foo: 'a', bar: 'b'};
const updatedObj = {...obj, foo: 1};
assert.deepEqual(updatedObj, {foo: 1, bar: 'b'});
```



Exercise: Non-destructively updating properties via spreading (fixed key)

`exercises/single-objects/update_name_test.js`

## 25.4 Methods

### 25.4.1 Methods are properties whose values are functions

Let's revisit the example that was used to introduce methods:

```
const jane = {
  first: 'Jane',
  says(text) {
    return `${this.first} says "${text}"`;
  },
};
```

Somewhat surprisingly, methods are functions:

```
assert.equal(typeof jane.says, 'function');
```

Why is that? Remember that, in [the chapter on callable entities](#), we learned that ordinary functions play several roles. *Method* is one of those roles. Therefore, under the hood, `jane` roughly looks as follows.

```
const jane = {
  first: 'Jane',
  says: function (text) {
    return `${this.first} says "${text}"`;
  },
};
```

```
    },
  };

```

### 25.4.2 `.call()`: explicit parameter `this`

Remember that each function `someFunc` is also an object and therefore has methods. One such method is `.call()` – it lets you call functions while specifying `this` explicitly:

```
someFunc.call(thisValue, arg1, arg2, arg3);
```

#### 25.4.2.1 Methods and `.call()`

If you make a method call, `this` is always an implicit parameter:

```
const obj = {
  method(x) {
    assert.equal(this, obj); // implicit parameter
    assert.equal(x, 'a');
  },
};

obj.method('a');
// Equivalent:
obj.method.call(obj, 'a');
```

As an aside, that means that there are actually two different dot operators:

1. One for accessing properties: `obj.prop`
2. One for making method calls: `obj.prop()`

They are different in that (2) is not just (1), followed by the function call operator `()`. Instead, (2) additionally specifies a value for `this` (as shown in the previous example).

#### 25.4.2.2 Functions and `.call()`

However, `this` is also an implicit parameter if you function-call an ordinary function:

```
function func(x) {
  assert.equal(this, undefined); // implicit parameter
  assert.equal(x, 'a');
}

func('a');
// Equivalent:
func.call(undefined, 'a');
```

That is, during a function call, ordinary functions have a `this`, but it is set to `undefined`, which indicates that it doesn't really have a purpose here.

Next, we'll examine the pitfalls of using `this`. Before we can do that, we need one more tool: the method `.bind()` of functions.



### 25.4.3 .bind(): pre-filling this and parameters of functions

.bind() is another method of function objects. This method is invoked as follows.

```
const boundFunc = someFunc.bind(thisValue, arg1, arg2, arg3);
```

.bind() returns a new function boundFunc(). Calling that function invokes someFunc() with this set to thisValue and these parameters: arg1, arg2, arg3, followed by the parameters of boundFunc().

That is, the following two function calls are equivalent:

```
boundFunc('a', 'b')
someFunc.call(thisValue, arg1, arg2, arg3, 'a', 'b')
```

Another way of pre-filling this and parameters, is via an arrow function:

```
const boundFunc2 = (...args) =>
  someFunc.call(thisValue, arg1, arg2, arg3, ...args);
```

Therefore, .bind() can be implemented as a real function as follows:

```
function bind(func, thisValue, ...boundArgs) {
  return (...args) =>
    func.call(thisValue, ...boundArgs, ...args);
}
```

#### 25.4.3.1 Example: binding a real function

Using .bind() for real functions is somewhat unintuitive, because you have to provide a value for this. That value is usually undefined, mirroring what happens during function calls.

In the following example, we create add8(), a function that has one parameter, by binding the first parameter of add() to 8.

```
function add(x, y) {
  return x + y;
}

const add8 = add.bind(undefined, 8);
assert.equal(add8(1), 9);
```

#### 25.4.3.2 Example: binding a method

In the following code, we turn method .says() into the stand-alone function func():

```
const jane = {
  first: 'Jane',
  says(text) {
    return `${this.first} says "${text}"`; // (A)
  },
};
```

```
const func = jane.says.bind(jane, 'hello');
assert.equal(func(), 'Jane says "hello"');
```

Setting this to `jane` via `.bind()` is crucial here. Otherwise, `func()` wouldn't work properly, because `this` is used in line A.

#### 25.4.4 `this` pitfall: extracting methods

We now know quite a bit about functions and methods and are ready to take a look at the biggest pitfall involving methods and `this`: function-calling a method extracted from an object can fail if you are not careful.

In the following example, we fail when we extract method `jane.says()`, store it in the variable `func` and function-call `func()`.

```
const jane = {
  first: 'Jane',
  says(text) {
    return `${this.first} says "${text}"`;
  },
};
const func = jane.says; // extract the method
assert.throws(
  () => func('hello'), // (A)
  {
    name: 'TypeError',
    message: "Cannot read property 'first' of undefined",
  });
```

The function call in line A is equivalent to:

```
assert.throws(
  () => jane.says.call(undefined, 'hello'), // `this` is undefined!
  {
    name: 'TypeError',
    message: "Cannot read property 'first' of undefined",
  });
```

So how do we fix this? We need to use `.bind()` to extract method `.says()`:

```
const func2 = jane.says.bind(jane);
assert.equal(func2('hello'), 'Jane says "hello"');
```

The `.bind()` ensures that `this` is always `jane` when we call `func()`.

You can also use arrow functions to extract methods:

```
const func3 = text => jane.says(text);
assert.equal(func3('hello'), 'Jane says "hello"');
```



Exercise: Extracting a method

```
exercises/single-objects/method_extraction_exrc.js
```

### 25.4.5 this pitfall: accidentally shadowing this

Accidentally shadowing `this` is only an issue if you use ordinary functions.

Consider the following problem: When you are inside an ordinary function, you can't access the `this` of the surrounding scope, because the ordinary function has its own `this`. In other words: a variable in an inner scope hides a variable in an outer scope. That is called *shadowing*. The following code is an example:

```
const obj = {
  name: 'Jane',
  sayHiTo(friends) {
    return friends.map(
      function (friend) { // (A)
        return `${this.name} says hi to ${friend}`; // (B)
      });
  }
};
assert.throws(
  () => obj.sayHiTo(['Tarzan', 'Cheeta']),
  {
    name: 'TypeError',
    message: "Cannot read property 'name' of undefined",
  });
```

Why the error? The `this` in line B isn't the `this` of `.sayHiTo()`, it is the `this` of the ordinary function starting in line B.

There are several ways to fix this. The easiest is to use an arrow function – which doesn't have its own `this`, so shadowing is not an issue.

```
const obj = {
  name: 'Jane',
  sayHiTo(friends) {
    return friends.map(
      (friend) => {
        return `${this.name} says hi to ${friend}`;
      });
  }
};
assert.deepEqual(
  obj.sayHiTo(['Tarzan', 'Cheeta']),
  ['Jane says hi to Tarzan', 'Jane says hi to Cheeta']);
```

### 25.4.6 The value of this in various contexts

What is the value of `this` in various contexts?

Inside a callable entity, the value of `this` depends on how the callable entity is invoked and what kind of callable entity it is:

- Function call:
  - Ordinary functions: `this === undefined`
  - Arrow functions: `this` is same as in surrounding scope (lexical `this`)
- Method call: `this` is receiver of call
- `new`: `this` refers to newly created instance

You can also access `this` in all common top-level scopes:

- `<script>` element: `this === window`
- ES modules: `this === undefined`
- CommonJS modules: `this === module.exports`

However, I like to pretend that you can't access `this` in top-level scopes, because top-level `this` is confusing and not that useful.

### 25.4.7 Tips for using `this`

We have seen two big `this`-related pitfalls:

1. **Extracting methods**
2. **Accidentally shadowing `this`**

One simple rule helps avoid the second pitfall:

“Avoid the keyword `function`”: Never use ordinary functions, only arrow functions (for real functions) and method definitions.

Let's break down this rule:

- If all real functions are arrow functions, the second pitfall can never occur.
- Using method definitions means that you'll only see `this` inside methods, which makes this feature less confusing.

However, even though I don't use (ordinary) function *expressions*, anymore, I do like function *declarations*: They look nice and hoisting is convenient. You can use them safely if you don't refer to `this` inside them. The checking tool ESLint has a rule<sup>1</sup> that helps with that.

Alas, there is no simple way around the first pitfall: Whenever you extract a method, you have to be careful and do it properly. For example, by binding `this`.

## 25.5 Objects as dictionaries

Objects work best as records. But before ES6, JavaScript did not have a data structure for dictionaries (ES6 brought Maps). Therefore, objects had to be used as dictionaries. As a consequence, keys had to be strings, but values could have arbitrary types.

---

<sup>1</sup><https://eslint.org/docs/rules/no-invalid-this>

We first look at features of objects that are related to dictionaries, but also occasionally useful for objects-as-records. This section concludes with tips for actually using objects as dictionaries (spoiler: avoid, use Maps if you can).

### 25.5.1 Arbitrary fixed strings as property keys

When going from objects-as-records to objects-as-dictionaries, one important change is that we must be able to use arbitrary strings as property keys. This subsection explains how to achieve that for fixed string keys. The next subsection explains how to dynamically compute arbitrary keys.

So far, we have only seen legal JavaScript identifiers as property keys (with the exception of symbols):

```
const obj = {
  mustBeAnIdentifier: 123,
};

// Get property
assert.equal(obj.mustBeAnIdentifier, 123);

// Set property
obj.mustBeAnIdentifier = 'abc';
assert.equal(obj.mustBeAnIdentifier, 'abc');
```

Two techniques allow us to use arbitrary strings as property keys.

First – when creating property keys via object literals, we can quote property keys (with single or double quotes):

```
const obj = {
  'Can be any string!': 123,
};
```

Second – when getting or setting properties, we can use square brackets with strings inside them:

```
// Get property
assert.equal(obj['Can be any string!'], 123);

// Set property
obj['Can be any string!'] = 'abc';
assert.equal(obj['Can be any string!'], 'abc');
```

You can also quote the keys of methods:

```
const obj = {
  'A nice method'() {
    return 'Yes!';
  },
};

assert.equal(obj['A nice method'](), 'Yes!');
```

### 25.5.2 Computed property keys

So far, we were limited by what we could do with property keys inside object literals: They were always fixed and they were always strings. We can dynamically compute arbitrary keys if we put expressions in square brackets:

```
const obj = {
  ['Hello world!']: true,
  ['f'+ 'o'+ 'o']: 123,
  [Symbol.toStringTag]: 'Goodbye', // (A)
};

assert.equal(obj['Hello world!'], true);
assert.equal(obj.foo, 123);
assert.equal(obj[Symbol.toStringTag], 'Goodbye');
```

The main use case for computed keys is having symbols as property keys (line A).

Note that the square brackets operator for getting and setting properties works with arbitrary expressions:

```
assert.equal(obj['f'+ 'o'+ 'o'], 123);
assert.equal(obj['==> foo'.slice(-3)], 123);
```

Methods can have computed property keys, too:

```
const methodKey = Symbol();
const obj = {
  [methodKey]() {
    return 'Yes!';
  },
};

assert.equal(obj[methodKey](), 'Yes!');
```

We are now switching back to fixed property keys, but you can always use square brackets if you need computed property keys.



**Exercise: Non-destructively updating properties via spreading (computed key)**

`exercises/single-objects/update_property_test.js`

### 25.5.3 The `in` operator: is there a property with a given key?

The `in` operator checks if an object has a property with a given key:

```
const obj = {
  foo: 'abc',
  bar: undefined,
};
```

```
assert.equal('foo' in obj, true);
assert.equal('unknownKey' in obj, false);
```

You can also use a truthiness check to determine if a property exists:

```
assert.equal(
  obj.unknownKey ? 'exists' : 'does not exist',
  'does not exist');
assert.equal(
  obj.foo ? 'exists' : 'does not exist',
  'exists');
```

The previous check works, because reading a non-existent property returns undefined, which is falsy. And because obj.foo is truthy.

There is, however, one important caveat: Truthiness checks fail if the property exists, but has a falsy value (undefined, null, false, 0, "", etc.):

```
assert.equal(
  obj.bar ? 'exists' : 'does not exist',
  'does not exist'); // alas: not 'exists'
```

### 25.5.4 Deleting properties

You can delete properties via the delete operator:

```
const obj = {
  foo: 123,
};
assert.deepEqual(Object.keys(obj), ['foo']);

delete obj.foo;

assert.deepEqual(Object.keys(obj), []);
```

### 25.5.5 Dictionary pitfalls

If you use plain objects (created via object literals) as dictionaries, you have to look out for two pitfalls.

The first pitfall is that the in operator also finds inherited properties:

```
const dict = {};
assert.equal('toString' in dict, true);
```

We want dict to be treated as empty, but the in operator detects the properties it inherits from its prototype, Object.prototype.

The second pitfall is that you can't use the property key \_\_proto\_\_ because it has special powers (it sets the prototype of the object):

```
const dict = {};

dict['__proto__'] = 123;
```

```
// No property was added to dict:
assert.deepEqual(Object.keys(dict), []);
```

So how do we navigate around these pitfalls?

- Whenever you can, use Maps. They are the best solution for dictionaries.
- If you can't: use a library for objects-as-dictionaries that does everything safely.
- If you can't: use an object without a prototype. That eliminates the two pitfalls in modern JavaScript.

```
const dict = Object.create(null); // no prototype

assert.equal('toString' in dict, false);

dict['__proto__'] = 123;
assert.deepEqual(Object.keys(dict), ['__proto__']);
```



#### Exercise: Using an object as a dictionary

`exercises/single-objects/simple_dict_test.js`

### 25.5.6 Listing property keys

Table 25.1: Standard library methods for listing *own* (non-inherited) property keys. All of them return Arrays with strings and/or symbols.

	enumerable	non-e.	string	symbol
<code>Object.keys()</code>	✓		✓	
<code>Object.getOwnPropertyNames()</code>	✓	✓	✓	
<code>Object.getOwnPropertySymbols()</code>	✓	✓		✓
<code>Reflect.ownKeys()</code>	✓	✓	✓	✓

Each of the methods in [tbl. 25.1](#) returns an Array with the own property keys of the parameter. In the names of the methods you can see the distinction between property keys (strings and symbols), property names (only strings) and property symbols (only symbols) that we discussed previously.

Enumerability is an *attribute* of properties. By default, properties are enumerable, but there are ways to change that (shown in the next example and [described in slightly more detail later](#)).

For example:

```
const enumerableSymbolKey = Symbol('enumerableSymbolKey');
const nonEnumSymbolKey = Symbol('nonEnumSymbolKey');

// We create the enumerable properties via an object literal
const obj = {
  enumerableStringKey: 1,
  [enumerableSymbolKey]: 2,
```



```

}

// For the non-enumerable properties,
// we need a more powerful tool:
Object.defineProperty(obj, {
  nonEnumStringKey: {
    value: 3,
    enumerable: false,
  },
  [nonEnumSymbolKey]: {
    value: 4,
    enumerable: false,
  },
});

assert.deepEqual(
  Object.keys(obj),
  [ 'enumerableStringKey' ] );
assert.deepEqual(
  Object.getOwnPropertyNames(obj),
  [ 'enumerableStringKey', 'nonEnumStringKey' ] );
assert.deepEqual(
  Object.getOwnPropertySymbols(obj),
  [ enumerableSymbolKey, nonEnumSymbolKey ] );
assert.deepEqual(
  Reflect.ownKeys(obj),
  [ 'enumerableStringKey',
    'nonEnumStringKey',
    enumerableSymbolKey,
    nonEnumSymbolKey ] );

```

### 25.5.7 Listing property values

`Object.values()` lists the values of all enumerable properties of an object:

```

const obj = {foo: 1, bar: 2};
assert.deepEqual(
  Object.values(obj),
  [1, 2]);

```

### 25.5.8 Listing property entries

`Object.entries()` lists key-value pairs of enumerable properties. Each pair is encoded as a two-element Array:

```

const obj = {foo: 1, bar: 2};
assert.deepEqual(
  Object.entries(obj),

```

```
[
  ['foo', 1],
  ['bar', 2],
];
```

### 25.5.9 Properties are listed deterministically

Own (non-inherited) properties of objects are always listed in the following order:

1. Properties with integer indices (e.g. Array indices)
  - In ascending numeric order
2. Remaining properties with string keys
  - In the order in which they were added
3. Properties with symbol keys
  - In the order in which they were added

The following example demonstrates how property keys are sorted according to these rules:

```
> Object.keys({b:0,a:0, 2:0,1:0})
[ '1', '2', 'b', 'a' ]
```

(You can look up the details in the spec.<sup>2</sup>)

## 25.6 Standard methods

`Object.prototype` defines several standard methods that can be overridden. Two important ones are:

- `.toString()`
- `.valueOf()`

Roughly, `.toString()` configures how objects are converted to strings:

```
> String({toString() { return 'Hello!' }})
'Hello!'
> String({})
'[object Object]'
```

And `.valueOf()` configures how objects are converted to numbers:

```
> Number({valueOf() { return 123 }})
123
> Number({})
NaN
```

---

<sup>2</sup><https://tc39.github.io/ecma262/#sec-ordinaryownpropertykeys>

## 25.7 Advanced topics

The following subsections give a brief overview of topics that are beyond the scope of this book.

### 25.7.1 `Object.assign()`

`Object.assign()` is a tool method:

```
Object.assign(target, source_1, source_2, ...)
```

This expression (destructively) merges `source_1` into `target`, then `source_2` etc. At the end, it returns `target`. For example:

```
const target = { foo: 1 };
const result = Object.assign(
  target,
  {bar: 2},
  {baz: 3, bar: 4});
assert.deepEqual(
  result, { foo: 1, bar: 4, baz: 3 });
// target was changed!
assert.deepEqual(target, result);
```

The use cases for `Object.assign()` are similar to those for spread properties. In a way, it spreads destructively.

For more information on `Object.assign()`, consult “Exploring ES6”<sup>3</sup>.

### 25.7.2 Freezing objects

`Object.freeze(obj)` makes `obj` immutable: You can’t change or add properties or change the prototype of `obj`.

For example:

```
const frozen = Object.freeze({ x: 2, y: 5 });
assert.throws(
  () => { frozen.x = 7 },
  {
    name: 'TypeError',
    message: /^Cannot assign to read only property 'x'/,
  });
```

There is one caveat: `Object.freeze(obj)` freezes shallowly. That is, only the properties of `obj` are frozen, but not objects stored in properties.

For more information on `Object.freeze()`, consult “Speaking JavaScript”<sup>4</sup>.

---

<sup>3</sup>[http://exploringjs.com/es6/ch\\_oop-besides-classes.html#Object\\_assign](http://exploringjs.com/es6/ch_oop-besides-classes.html#Object_assign)

<sup>4</sup>[http://speakingjs.com/es5/ch17.html#freezing\\_objects](http://speakingjs.com/es5/ch17.html#freezing_objects)

### 25.7.3 Property attributes and property descriptors

Just as objects are composed of properties, properties are composed of *attributes*. That is, you can configure more than just the value of a property – which is just one of several attributes. Other attributes include:

- **writable**: Is it possible to change the value of the property?
- **enumerable**: Is the property listed by `Object.keys()`?

When you are using one of the operations for accessing property attributes, attributes are specified via *property descriptors*: objects where each property represents one attribute. For example, this is how you read the attributes of a property `obj.foo`:

```
const obj = { foo: 123 };
assert.deepEqual(
  Object.getOwnPropertyDescriptor(obj, 'foo'),
  {
    value: 123,
    writable: true,
    enumerable: true,
    configurable: true,
  });
```

And this is how you set the attributes of a property `obj.bar`:

```
const obj = {
  foo: 1,
  bar: 2,
};

assert.deepEqual(Object.keys(obj), ['foo', 'bar']);

// Hide property `bar` from Object.keys()
Object.defineProperty(obj, 'bar', {
  enumerable: false,
});

assert.deepEqual(Object.keys(obj), ['foo']);
```

For more on property attributes and property descriptors, consult “Speaking JavaScript<sup>5</sup>”.



#### Quiz

See quiz app.

---

<sup>5</sup>[http://speakingjs.com/es5/ch17.html#property\\_attributes](http://speakingjs.com/es5/ch17.html#property_attributes)

## Chapter 26

# Prototype chains and classes

### Contents

---

<b>26.1 Prototype chains</b>	<b>238</b>
26.1.1 Pitfall: only first member of prototype chain is mutated	239
26.1.2 Tips for working with prototypes (advanced)	239
26.1.3 Sharing data via prototypes	241
26.1.4 Dispatched vs. direct method calls (advanced)	242
<b>26.2 Classes</b>	<b>243</b>
26.2.1 A class for persons	243
26.2.2 Class expressions	243
26.2.3 Classes under the hood (advanced)	244
26.2.4 Class definitions: prototype properties	245
26.2.5 Class definitions: static properties	246
26.2.6 The instanceof operator	246
26.2.7 Why I recommend classes	246
<b>26.3 Private data for classes</b>	<b>247</b>
26.3.1 Private data: naming convention	247
26.3.2 Private data: WeakMaps	247
26.3.3 More techniques for private data	248
<b>26.4 Subclassing</b>	<b>248</b>
26.4.1 Subclasses under the hood (advanced)	249
26.4.2 instanceof in more detail (advanced)	250
26.4.3 Prototype chains of built-in objects (advanced)	251
26.4.4 Mixin classes (advanced)	252

---

In this book, JavaScript's style of object-oriented programming (OOP) is introduced in four steps. This chapter covers steps 2–4, [the previous chapter](#) covers step 1. The steps are (fig. 26.1):

1. Single objects: How do *objects*, JavaScript's basic OOP building blocks, work in isolation?
2. **Prototype chains:** Each object has a chain of zero or more *prototype objects*. Prototypes are JavaScript's core inheritance mechanism.

3. **Classes:** JavaScript's *classes* are factories for objects. The relationship between a class and its instances is based on prototypical inheritance.
4. **Subclassing:** The relationship between a *subclass* and its *superclass* is also based on prototypical inheritance.

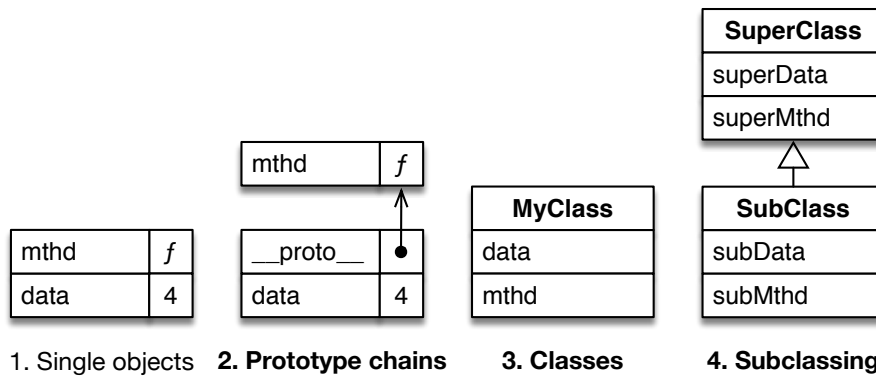


Figure 26.1: This book introduces object-oriented programming in JavaScript in four steps.

## 26.1 Prototype chains

Prototypes are JavaScript's only inheritance mechanism: Each object has a prototype that is either `null` or an object. In the latter case, the object inherits all of the prototype's properties.

In an object literal, you can set the prototype via the special property `__proto__`:

```
const proto = {
  protoProp: 'a',
};
const obj = {
  __proto__: proto,
  objProp: 'b',
};

// obj inherits .protoProp:
assert.equal(obj.protoProp, 'a');
assert.equal('protoProp' in obj, true);
```

Given that a prototype object can have a prototype itself, we get a chain of objects – the so-called *prototype chain*. That means that inheritance gives us the impression that we are dealing with single objects, but we are actually dealing with chains of objects.

Fig. 26.2 shows what the prototype chain of `obj` looks like.

Non-inherited properties are called *own properties*. `obj` has one own property, `.objProp`.

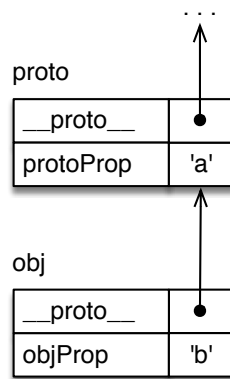


Figure 26.2: `obj` starts a chain of objects that continues with `proto` and other objects.

### 26.1.1 Pitfall: only first member of prototype chain is mutated

One aspect of prototype chains that may be counter-intuitive is that setting *any* property via an object – even an inherited one – only changes that object – never one of the prototypes.

Consider the following object `obj`:

```

const proto = {
  protoProp: 'a',
};
const obj = {
  __proto__: proto,
  objProp: 'b',
};
  
```

When we set the inherited property `obj.protoProp` in line A, we “change” it by creating an own property: When reading `obj.protoProp`, the own property is found first and its value overrides the value of the inherited property.

```

assert.deepEqual(Object.keys(obj), ['objProp']);

obj.protoProp = 'x'; // (A)

// We created a new own property:
assert.deepEqual(Object.keys(obj), ['objProp', 'protoProp']);

// The inherited property itself is unchanged:
assert.equal(proto.protoProp, 'a');
  
```

The prototype chain of `obj` is depicted in fig. 26.3.

### 26.1.2 Tips for working with prototypes (advanced)

These are a few things to keep in mind when working with prototypes:

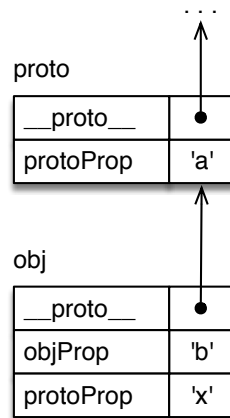


Figure 26.3: The own property `.protoProp` of `obj` overrides the property inherited from `proto`.

- Avoid the special property `__proto__`. It is implemented via a getter and a setter in `Object.prototype` and may therefore be switched off, by creating an object that doesn't have `Object.prototype` in its prototype chain.
  - However, `__proto__` in object literals is different. It is a built-in feature there and you can always use it safely.
- The best way to set a prototype is when creating an object. E.g. via:
 

```
Object.create(proto: Object) : Object
```

 If you have to, you can also use `Object.setPrototypeOf()` to change the prototype of an existing object.
- The best way to get a prototype is via the following method:
 

```
Object.getPrototypeOf(obj: Object) : Object
```
- There is also a method for checking if `o1` is in the prototype chain of `o2` that is provided via `Object.prototype`:
 

```
> Array.prototype.isPrototypeOf([])
true
```

The following code demonstrates some of the mentioned features:

```

const proto1 = {};
const proto2 = {};

const obj = Object.create(proto1);
assert.equal(Object.getPrototypeOf(obj), proto1);
assert.equal(proto1.isPrototypeOf(obj), true);

Object.setPrototypeOf(obj, proto2);
assert.equal(proto2.isPrototypeOf(obj), true);
assert.equal(proto1.isPrototypeOf(obj), false);
  
```



### 26.1.3 Sharing data via prototypes

Consider the following code:

```
const jane = {
  name: 'Jane',
  describe() {
    return 'Person named '+this.name;
  },
};
const tarzan = {
  name: 'Tarzan',
  describe() {
    return 'Person named '+this.name;
  },
};

assert.equal(jane.describe(), 'Person named Jane');
assert.equal(tarzan.describe(), 'Person named Tarzan');
```

We have two objects that are very similar. Both have two properties whose names are `.name` and `.describe`. Additionally, method `.describe()` is the same. How can we avoid that method being duplicated?

We can move it to a shared prototype, `PersonProto`:

```
const PersonProto = {
  describe() {
    return 'Person named ' + this.name;
  },
};
const jane = {
  __proto__: PersonProto,
  name: 'Jane',
};
const tarzan = {
  __proto__: PersonProto,
  name: 'Tarzan',
};
```

The name of the prototype reflects that both `jane` and `tarzan` are persons.

The diagram in fig. 26.4 illustrates how the three objects are connected: The objects at the bottom now contain the properties that are specific to `jane` and `tarzan`. The object at the top contains the properties that is shared between them.

When you make the method call `jane.describe()`, this points to the receiver of that method call, `jane` (in the bottom left corner of the diagram). That's why the method still works. The analogous thing happens when you call `tarzan.describe()`.

```
assert.equal(jane.describe(), 'Person named Jane');
assert.equal(tarzan.describe(), 'Person named Tarzan');
```

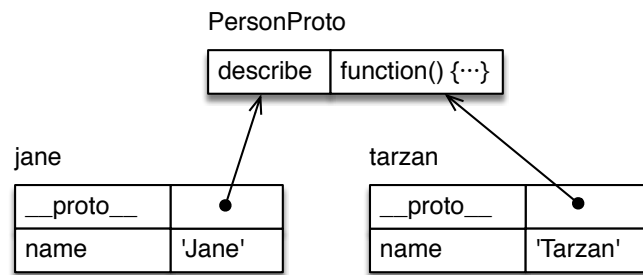


Figure 26.4: Objects `jane` and `tarzan` share method `.describe()`, via their common prototype `PersonProto`.

#### 26.1.4 Dispatched vs. direct method calls (advanced)

Let's examine how method calls work with prototype chains. We revisit `jane` from the previous example:

```

const PersonProto = {
  describe() {
    return 'Person named ' + this.name;
  },
};
const jane = {
  __proto__: PersonProto,
  name: 'Jane',
};
  
```

Fig. 26.5 has a diagram with `jane`'s prototype chain.

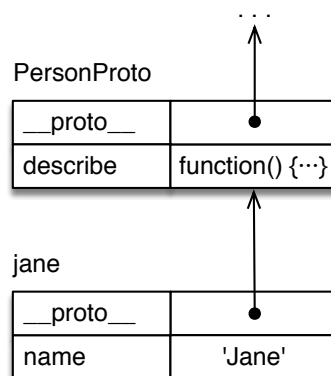


Figure 26.5: The prototype chain of `jane` starts with `jane` and continues with `PersonProto`.

Normal method calls are *dispatched*. To make the method call `jane.describe()`:

- JavaScript first looks for the value of `jane.describe`, by traversing the prototype chain.
- Then it calls the function it found, while setting `this` to `jane`. `this` is the *receiver* of the method call (where the search for property `.describe` started).

This way of dynamically looking for methods, is called *dynamic dispatch*.

You can make the same method call while (mostly) bypassing dispatch:

```
PersonProto.describe.call(jane)
```

This time, `PersonProto.describe` is an own property and there is no need to search the prototypes. We also specify this ourselves, via `.call()`.

Note how this always points to the beginning of a prototype chain. That enables `.describe()` to access `.name`. And it is where the mutations happen (should a method want to set the `.name`).

## 26.2 Classes

We are now ready to take on classes, which are basically a compact syntax for setting up prototype chains. While their foundations may be unconventional, working with JavaScript's classes should still feel familiar – if you have used an object-oriented language before.

### 26.2.1 A class for persons

We have previously worked with `jane` and `tarzan`, single objects representing persons. Let's use a class to implement a factory for persons:

```
class Person {
  constructor(name) {
    this.name = name;
  }
  describe() {
    return 'Person named '+this.name;
  }
}
```

`jane` and `tarzan` can now be created via `new Person()`:

```
const jane = new Person('Jane');
assert.equal(jane.describe(), 'Person named Jane');

const tarzan = new Person('Tarzan');
assert.equal(tarzan.describe(), 'Person named Tarzan');
```

### 26.2.2 Class expressions

The previous class definition was a *class declaration*. There are also *anonymous class expressions*:

```
const Person = class { ... };
```

And *named class expressions*:

```
const Person = class MyClass { ... };
```

### 26.2.3 Classes under the hood (advanced)

There is a lot going on under the hood of classes. Let's look at the diagram for `jane` (fig. 26.6).

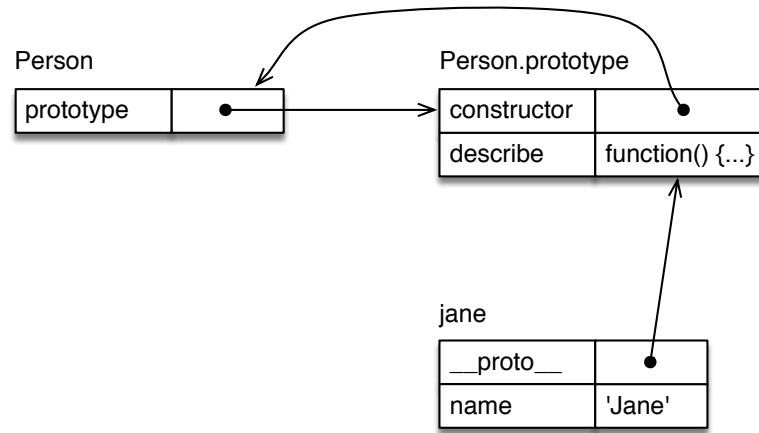


Figure 26.6: The class `Person` has the property `.prototype` that points to an object that is the prototype of all instances of `Person`. `jane` is one such instance.

The main purpose of class `Person` is to set up the prototype chain on the right (`jane`, followed by `Person.prototype`). It is interesting to note that both constructs inside class `Person` (`.constructor` and `.describe()`) created properties for `Person.prototype`, not for `Person`.

The reason for this slightly odd approach is backward compatibility: Prior to classes, *constructor functions* (ordinary functions, invoked via the `new` operator) were often used as factories for objects. Classes are mostly better syntax for constructor functions and therefore remain compatible with old code. That explains why classes are functions:

```
> typeof Person
'function'
```

In this book, I use the terms *constructor* (function) and *class* interchangeably.

Many people confuse `__proto__` and `.prototype`. Hopefully, the diagram in fig. 26.6 makes it clear, how they differ:

- `__proto__` is a special property for accessing the prototype of an object.
- `.prototype` is a normal property that is only special due to how the `new` operator uses it. The name is not ideal: `Person.prototype` does not point to the prototype of `Person`, it points to the prototype of all instances of `Person`.

#### 26.2.3.1 `Person.prototype.constructor`

There is one detail in fig. 26.6 that we haven't look at, yet: `Person.prototype.constructor` points back to `Person`:

```
> Person.prototype.constructor === Person
true
```

This setup is also there for historical reasons. But it also has two benefits.

First, each instance of a class inherits property `.constructor`. Therefore, given an instance, you can make “similar” objects via it:

```
const jane = new Person('Jane');

const cheeta = new jane.constructor('Cheeta');
assert.ok(cheeta instanceof Person);
```

Second, you can get the name of the class that created a given instance:

```
const tarzan = new Person('Tarzan');

assert.equal(tarzan.constructor.name, 'Person');
```

### 26.2.4 Class definitions: prototype properties

The following code demonstrates all parts of a class definition `Foo` that create properties of `Foo.prototype`:

```
class Foo {
  constructor(prop) {
    this.prop = prop;
  }
  protoMethod() {
    return 'protoMethod';
  }
  get protoGetter() {
    return 'protoGetter';
  }
}
```

Let’s examine them in order:

- `.constructor()` is called after creating a new instance of `Foo`, to set up that instance.
- `.protoMethod()` is a normal method. It is stored in `Foo.prototype`.
- `.protoGetter` is a getter that is stored in `Foo.prototype`.

The following interaction uses class `Foo`:

```
> const foo = new Foo(123);
> foo.prop
123

> foo.protoMethod()
'protoMethod'
> foo.protoGetter
'protoGetter'
```

### 26.2.5 Class definitions: static properties

The following code demonstrates all parts of a class definition that create so-called *static properties* – properties of the class itself.

```
class Bar {  
  static staticMethod() {  
    return 'staticMethod';  
  }  
  static get staticGetter() {  
    return 'staticGetter';  
  }  
}
```

The static method and the static getter are used as follows.

```
> Bar.staticMethod()  
'staticMethod'  
> Bar.staticGetter  
'staticGetter'
```

### 26.2.6 The instanceof operator

The instanceof operator tells you if a value is an instance of a given class:

```
> new Person('Jane') instanceof Person  
true  
> ({}) instanceof Person  
false  
> ({}) instanceof Object  
true  
> [] instanceof Array  
true
```

### 26.2.7 Why I recommend classes

I recommend using classes for the following reasons:

- Classes are a common standard for object creation and inheritance that is now widely supported across frameworks (React, Angular, Ember, etc.).
- They help tools such as IDEs and type checkers with their work and enable new features.
- They are a foundation for future features such as value objects, immutable objects, decorators, etc.
- They make it easier for newcomers to get started with JavaScript.
- JavaScript engines optimize them. That is, code that uses classes is usually faster than code that uses a custom inheritance library.

That doesn't mean that classes are perfect. One issue I have with them, is:

- Classes look different from what they are under the hood. In other words, there is a disconnect between syntax and semantics.

It would be nice if classes were (syntax for) constructor *objects* (new-able prototype objects) and not to constructor *functions*. But backward compatibility is a legitimate reason for them being the latter.



#### Exercise: Implementing a class

exercises/proto-chains-classes/point\_class\_test.js

## 26.3 Private data for classes

This section describes techniques for hiding some of the data of an object from the outside. We discuss them in the context of classes, but they also work for objects created directly, via object literals etc.

### 26.3.1 Private data: naming convention

The first technique makes a property private by prefixing its name with an underscore. This doesn't protect the property in any way; it merely signals to the outside: "You don't need to know about this property."

In the following code, the properties `._counter` and `._action` are private.

```
class Countdown {
  constructor(counter, action) {
    this._counter = counter;
    this._action = action;
  }
  dec() {
    if (this._counter < 1) return;
    this._counter--;
    if (this._counter === 0) {
      this._action();
    }
  }
}

// The two properties aren't really private:
assert.deepEqual(
  Reflect.ownKeys(new Countdown()),
  ['_counter', '_action']);
```

With this technique, you don't get any protection and private names can clash. On the plus side, it is easy to use.

### 26.3.2 Private data: WeakMaps

Another technique is to use WeakMaps. How exactly that works is explained in [the chapter on WeakMaps](#). This is a preview:

```

let _counter = new WeakMap();
let _action = new WeakMap();

class Countdown {
  constructor(counter, action) {
    _counter.set(this, counter);
    _action.set(this, action);
  }
  dec() {
    let counter = _counter.get(this);
    if (counter < 1) return;
    counter--;
    _counter.set(this, counter);
    if (counter === 0) {
      _action.get(this)();
    }
  }
}

// The two pseudo-properties are truly private:
assert.deepEqual(
  Reflect.ownKeys(new Countdown()),
  []);

```

This technique offers you considerable protection from outside access and there can't be any name clashes. But it is also more complicated to use.

### 26.3.3 More techniques for private data

There are more techniques for private data for classes. These are explained in “Exploring ES6<sup>1</sup>”.

The reason why this section does not go into much depth, is that JavaScript will probably soon have built-in support for private data. Consult the ECMAScript proposal “Class Public Instance Fields & Private Instance Fields<sup>2</sup>” for details.

## 26.4 Subclassing

Classes can also subclass (“extend”) existing classes. As an example, the following class `Employee` subclasses `Person`:

```

class Person {
  constructor(name) {
    this.name = name;
  }
  describe() {
    return `Person named ${this.name}`;
  }
}

```

<sup>1</sup>[http://exploringjs.com/es6/ch\\_classes.html#sec\\_private-data-for-classes](http://exploringjs.com/es6/ch_classes.html#sec_private-data-for-classes)

<sup>2</sup><https://github.com/tc39/proposal-class-fields>



```

    }
    static logNames(persons) {
      for (const person of persons) {
        console.log(person.name);
      }
    }
  }
}

class Employee extends Person {
  constructor(name, title) {
    super(name);
    this.title = title;
  }
  describe() {
    return super.describe() +
      ` (${this.title})`;
  }
}

const jane = new Employee('Jane', 'CTO');
assert.equal(
  jane.describe(),
  'Person named Jane (CTO)');

```

Two comments:

- Inside a `.constructor()` method, you must call the super-constructor via `super()`, before you can access `this`. That's because `this` doesn't exist before the super-constructor was called (this phenomenon is specific to classes).
- Static methods are also inherited. For example, `Employee` inherits the static method `.logNames()`:

```

> 'logNames' in Employee
true

```



### Exercise: Subclassing

`exercises/proto-chains-classes/color_point_class_test.js`

## 26.4.1 Subclasses under the hood (advanced)

The classes `Person` and `Employee` from the previous section are made up of several objects (fig. 26.7). One key insight for understanding how these objects are related, is that there are two prototype chains:

- The instance prototype chain, on the right.
- The class prototype chain, on the left.

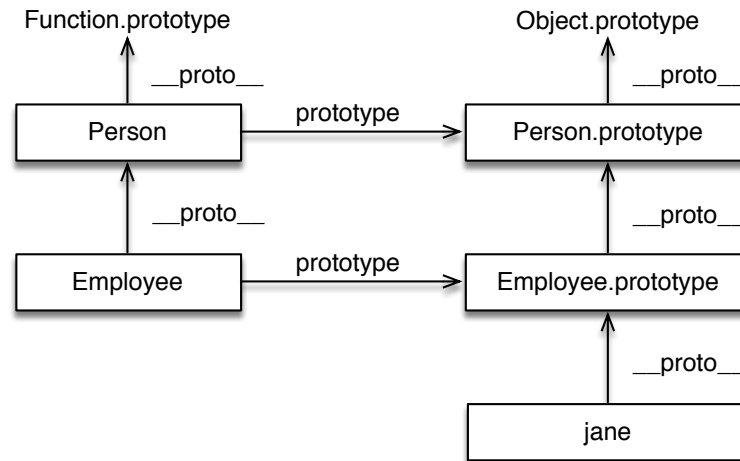


Figure 26.7: These are the objects that make up class `Person` and its subclass, `Employee`. The left column is about classes. The right column is about the `Employee` instance `jane` and its prototype chain.

#### 26.4.1.1 The instance prototype chain (right column)

The instance prototype chain starts with `jane` and continues with `Employee.prototype` and `Person.prototype`. In principle, the prototype chain ends at this point, but we get one more object: `Object.prototype`. This prototype provides services to virtually all objects, which is why it is included here, too:

```
> Object.getPrototypeOf(Person.prototype) === Object.prototype
true
```

#### 26.4.1.2 The class prototype chain (left column)

In the class prototype chain, `Employee` comes first, `Person` next. Afterwards, the chain continues with `Function.prototype`, which is only there, because `Person` is a function and functions need the services of `Function.prototype`.

```
> Object.getPrototypeOf(Person) === Function.prototype
true
```

### 26.4.2 `instanceof` in more detail (advanced)

We have not yet seen how `instanceof` really works. Given the expression `x instanceof C`, how does `instanceof` determine if `x` is an instance of `C`? It does so by checking if `C.prototype` is in the prototype chain of `x`. That is, the following two expressions are equivalent:

```
x instanceof C
C.prototype.isPrototypeOf(x)
```

If we go back to fig. 26.7, we can confirm that the prototype chain does lead us to the following answers:

```

> jane instanceof Employee
true
> jane instanceof Person
true
> jane instanceof Object
true

```

### 26.4.3 Prototype chains of built-in objects (advanced)

Next, we'll use our knowledge of subclassing to understand the prototype chains of a few built-in objects. The following tool function `p()` helps us with our explorations.

```
const p = Object.getPrototypeOf.bind(Object);
```

We extracted method `.getPrototypeOf()` of `Object` and assigned it to `p`.

#### 26.4.3.1 The prototype chain of `{}`

Let's start by examining plain objects:

```

> p({}) === Object.prototype
true
> p(p({})) === null
true

```

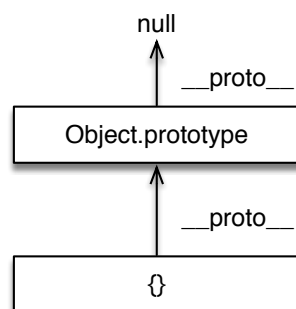


Figure 26.8: The prototype chain of an object created via an object literal starts with that object, continues with `Object.prototype` and ends with `null`.

Fig. 26.8 shows a diagram for this prototype chain. We can see that `{}` really is an instance of `Object` – `Object.prototype` is in its prototype chain.

`Object.prototype` is a curious value: It is an object, but it is not an instance of `Object`:

```

> typeof Object.prototype
'object'
> Object.prototype instanceof Object
false

```

That can't be avoided, because `Object.prototype` can't be in its own prototype chain.

### 26.4.3.2 The prototype chain of []

What does the prototype chain of an Array look like?

```
> p([]) === Array.prototype
true
> p(p([])) === Object.prototype
true
> p(p(p([]))) === null
true
```

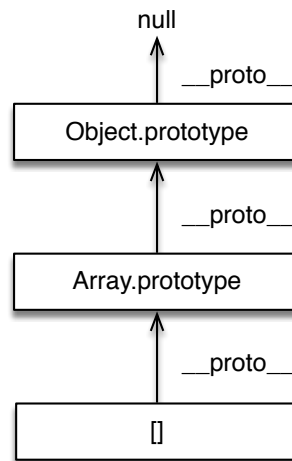


Figure 26.9: The prototype chain of an Array has these members: the Array instance, `Array.prototype`, `Object.prototype`, `null`.

This prototype chain (visualized in fig. 26.9) tells us that an Array object is an instance of Array, which is a subclass of Object.

### 26.4.3.3 The prototype chain of function () {}

Lastly, the prototype chain of an ordinary function tells us that all functions are objects:

```
> p(function () {}) === Function.prototype
true
> p(p(function () {})) === Object.prototype
true
```

## 26.4.4 Mixin classes (advanced)

JavaScript's class system only supports *single inheritance*. That is, each class can have at most one superclass. A way around this limitation is via a technique called *mixin classes* (short: *mixins*).

The idea is as follows: Let's assume there is a class C that extends a class S – its superclass. Mixins are class fragments that are inserted between C and S.

In JavaScript, you can implement a mixin `Mix` via function whose input is a class and whose output is the mixin class fragment – a new class that extends the input. To use `Mix()`, you create `C` as follows.

```
class C extends Mix(S) {
  ...
}
```

Let's look at an example:

```
const Branded = S => class extends S {
  setBrand(brand) {
    this._brand = brand;
    return this;
  }
  getBrand() {
    return this._brand;
  }
};
```

We use this mixin to insert a class between `Car` and `Object`:

```
class Car extends Branded(Object) {
  constructor(model) {
    super();
    this._model = model;
  }
  toString() {
    return `${this.getBrand()} ${this._model}`;
  }
}
```

The following code confirms that the mixin worked: `Car` has method `.setBrand()` of `Branded`.

```
const modelT = new Car('Model T').setBrand('Ford');
assert.equal(modelT.toString(), 'Ford Model T');
```

Mixins are more flexible than normal classes:

- First, you can use the same mixin multiple times, in multiple classes.
- Second, you can use multiple mixins at the same time. As an example, consider an additional mixin called `Stringifiable` that helps with implementing `.toString()`. We could use both `Branded` and `Stringifiable` as follows:

```
class Car extends Stringifiable(Branded(Object)) {
  ...
}
```



Quiz

See quiz app.



**Part VII**

**Collections**





## Chapter 27

# Synchronous iteration

### Contents

---

27.1 What is synchronous iteration about? . . . . .	257
27.2 Core iteration constructs: iterables and iterators . . . . .	258
27.3 Iterating manually . . . . .	259
27.3.1 Iterating over an iterable via <code>while</code> . . . . .	259
27.4 Iteration in practice . . . . .	260
27.4.1 Arrays . . . . .	260
27.4.2 Sets . . . . .	260
27.5 Quick reference: synchronous iteration . . . . .	261
27.5.1 Iterable data sources . . . . .	261
27.5.2 Iterating constructs . . . . .	261
27.6 Further reading . . . . .	262

---

### 27.1 What is synchronous iteration about?

Synchronous iteration is a *protocol* (interfaces plus rules for using them) that connects two groups of entities in JavaScript:

- **Data sources:** On one hand, data comes in all shapes and sizes. In JavaScript’s standard library, you have the linear data structure `Array`, the ordered collection `Set` (elements are ordered by time of addition), the ordered dictionary `Map` (entries are ordered by time of addition), and more. In libraries, you may find tree-shaped data structures and more.
- **Data consumers:** On the other hand, you have a whole class of mechanisms and algorithms that only need to access values *sequentially*: one at a time plus a way to tell when everything is done. Examples include the `for-of` loop and spreading into `Array` literals (via `...`).

The iteration protocol connects these two groups via the interface `Iterable`: data sources deliver their contents sequentially “through it”; data consumers get their input via it.

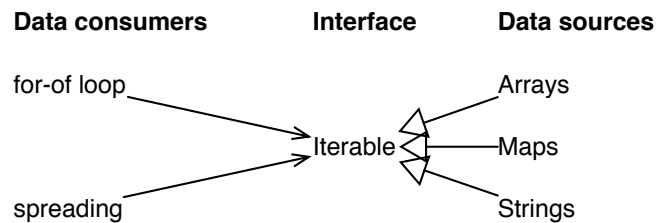


Figure 27.1: Data consumers such as the `for-of` loop use the interface `Iterable`. Data sources such as `Arrays` implement that interface.

The diagram in fig. 27.1 illustrates how iteration works: data consumers use the interface `Iterable`; data sources implement it. Note that in JavaScript, *implementing* only means having the methods described by the interface; the interface itself only exists in the specification.

Both sources and consumers of data profit from this arrangement:

- If you develop a new data structure, you only need to implement `Iterable` and a raft of tools can immediately be applied to it.
- If you write code that uses iteration, it automatically works with many sources of data.

## 27.2 Core iteration constructs: iterables and iterators

Two roles (described by interfaces) form the core of iteration (fig. 27.2):

- An *iterable* is an object whose contents can be traversed sequentially.
- An *iterator* is the pointer used for the traversal.

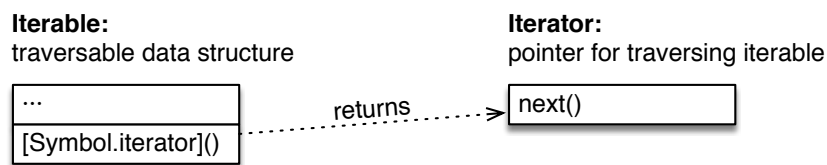


Figure 27.2: Iteration has two main interfaces: `Iterable` and `Iterator`. The former has a method that returns the latter.

These are type definitions (in TypeScript's notation) for the interfaces of the iteration protocol:

```

interface Iterable<T> {
  [Symbol.iterator]() : Iterator<T>;
}
  
```

```

interface Iterator<T> {
  next() : IteratorResult<T>;
}
  
```

```

interface IteratorResult<T> {
  value: T;
}
  
```

```

    done: boolean;
  }

```

The interfaces are used as follows:

- You ask an `Iterable` for an iterator via the method whose key is `Symbol.iterator`.
- The `Iterator` returns the iterated values via its method `.next()`.
- The values are not returned directly, but wrapped in objects with two properties:
  - `.value` is the iterated value.
  - `.done` indicates if the end of the iteration has been reached, yet. It is `true` after the last iterated value and `false` beforehand.

## 27.3 Iterating manually

This is an example of using the iteration protocol:

```

const iterable = ['a', 'b'];

// The iterable is a factory for iterators:
const iterator = iterable[Symbol.iterator]();

// Call .next() until .done is true:
assert.deepEqual(
  iterator.next(), { value: 'a', done: false });
assert.deepEqual(
  iterator.next(), { value: 'b', done: false });
assert.deepEqual(
  iterator.next(), { value: undefined, done: true });

```

### 27.3.1 Iterating over an iterable via `while`

The following code demonstrates how to use a `while` loop to iterate over an iterable:

```

function logAll(iterable) {
  const iterator = iterable[Symbol.iterator]();
  while (true) {
    const {value, done} = iterator.next();
    if (done) break;
    console.log(value);
  }
}

logAll(['a', 'b']);
// Output:
// 'a'
// 'b'

```

**Exercise: Using sync iteration manually**`exercises/sync-iteration-use/sync_iteration_manually_exrc.js`

## 27.4 Iteration in practice

We have seen how to use the iteration protocol manually and it is relatively cumbersome. But the protocol is not meant to be used directly – it is meant to be used via higher-level language constructs built on top of it. This section shows what that looks like.

### 27.4.1 Arrays

JavaScript's Arrays are iterable. That enables you to use the `for-of` loop:

```
const myArray = ['a', 'b', 'c'];

for (const x of myArray) {
  console.log(x);
}
// Output:
// 'a'
// 'b'
// 'c'
```

Destructuring via Array patterns (explained later) also uses iteration, under the hood:

```
const [first, second] = myArray;
assert.equal(first, 'a');
assert.equal(second, 'b');
```

### 27.4.2 Sets

JavaScript's Set data structure is iterable. That means, `for-of` works:

```
const mySet = new Set().add('a').add('b').add('c');

for (const x of mySet) {
  console.log(x);
}
// Output:
// 'a'
// 'b'
// 'c'
```

As does Array-destructuring:

```
const [first, second] = mySet;
assert.equal(first, 'a');
assert.equal(second, 'b');
```

## 27.5 Quick reference: synchronous iteration

### 27.5.1 Iterable data sources

The following built-in data sources are iterable:

- Arrays
- Strings
- Maps
- Sets
- (Browsers: DOM data structures)

To iterate over the properties of objects, you need helpers such as `Object.keys()` and `Object.entries()`. That is necessary, because properties exist at a different level that is complementary to the level of data structures.

### 27.5.2 Iterating constructs

The following constructs support iteration:

- Destructuring via an Array pattern:
 

```
const [x,y] = iterable;
```
- The for-of loop:
 

```
for (const x of iterable) { /*...*/ }
```
- `Array.from()`:
 

```
const arr = Array.from(iterable);
```
- Spreading (via `...`) into Arrays and function calls:
 

```
const arr = [...iterable];
func(...iterable);
```
- `new Map()` and `new Set()`:
 

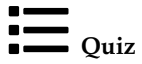
```
const m = new Map(iterableOverKeyValuePairs);
const s = new Set(iterableOverElements);
```
- `Promise.all()` and `Promise.race()`:
 

```
const promise = Promise.all(iterableOverPromises);
const promise = Promise.race(iterableOverPromises);
```
- `yield*`:

```
function* generatorFunction() {  
  yield* iterable;  
}
```

## 27.6 Further reading

For more details on synchronous iteration, consult “Exploring ES6<sup>1</sup>”.



Quiz

See quiz app.

---

<sup>1</sup>[http://exploringjs.com/es6/ch\\_iteration.html](http://exploringjs.com/es6/ch_iteration.html)

# Chapter 28

## Arrays (Array)

### Contents

---

<b>28.1 The two roles of Arrays in JavaScript</b>	<b>264</b>
<b>28.2 Basic Array operations</b>	<b>264</b>
28.2.1 Arrays: creating, reading, writing	264
28.2.2 Arrays: <code>.length</code>	264
28.2.3 Spreading into Arrays	265
28.2.4 Arrays: listing indices and entries	266
28.2.5 Is a value an Array?	266
<b>28.3 for-of and Arrays</b>	<b>266</b>
28.3.1 for-of: iterating over elements	267
28.3.2 for-of: iterating over [index, element] pairs	267
<b>28.4 Array-like objects</b>	<b>267</b>
<b>28.5 Converting iterable and Array-like values to Arrays</b>	<b>268</b>
28.5.1 Converting iterables to Arrays via spreading ( <code>...</code> )	268
28.5.2 Converting iterables and Array-like objects to Arrays via <code>Array.from()</code> (advanced)	268
<b>28.6 Creating and filling Arrays with arbitrary lengths</b>	<b>269</b>
<b>28.7 Multidimensional Arrays</b>	<b>270</b>
<b>28.8 More Array features (advanced)</b>	<b>270</b>
28.8.1 Array elements are (slightly special) properties	270
28.8.2 Arrays are dictionaries and can have holes	271
<b>28.9 Adding and removing elements (destructively and non-destructively)</b>	<b>272</b>
28.9.1 Prepending elements and Arrays	272
28.9.2 Appending elements and Arrays	273
28.9.3 Removing elements	274
<b>28.10 Methods: iteration and transformation (<code>.find()</code>, <code>.map()</code>, <code>.filter()</code>, etc.)</b>	<b>274</b>
28.10.1 External iteration vs. internal iteration	274
28.10.2 Callbacks for iteration and transformation methods	275
28.10.3 Searching elements: <code>.find()</code> , <code>.findIndex()</code>	275
28.10.4 <code>.map()</code> : copy while giving elements new values	276
28.10.5 <code>.filter()</code> : only keep some of the elements	276

28.10.6 <code>.reduce()</code> : deriving a value from an Array (advanced)	277
<b>28.11 <code>.sort()</code>: sorting Arrays</b>	<b>280</b>
28.11.1 Customizing the sort order	280
28.11.2 Sorting numbers	281
28.11.3 Sorting objects	281
<b>28.12 Quick reference: <code>Array&lt;T&gt;</code></b>	<b>281</b>
28.12.1 <code>new Array()</code>	282
28.12.2 Static methods of Array	282
28.12.3 Methods of <code>Array&lt;T&gt;.prototype</code>	282
28.12.4 Sources	288

---

## 28.1 The two roles of Arrays in JavaScript

Arrays play two roles in JavaScript:

- Tuples: Arrays-as-tuples have a fixed number of indexed elements. Each of those elements can have a different type.
- Sequences: Arrays-as-sequences have a variable number of indexed elements. Each of those elements has the same type.

In reality, there are also mixtures of these two roles.

Notably, Arrays-as-sequences are so flexible that you can use them as (traditional) arrays, stacks and queues (see exercise at the end of this chapter).

## 28.2 Basic Array operations

### 28.2.1 Arrays: creating, reading, writing

Create an Array:

```
const arr = ['a', 'b', 'c'];
```

Read an Array element (indices start at zero):

```
assert.equal(arr[0], 'a');
```

Change an Array element:

```
arr[0] = 'x';
assert.deepEqual(arr, ['x', 'b', 'c']);
```

The range of Array indices is 32 bits (excluding the maximum length):  $[0, 2^{32}-1)$

### 28.2.2 Arrays: `.length`

Every Array has a property `.length` that can be used to both read and change(!) the number of elements in an Array.



The length of an Array is always the highest index plus one:

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

If you write to the Array at the index of the length, you append an element:

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

Another way of (destructively) appending an element is via the Array method `.push()`:

```
> arr.push('d');
> arr
[ 'a', 'b', 'c', 'd' ]
```

If you set `.length`, you are pruning the Array, by removing elements:

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



**Exercise: Removing empty lines via `.push()`**

`exercises/arrays/remove_empty_lines_push_test.js`

### 28.2.3 Spreading into Arrays

Inside an Array literal, a *spread element* consists of three dots (`...`) followed by an expression. It results in the expression being evaluated and then iterated over. Each iterated value becomes an additional Array element. For example:

```
> const iterable = ['b', 'c'];
> ['a', ...iterable, 'd']
[ 'a', 'b', 'c', 'd' ]
```

Spreading is convenient for concatenating Arrays and other iterables into Arrays:

```
const arr1 = ['a', 'b'];
const arr2 = ['c', 'd'];

const concatenated = [...arr1, ...arr2, 'e'];
assert.deepEqual(
  concatenated,
  ['a', 'b', 'c', 'd', 'e']);
```

### 28.2.4 Arrays: listing indices and entries

Method `.keys()` lists the indices of an Array:

```
const arr = ['a', 'b'];
assert.deepEqual(
  [...arr.keys()], // (A)
  [0, 1]);
```

`.keys()` returns an iterable. In line A, we spread to obtain an Array.

Listing Array indices is different from listing properties. When you do the latter, you get the indices – but as strings – plus non-index property keys:

```
const arr = ['a', 'b'];
arr.prop = true;

assert.deepEqual(
  Object.keys(arr),
  ['0', '1', 'prop']);
```

Method `.entries()` lists the contents of an Array as `[index, element]` pairs:

```
const arr = ['a', 'b'];
assert.deepEqual(
  [...arr.entries()],
  [[0, 'a'], [1, 'b']]);
```

### 28.2.5 Is a value an Array?

These are two ways of checking if a value is an Array:

```
> [] instanceof Array
true
> Array.isArray([])
true
```

`instanceof` is usually fine. You need `Array.isArray()` if a value may come from another *realm*. Roughly, a realm is an instance of JavaScript's global scope. Some realms are isolated from each other (e.g. web workers in browsers), but there are also realms between which you can move data (e.g. same-origin iframes in browsers). `x instanceof Array` checks the prototype chain of `x` and therefore returns `false` if `x` is an Array from another realm.

`typeof` categorizes Arrays as objects:

```
> typeof []
'object'
```

## 28.3 for-of and Arrays

We have already encountered the `for-of` loop. This section briefly recaps how to use it for Arrays.

### 28.3.1 for-of: iterating over elements

The following for-of loop iterates over the elements of an Array.

```
for (const element of ['a', 'b']) {
  console.log(element);
}
// Output:
// 'a'
// 'b'
```

### 28.3.2 for-of: iterating over [index, element] pairs

The following for-of loop iterates over [index, element] pairs. Destructuring (described later), gives us convenient syntax for setting up index and element in the head of for-of.

```
for (const [index, element] of ['a', 'b'].entries()) {
  console.log(index, element);
}
// Output:
// 0, 'a'
// 1, 'b'
```

## 28.4 Array-like objects

Some operations that work with Arrays, require only the bare minimum: Values must only be *Array-like*. An Array-like value is an object with the following properties:

- `.length`: holds the length of the Array-like object.
- `['0']`: holds the element at index 0. (Etc.)

A TypeScript interface for `ArrayLike` looks as follows.

```
interface ArrayLike<T> {
  length: number;
  [n: number]: T;
}
```

`Array.from()` accepts Array-like objects and converts them to Arrays:

```
// If you omit .length, it is interpreted as 0
assert.deepStrictEqual(
  Array.from({}),
  []);

assert.deepStrictEqual(
  Array.from({length:2, 0:'a', 1:'b'}),
  ['a', 'b']);
```

Array-like objects used to be common before ES6; now you don't see them very often.

## 28.5 Converting iterable and Array-like values to Arrays

There are two common ways of converting iterable and Array-like values to Arrays: spreading and `Array.from()`.

### 28.5.1 Converting iterables to Arrays via spreading (...)

Inside an Array literal, spreading via `...` converts any iterable object into a series of Array elements. For example:

```
// Get an Array-like collection from a web browser's DOM
const domCollection = document.querySelectorAll('a');

// Alas, the collection is missing many Array methods
assert.equal('map' in domCollection, false);

// Solution: convert it to an Array
const arr = [...domCollection];
assert.deepEqual(
  arr.map(x => x.href),
  ['http://2ality.com', 'http://exploringjs.com']);
```

The conversion works, because the DOM collection is iterable.

### 28.5.2 Converting iterables and Array-like objects to Arrays via `Array.from()` (advanced)

`Array.from()` can be used in two modes.

#### 28.5.2.1 Mode 1 of `Array.from()`: converting

The first mode has the following type signature:

```
.from<T>(iterable: Iterable<T> | ArrayLike<T>): T[];
```

Interface `Iterable` is shown in [the chapter that introduces iteration](#). Interface `ArrayLike` appeared [earlier in this chapter](#).

With a single parameter, `Array.from()` converts anything iterable or Array-like to an Array:

```
> Array.from(new Set(['a', 'b']))
[ 'a', 'b' ]
> Array.from({length: 2, 0:'a', 1:'b'})
[ 'a', 'b' ]
```

#### 28.5.2.2 Mode 2 of `Array.from()`: converting and mapping

The second mode of `Array.from()` involves two parameters:

```
.from<T, U>(
  iterable: Iterable<T> | ArrayLike<T>,
  mapFunc: (v: T, i: number) => U,
  thisArg?: any)
: U[];
```

In this mode, `Array.from()` does several things:

- It iterates over `iterable`.
- It applies `mapFunc` to each iterated value.
- It collects the results in a new `Array` and returns it.

The optional parameter `thisArg` specifies a `this` for `mapFunc`.

That means that we are going from an iterable with elements of type `T` to an `Array` with elements of type `U`.

This is an example:

```
> Array.from(new Set(['a', 'b']), x => x + x)
[ 'aa', 'bb' ]
```

## 28.6 Creating and filling Arrays with arbitrary lengths

The best way of creating an `Array` is via an `Array` literal. However, you can't use one if you don't know the length the `Array` during development. Or if you want to keep the length flexible. Then I recommend the following techniques for creating and possibly filling `Arrays`:

- Do you need to create an empty `Array` that you'll fill completely, later on?

```
> new Array(3)
[ , , ]
```

Note that the result has 3 holes – the last comma in an `Array` literal is always ignored.

- Do you need to create an `Array` initialized with a primitive value?

```
> new Array(3).fill(0)
[0, 0, 0]
```

Caveat: If you use `.fill()` with an object then each `Array` element will refer to the same object.

- Do you need to create an `Array` initialized with objects?

```
> Array.from({length: 3}, () => ({}))
[{}, {}, {}]
```

- Do you need to create a range of integers?

```
const START = 2;
const END = 5;
assert.deepEqual(
  Array.from({length: END-START}, (x, i) => i+START),
  [2, 3, 4]);
```

If you are dealing with Arrays of integers or floats, consider *Typed Arrays* – which were created for this purpose.

## 28.7 Multidimensional Arrays

JavaScript does not have real multidimensional Arrays; you need to resort to Arrays whose elements are Arrays:

```
const DIM_X = 4;
const DIM_Y = 3;
const DIM_Z = 2;

const arr = [];
for (let x=0; x<DIM_X; x++) {
  arr[x] = []; // (A)
  for (let y=0; y<DIM_Y; y++) {
    arr[x][y] = []; // (B)
    for (let z=0; z<DIM_Z; z++) {
      arr[x][y][z] = 0; // (C)
    }
  }
}
arr[3][0][1] = 7;
assert.deepEqual(arr, [
  [ [ 0, 0 ], [ 0, 0 ], [ 0, 0 ] ],
  [ [ 0, 0 ], [ 0, 0 ], [ 0, 0 ] ],
  [ [ 0, 0 ], [ 0, 0 ], [ 0, 0 ] ],
  [ [ 0, 7 ], [ 0, 0 ], [ 0, 0 ] ],
]);
```

Observations:

- We grow the Arrays by assigning values to slots whose indices are the current lengths.
- Each dimension – except the last one – is an Array whose elements are the next dimension (line A, line B).
- The last dimension contains the actual values (line C).

## 28.8 More Array features (advanced)

In this section, we look at phenomena you don't encounter that often when working with Arrays.

### 28.8.1 Array elements are (slightly special) properties

You'd think that Array elements are special, because you are accessing them via numbers. But the square brackets operator ([ ]) for doing so, is the same operator that is used for accessing properties. And,

according to the language specification, it coerces any value (that is not a symbol) to a string. Therefore: Array elements are (almost) normal properties (line A) and it doesn't matter if you use numbers or strings as indices (lines B and C):

```
const arr = ['a', 'b'];
arr.prop = 123;
assert.deepEqual(
  Object.keys(arr),
  ['0', '1', 'prop']); // (A)

assert.equal(arr[0], 'a'); // (B)
assert.equal(arr['0'], 'a'); // (C)
```

To make matters even more confusing, this is only how the language specification defines things (the theory of JavaScript, if you will). Most JavaScript engines optimize under the hood and do use numbers (even integers) to access Array elements (the practice of JavaScript, if you will).

Property keys (strings!) that are used for Array elements are called *indices*. A string `str` is an index if converting it to a 32-bit unsigned integer and back results in the original value. Written as a formula:

`ToString(ToUint32(key)) === key`

JavaScript treats indices specially when listing property keys (of all objects!). They always come first and are sorted numerically, not lexicographically (where `'10'` would come before `'2'`):

```
const arr = 'abcdefghijk'.split('');
arr.prop = 123;
assert.deepEqual(
  Object.keys(arr),
  ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'prop']);
```

Note that `.length`, `.entries()` and `.keys()` treat Array indices as numbers and ignore non-index properties:

```
const arr = ['a', 'b'];
arr.prop = true;

assert.deepEqual(
  Object.keys(arr), ['0', '1', 'prop']);

assert.equal(arr.length, 2);
assert.deepEqual(
  [...arr.keys()], [0, 1]);
assert.deepEqual(
  [...arr.entries()], [[0, 'a'], [1, 'b']]);
```

We used a spread element (`...`) to convert the iterables returned by `.keys()` and `.entries()` to Arrays.

## 28.8.2 Arrays are dictionaries and can have holes

JavaScript supports two kinds of Arrays:

- Dense Arrays: are Arrays whose elements form a contiguous sequence. That is the only kind of Array we have seen so far.
- Sparse Arrays: are Arrays that have holes in them. That is, some indices are missing.

In general, it's best to avoid holes, because they make your code more complicated and are not handled consistently by Array methods. Additionally, JavaScript engines optimize dense Arrays, so they are faster.

You can create holes by skipping indices when assigning elements:

```
const arr = [];
arr[0] = 'a';
arr[2] = 'c';

assert.deepEqual(Object.keys(arr), ['0', '2']); // (A)

assert.equal(0 in arr, true); // element
assert.equal(1 in arr, false); // hole
```

In line A, we are using `Object.keys()`, because `arr.keys()` treats holes as if they were undefined elements and does not reveal them.

Another way of creating holes is to skip elements in Array literals:

```
const arr = ['a', , 'c'];

assert.deepEqual(Object.keys(arr), ['0', '2']);
```

And you can delete Array elements:

```
const arr = ['a', 'b', 'c'];
assert.deepEqual(Object.keys(arr), ['0', '1', '2']);
delete arr[1];
assert.deepEqual(Object.keys(arr), ['0', '2']);
```

For more information on how JavaScript handles holes in Arrays, consult “Exploring ES6<sup>1</sup>”.

## 28.9 Adding and removing elements (destructively and non-destructively)

JavaScript's Array is quite flexible and more like a combination of array, stack and queue. This section explores ways of adding and removing Array elements. Most operations can be performed both destructively (modifying the Array) and non-destructively (producing a modified copy).

### 28.9.1 Prepending elements and Arrays

In the following code, we destructively prepend single elements to `arr1` and an Array to `arr2`:

```
const arr1 = ['a', 'b'];
arr1.unshift('x', 'y'); // prepend single elements
assert.deepEqual(arr1, ['x', 'y', 'a', 'b']);
```

---

<sup>1</sup>[http://exploringjs.com/es6/ch\\_arrays.html#sec\\_array-holes](http://exploringjs.com/es6/ch_arrays.html#sec_array-holes)



```
const arr2 = ['a', 'b'];
arr2.unshift(...['x', 'y']); // prepend Array
assert.deepEqual(arr2, ['x', 'y', 'a', 'b']);
```

Spreading lets us unshift an Array into arr2.

Non-destructive prepending is done via spread elements:

```
const arr1 = ['a', 'b'];
assert.deepEqual(
  ['x', 'y', ...arr1], // prepend single elements
  ['x', 'y', 'a', 'b']);
assert.deepEqual(arr1, ['a', 'b']); // unchanged!

const arr2 = ['a', 'b'];
assert.deepEqual(
  [...['x', 'y'], ...arr2], // prepend Array
  ['x', 'y', 'a', 'b']);
assert.deepEqual(arr2, ['a', 'b']); // unchanged!
```

## 28.9.2 Appending elements and Arrays

In the following code, we destructively append single elements to arr1 and an Array to arr2:

```
const arr1 = ['a', 'b'];
arr1.push('x', 'y'); // append single elements
assert.deepEqual(arr1, ['a', 'b', 'x', 'y']);

const arr2 = ['a', 'b'];
arr2.push(...['x', 'y']); // append Array
assert.deepEqual(arr2, ['a', 'b', 'x', 'y']);
```

Spreading lets us push an Array into arr2.

Non-destructive appending is done via spread elements:

```
const arr1 = ['a', 'b'];
assert.deepEqual(
  [...arr1, 'x', 'y'], // append single elements
  ['a', 'b', 'x', 'y']);
assert.deepEqual(arr1, ['a', 'b']); // unchanged!

const arr2 = ['a', 'b'];
assert.deepEqual(
  [...arr2, ...['x', 'y']], // append Array
  ['a', 'b', 'x', 'y']);
assert.deepEqual(arr2, ['a', 'b']); // unchanged!
```

### 28.9.3 Removing elements

These are three destructive ways of removing Array elements:

```
// Destructively remove first element:
const arr1 = ['a', 'b', 'c'];
assert.equal(arr1.shift(), 'a');
assert.deepEqual(arr1, ['b', 'c']);

// Destructively remove last element:
const arr2 = ['a', 'b', 'c'];
assert.equal(arr2.pop(), 'c');
assert.deepEqual(arr2, ['a', 'b']);

// Remove one or more elements anywhere:
const arr3 = ['a', 'b', 'c'];
assert.deepEqual(arr3.splice(1, 1), ['b']);
assert.deepEqual(arr3, ['a', 'c']);
```

`.splice()` is covered in more detail in [the quick reference section](#).

Destructuring via a rest element lets you non-destructively remove elements from the beginning of an Array (destructuring is covered later).

```
const arr1 = ['a', 'b', 'c'];
// Ignore first element, extract remaining elements
const [, ...arr2] = arr1;

assert.deepEqual(arr1, ['a', 'b', 'c']); // unchanged!
assert.deepEqual(arr2, ['b', 'c']);
```

Alas, a rest element must always come last in an Array. Therefore, you can only use it to extract suffixes.



#### Exercise: Implementing a queue via an Array

`exercises/arrays/queue_via_array_test.js`

## 28.10 Methods: iteration and transformation (`.find()`, `.map()`, `.filter()`, etc.)

In this section, we take a look at Array methods for iterating over Arrays and for transforming Arrays. Before we do so, let's consider two different approaches for iteration. It will help us understand how the methods work.

### 28.10.1 External iteration vs. internal iteration

Let's assume that your code wants to iterate over the values “inside” an object. Two common approaches for making that possible, are:

- External iteration (pull): Your code asks the object for the values, via an iteration protocol. For example, the for-of loop is based on JavaScript's iteration protocol:

```
for (const x of ['a', 'b']) {
  console.log(x);
}
```

- Internal iteration (push): Your code is passed to a method of the object, via a function. And that method successively feeds the values to the function. For example, Arrays have the method .forEach():

```
['a', 'b'].forEach((x) => {
  console.log(x);
});
```

### 28.10.2 Callbacks for iteration and transformation methods

Callbacks for iteration and transformation methods have signatures such as the following one.

callback: (value: T, index: number, array: Array<T>) => boolean

That is, the callback gets three parameters (it is free to ignore any of them):

- value is the most important one. This parameter holds the iterated value that is currently being processed.
- index can additionally tell the callback what the index of the iterated value is.
- array points to the current Array (the receiver of the method call). Some algorithms need to refer to the whole Array – e.g. to search it for answers. This parameter lets you write reusable callbacks for such algorithms.

What the callback is expected to return, depends on the iteration method: Sometimes it's an arbitrary value, sometimes it's a boolean.

### 28.10.3 Searching elements: .find(), .findIndex()

.find() returns the first element for which its callback returns a truthy value:

```
> [6, -5, 8].find(x => x < 0)
-5
> [6, 5, 8].find(x => x < 0)
undefined
```

.findIndex() returns the index of the first element for which its callback returns a truthy value:

```
> [6, -5, 8].findIndex(x => x < 0)
1
> [6, 5, 8].findIndex(x => x < 0)
-1
```

.findIndex() can be implemented as follows:

```
function findIndex(arr, callback) {
  for (const [i, x] of arr.entries()) {
```

```

    if (callback(x, i, arr)) {
      return i;
    }
  }
  return -1;
}

assert.equal(1, findIndex(['a', 'b', 'c'], x => x === 'b'));

```

#### 28.10.4 .map(): copy while giving elements new values

`.map()` returns a copy of the receiver. The elements of the copy are the results of applying `map`'s callback parameter to the elements of the receiver.

All of this is easier to understand via examples:

```

> [1, 2, 3].map(x => x * 3)
[ 3, 6, 9 ]
> ['how', 'are', 'you'].map(str => str.toUpperCase())
[ 'HOW', 'ARE', 'YOU' ]
> [true, true, true].map( (_, index) => index)
[ 0, 1, 2 ]

```

Note: `_` is just another variable name.

`.map()` can be implemented as follows:

```

function map(arr, mapFunc) {
  const result = [];
  for (const [i, x] of arr.entries()) {
    result.push(mapFunc(x, i, arr));
  }
  return result;
}

assert.deepEqual(
  map(['a', 'b', 'c'], (x, i) => `${i}.${x}`),
  ['0.a', '1.b', '2.c']);

```



#### Exercise: Numbering lines via `.map()`

`exercises/arrays/number_lines_test.js`

#### 28.10.5 .filter(): only keep some of the elements

The Array method `.filter()` returns an Array collecting all elements for which the callback returns a truthy value.

For example:

```
> [-1, 2, 5, -7, 6].filter(x => x >= 0)
[ 2, 5, 6 ]
> ['a', 'b', 'c', 'd'].filter((_,i) => (i%2)===0)
[ 'a', 'c' ]
```

.filter() can be implemented as follows:

```
function filter(arr, filterFunc) {
  const result = [];
  for (const [i, x] of arr.entries()) {
    if (filterFunc(x, i, arr)) {
      result.push(x);
    }
  }
  return result;
}

assert.deepEqual(
  filter([ 1, 'a', 5, 4, 'x'], x => typeof x === 'number'),
  [1, 5, 4]);
assert.deepEqual(
  filter([ 1, 'a', 5, 4, 'x'], x => typeof x === 'string'),
  ['a', 'x']);
```



#### Exercise: Removing empty lines via .filter()

exercises/arrays/remove\_empty\_lines\_filter\_test.js

### 28.10.6 .reduce(): deriving a value from an Array (advanced)

Method .reduce() is a powerful tool for computing a “summary” of an Array. A summary can be any kind of value:

- A number. For example, the sum of all Array elements.
- An Array. For example, a copy of the Array, with elements multiplied by 2.
- Etc.

This operation is also known as `foldl` (“fold left”) in functional programming and popular there. One caveat is that it can make code difficult to understand.

.reduce() has the following type signature (inside an `Array<T>`):

```
.reduce<U>(
  callback: (accumulator: U, element: T, index: number, array: T[]) => U,
  init?: U)
: U
```

T is the type of the Array elements, U is the type of the summary. The two may or may not be different.

To compute the summary of an Array `arr`, .reduce() feeds all Array elements to its callback, one at a time:

```

const accumulator_0 = callback(init, arr[0]);
const accumulator_1 = callback(accumulator_0, arr[1]);
const accumulator_2 = callback(accumulator_1, arr[2]);
// Etc.

```

`callback` combines the previously computed summary (stored in its parameter `accumulator`) with the current Array element and returns the next accumulator.

The result of `.reduce()` is the last result of `callback`, after it has visited all elements. In other words: `callback` does most of the work, `.reduce()` just invokes it in a useful manner.

Let's look at an example of `.reduce()` in action: function `addAll()` computes the sum of all numbers in an Array `arr`.

```

function addAll(arr) {
  const startSum = 0;
  const callback = (sum, element) => sum + element;
  return arr.reduce(callback, startSum);
}
assert.equal(addAll([1, 2, 3]), 6); // (A)
assert.equal(addAll([7, -4, 2]), 5);

```

In this case, the accumulator holds the sum of all Array elements that `callback` has already visited.

How was the result 6 derived from the Array in line A? Via the following invocations of `callback`:

```

callback(0, 1) --> 1
callback(1, 2) --> 3
callback(3, 3) --> 6

```

Notes:

- The first parameters are the current accumulators (starting with parameter `init` of `.reduce()`).
- The second parameters are the current Array elements.
- The results are the next accumulators.
- The last result of `callback` is also the result of `.reduce()`.

Alternatively, we could have implemented `addAll()` via a `for-of` loop:

```

function addAll(arr) {
  let sum = 0;
  for (const element of arr) {
    sum = sum + element;
  }
  return sum;
}

```

It's hard to say which of the two implementations is "better": The one based on `.reduce()` is a little more concise, while the one based on `for-of` may be a little easier to understand – especially if you are not familiar with functional programming.

**28.10.6.1 Example: finding indices via .reduce()**

The following function is an implementation of the Array method `.indexOf()`. It returns the first index at which the given `searchValue` appears inside the Array `arr`:

```
const NOT_FOUND = -1;
function indexOf(arr, searchValue) {
  return arr.reduce(
    (result, elem, index) => {
      if (result !== NOT_FOUND) {
        // We have already found something: don't change anything
        return result;
      }
      else if (elem === searchValue) {
        return index;
      }
      else {
        return NOT_FOUND;
      }
    },
    NOT_FOUND);
}
assert.equal(indexOf(['a', 'b', 'c'], 'b'), 1);
assert.equal(indexOf(['a', 'b', 'c'], 'x'), -1);
```

One limitation of `.reduce()` is that you can't finish early (in a `for-of` loop, you can `break`). Here, we don't do anything once we have found what we were looking for.

**28.10.6.2 Example: doubling Array elements**

Function `double(arr)` returns a copy of `inArr` whose elements are all multiplied by 2:

```
function double(inArr) {
  return inArr.reduce(
    (outArr, element) => {
      outArr.push(element * 2);
      return outArr;
    },
    []);
}
assert.deepEqual(
  double([1, 2, 3]),
  [2, 4, 6]);
```

We modify the initial value `[]` by pushing into it. A non-destructive, more functional version of `double()` looks as follows:

```
function double(inArr) {
  return inArr.reduce(
    // Don't change `outArr`, return a fresh Array
    (outArr, element) => [...outArr, element * 2],
```

```

    []);
  }
  assert.deepEqual(
    double([1, 2, 3]),
    [2, 4, 6]);

```

This version is more elegant, but also slower and uses more memory.



#### Exercises: `.reduce()`

- `map()` via `.reduce()`: `exercises/arrays/map_via_reduce_test.js`
- `filter()` via `.reduce()`: `exercises/arrays/filter_via_reduce_test.js`
- `countMatches()` via `.reduce()`: `exercises/arrays/count_matches_via_reduce_test.js`

## 28.11 `.sort()`: sorting Arrays

`.sort()` has the following type definition:

```
sort(compareFunc?: (a: T, b: T) => number): this
```

`.sort()` always sorts string representations of the elements. These representations are compared via `<`. This operator compares *lexicographically* (the first characters are most significant). You can see that when comparing numbers:

```

> [200, 3, 10].sort()
[ 10, 200, 3 ]

```

When comparing human-language strings, you need to be aware that they are compared according to their code unit values (char codes):

```

> ['pie', 'cookie', 'éclair', 'Pie', 'Cookie', 'Éclair'].sort()
[ 'Cookie', 'Pie', 'cookie', 'pie', 'Éclair', 'éclair' ]

```

As you can see, all unaccented uppercase letters come before all unaccented lowercase letter, which come before all accented letters. Use `Intl`, the JavaScript internationalization API<sup>2</sup>, if you want proper sorting for human languages.

Lastly, `.sort()` sorts *in place*: it changes and returns its receiver:

```

> const arr = ['a', 'c', 'b'];
> arr.sort() === arr
true
> arr
[ 'a', 'b', 'c' ]

```

### 28.11.1 Customizing the sort order

You can customize the sort order via the parameter, `compareFunc`, which returns a number that is:

<sup>2</sup>[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Intl](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl)



- negative if `a < b`
- zero if `a === b`
- positive if `a > b`

Tip for remembering these rules: a negative number is *less than* zero (etc.).

### 28.11.2 Sorting numbers

You can use the following helper function to compare numbers:

```
function compareNumbers(a, b) {
  if (a < b) {
    return -1;
  } else if (a === b) {
    return 0;
  } else {
    return 1;
  }
}
assert.deepEqual(
  [200, 3, 10].sort(compareNumbers),
  [3, 10, 200]);
```

The following is a quick and dirty alternative. Its downsides are that it is cryptic and that there is a risk of numeric overflow:

```
> [200, 3, 10].sort((a,b) => a-b)
[ 3, 10, 200 ]
```

### 28.11.3 Sorting objects

You also need to use a compare function if you want to sort objects. As an example, the following code shows how to sort objects by age.

```
const arr = [ {age: 200}, {age: 3}, {age: 10} ];
assert.deepEqual(
  arr.sort((obj1, obj2) => obj1.age - obj2.age),
  [{ age: 3 }, { age: 10 }, { age: 200 } ] );
```



Exercise: Sorting objects by name

`exercises/arrays/sort_objects_test.js`

## 28.12 Quick reference: `Array<T>`

Legend:

- R: method does not change the receiver (non-destructive).

- `W`: method changes the receiver (destructive).

### 28.12.1 `new Array()`

`new Array(n)` creates an Array of length `n`, that contains `n` holes:

```
// Trailing commas are always ignored.
// Therefore: number of commas = number of holes
assert.deepEqual(new Array(3), [, , ,]);
```

`new Array()` creates an empty Array. However, I recommend to always use `[]`, instead.

### 28.12.2 Static methods of Array

- `Array.from<T>(iterable: Iterable<T> | ArrayLike<T>): T[]` <sup>[ES6]</sup>
- `Array.from<T,U>(iterable: Iterable<T> | ArrayLike<T>, mapFunc: (v: T, k: number) => U, thisArg?: any): U[]` <sup>[ES6]</sup>

Converts an iterable or an Array-like object to an Array. Optionally, the input values can be translated via a `mapFunc` before they are added to the output Array.

An Array-like object has a `.length` and `index` properties (roughly, string representations of non-negative integers):

```
interface ArrayLike<T> {
  length: number;
  [n: number]: T;
}
```

Examples:

```
> Array.from(new Set(['a', 'b']))
[ 'a', 'b' ]
> Array.from({length: 2, 0:'a', 1:'b'})
[ 'a', 'b' ]
```

- `Array.of<T>(...items: T[]): T[]` <sup>[ES6]</sup>

This static method is mainly useful for subclasses of Array and Typed Arrays, where it serves as a custom Array literal:

```
assert.ok(
  Uint8Array.of(1, 2, 3) instanceof Uint8Array);
```

### 28.12.3 Methods of `Array<T>.prototype`

- `.concat(...items: Array<T[] | T>): T[]` <sup>[R, ES3]</sup>

Returns a new Array that is the concatenation of the receiver and all `items`. Non-Array parameters are treated as if they were Arrays with single elements.

```
> ['a'].concat('b', ['c', 'd'])
[ 'a', 'b', 'c', 'd' ]
```

- `.copyWithin(target: number, start: number, end=this.length): this` <sup>[W, ES6]</sup>

Copies the elements whose indices range from `start` to (excl.) `end` to indices starting with `target`. Overlapping is handled correctly.

```
> ['a', 'b', 'c', 'd'].copyWithin(0, 2, 4)
[ 'c', 'd', 'c', 'd' ]
```

- `.entries(): Iterable<[number, T]>` <sup>[R, ES6]</sup>

Returns an iterable over `[index, element]` pairs.

```
> Array.from(['a', 'b'].entries())
[ [ 0, 'a' ], [ 1, 'b' ] ]
```

- `.every(callback: (value: T, index: number, array: Array<T>) => boolean, thisArg?: any): boolean` <sup>[R, ES5]</sup>

Returns `true` if `callback` returns `true` for every element and `false`, otherwise. Stops as soon as it receives `false`. This method corresponds to universal quantification (for all,  $\forall$ ) in mathematics.

```
> [1, 2, 3].every(x => x > 0)
true
> [1, -2, 3].every(x => x > 0)
false
```

- `.fill(value: T, start=0, end=this.length): this` <sup>[W, ES6]</sup>

Assigns `value` to every index between (incl.) `start` and (excl.) `end`.

```
> [0, 1, 2].fill('a')
[ 'a', 'a', 'a' ]
```

- `.filter(callback: (value: T, index: number, array: Array<T>) => any, thisArg?: any): T[]` <sup>[R, ES5]</sup>

Returns an `Array` with only those elements for which `callback` returns `true`.

```
> [1, -2, 3].filter(x => x > 0)
[ 1, 3 ]
```

- `.find(predicate: (value: T, index: number, obj: T[]) => boolean, thisArg?: any): T | undefined` <sup>[R, ES6]</sup>

The result is the first element for which `predicate` returns `true`. If it never does, the result is `undefined`.

```
> [1, -2, 3].find(x => x < 0)
-2
> [1, 2, 3].find(x => x < 0)
undefined
```

- `.findIndex(predicate: (value: T, index: number, obj: T[]) => boolean, thisArg?: any): number` <sup>[R, ES6]</sup>

The result is the index of the first element for which predicate returns true. If it never does, the result is -1.

```
> [1, -2, 3].findIndex(x => x < 0)
1
> [1, 2, 3].findIndex(x => x < 0)
-1
```

- `.forEach(callback: (value: T, index: number, array: Array<T>) => void, thisArg?: any): void` <sup>[R, ES5]</sup>

Calls callback for each element.

```
['a', 'b'].forEach((x, i) => console.log(x, i))
```

```
// Output:
// 'a', 0
// 'b', 1
```

- `.includes(searchElement: T, fromIndex=0): boolean` <sup>[R, ES2016]</sup>

Returns true if the receiver has an element whose value is searchElement and false, otherwise. Searching starts at index fromIndex.

```
> [0, 1, 2].includes(1)
true
> [0, 1, 2].includes(5)
false
```

- `.indexOf(searchElement: T, fromIndex=0): number` <sup>[R, ES5]</sup>

Returns the index of the first element that is strictly equal to searchElement. Returns -1 if there is no such element. Starts searching at index fromIndex, visiting higher indices next.

```
> ['a', 'b', 'a'].indexOf('a')
0
> ['a', 'b', 'a'].indexOf('a', 1)
2
> ['a', 'b', 'a'].indexOf('c')
-1
```

- `.join(separator = ','): string` <sup>[R, ES1]</sup>

Creates a string by concatenating string representations of all elements, separating them by separator.

```
> ['a', 'b', 'c'].join('##')
'a##b##c'
> ['a', 'b', 'c'].join()
'a,b,c'
```

- `.keys(): Iterable<number>` <sup>[R, ES6]</sup>

Returns an iterable over the keys of the receiver.

```
> [...['a', 'b'].keys()]
[ 0, 1 ]
```

- `.lastIndexOf(searchElement: T, fromIndex=this.length-1): number` <sup>[R, ES5]</sup>

Returns the index of the last element that is strictly equal to `searchElement`. Returns -1 if there is no such element. Starts searching at index `fromIndex`, visiting lower indices next.

```
> ['a', 'b', 'a'].lastIndexOf('a')
2
> ['a', 'b', 'a'].lastIndexOf('a', 1)
0
> ['a', 'b', 'a'].lastIndexOf('c')
-1
```

- `.map<U>(mapFunc: (value: T, index: number, array: Array<T>) => U, thisArg?: any): U[]` <sup>[R, ES5]</sup>

Returns a new Array, in which every element is the result of `mapFunc` being applied to the corresponding element of the receiver.

```
> [1, 2, 3].map(x => x * 2)
[ 2, 4, 6 ]
> ['a', 'b', 'c'].map((x, i) => i)
[ 0, 1, 2 ]
```

- `.pop(): T | undefined` <sup>[W, ES3]</sup>

Removes and returns the last element of the receiver. That is, it treats the end of the receiver as a stack. The opposite of `.push()`.

```
> const arr = ['a', 'b', 'c'];
> arr.pop()
'c'
> arr
[ 'a', 'b' ]
```

- `.push(...items: T[]): number` <sup>[W, ES3]</sup>

Adds zero or more items to the end of the receiver. That is, it treats the end of the receiver as a stack. The return value is the length of the receiver after the change. The opposite of `.pop()`.

```
> const arr = ['a', 'b'];
> arr.push('c', 'd')
4
> arr
[ 'a', 'b', 'c', 'd' ]
```

- `.reduce<U>(callback: (accumulator: U, element: T, index: number, array: T[]) => U, init?: U): U` <sup>[R, ES5]</sup>

This method produces a summary of the receiver: It feeds all Array elements to `callback`, which combines a current intermediate result (in parameter `accumulator`) with the current Array element and returns the next accumulator:

```
const accumulator_0 = callback(init, arr[0]);
const accumulator_1 = callback(accumulator_0, arr[1]);
const accumulator_2 = callback(accumulator_1, arr[2]);
// Etc.
```

The result of `.reduce()` is the last result of `callback`, after it has visited all Array elements.

If no `init` is provided, the Array element at index 0 is used, instead.

```
> [1, 2, 3].reduce((accu, x) => accu + x, 0)
6
> [1, 2, 3].reduce((accu, x) => accu + String(x), '')
'123'
```

- `.reduceRight<U>(callback: (accumulator: U, element: T, index: number, array: T[]) => U, init?: U): U` <sup>[R, ES5]</sup>

Works like `.reduce()`, but visits the Array elements backward, starting with the last element.

```
> [1, 2, 3].reduceRight((accu, x) => accu + String(x), '')
'321'
```

- `.reverse(): this` <sup>[W, ES1]</sup>

Rearranges the elements of the receiver so that they are in reverse order and then returns the receiver.

```
> const arr = ['a', 'b', 'c'];
> arr.reverse()
[ 'c', 'b', 'a' ]
> arr
[ 'c', 'b', 'a' ]
```

- `.shift(): T | undefined` <sup>[W, ES3]</sup>

Removes and returns the first element of the receiver. The opposite of `.unshift()`.

```
> const arr = ['a', 'b', 'c'];
> arr.shift()
'a'
> arr
[ 'b', 'c' ]
```

- `.slice(start=0, end=this.length): T[]` <sup>[R, ES3]</sup>

Returns a new Array, containing the elements of the receiver whose indices are between (incl.) `start` and (excl.) `end`.

```
> ['a', 'b', 'c', 'd'].slice(1, 3)
[ 'b', 'c' ]
> ['a', 'b'].slice() // shallow copy
[ 'a', 'b' ]
```

- `.some(callback: (value: T, index: number, array: Array<T>) => boolean, thisArg?: any): boolean` <sup>[R, ES5]</sup>

Returns `true` if `callback` returns `true` for at least one element and `false`, otherwise. Stops as soon as it receives `true`. This method corresponds to existential quantification (exists,  $\exists$ ) in mathematics.

```
> [1, 2, 3].some(x => x < 0)
false
> [1, -2, 3].some(x => x < 0)
```

true

- `.sort(compareFunc?: (a: T, b: T) => number): this` <sup>[W, ES1]</sup>

Sorts the receiver and returns it. By default, it sorts string representations of the elements. It does so lexicographically and according to the code unit values (char codes) of the characters:

```
> ['pie', 'cookie', 'éclair', 'Pie', 'Cookie', 'Éclair'].sort()
[ 'Cookie', 'Pie', 'cookie', 'pie', 'éclair', 'éclair' ]
> [200, 3, 10].sort()
[ 10, 200, 3 ]
```

You can customize the sort order via `compareFunc`, which returns a number that is:

- negative if `a < b`
- zero if `a === b`
- positive if `a > b`

Hack (with a risk of numeric overflow):

```
> [200, 3, 10].sort((a, b) => a - b)
[ 3, 10, 200 ]
```

- `.splice(start: number, deleteCount=this.length-start, ...items: T[]): T[]` <sup>[W, ES3]</sup>

At index `start`, it removes `deleteCount` elements and inserts the items. It returns the deleted elements.

```
> const arr = ['a', 'b', 'c', 'd'];
> arr.splice(1, 2, 'x', 'y')
[ 'b', 'c' ]
> arr
[ 'a', 'x', 'y', 'd' ]
```

- `.toString(): string` <sup>[R, ES1]</sup>

Returns a string with the stringifications of all elements, separated by commas.

```
> [1, 2, 3].toString()
'1,2,3'
> ['a', 'b', 'c'].toString()
'a,b,c'
> [].toString()
''
```

- `.unshift(...items: T[]): number` <sup>[W, ES3]</sup>

Inserts the items at the beginning of the receiver and returns its length after this modification.

```
> const arr = ['c', 'd'];
> arr.unshift('e', 'f')
4
> arr
[ 'e', 'f', 'c', 'd' ]
```

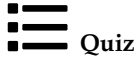
- `.values(): Iterable<T>` <sup>[R, ES6]</sup>

Returns an iterable over the values of the receiver.

```
> [...['a', 'b']].values()  
[ 'a', 'b' ]
```

#### 28.12.4 Sources

- TypeScript's built-in typings<sup>3</sup>
- MDN web docs for JavaScript<sup>4</sup>
- ECMAScript language specification<sup>5</sup>



Quiz

See quiz app.

---

<sup>3</sup><https://github.com/Microsoft/TypeScript/blob/master/lib/>

<sup>4</sup><https://developer.mozilla.org/en-US/docs/Web/JavaScript>

<sup>5</sup><https://tc39.github.io/ecma262/>



## Chapter 29

# Typed Arrays: handling binary data (Advanced)

### Contents

---

<b>29.1 The basics of the API</b>	<b>290</b>
29.1.1 Use cases for Typed Arrays	290
29.1.2 Browser APIs that support Typed Arrays	290
29.1.3 The core classes: <code>ArrayBuffer</code> , Typed Arrays, <code>DataView</code>	291
29.1.4 Using Typed Arrays	291
29.1.5 Using <code>DataViews</code>	293
<b>29.2 Foundations of the Typed Array API</b>	<b>293</b>
29.2.1 Element types	293
29.2.2 Handling overflow and underflow	294
29.2.3 Endianness	295
29.2.4 Indices and offsets	296
<b>29.3 <code>ArrayBuffers</code></b>	<b>296</b>
29.3.1 <code>new ArrayBuffer()</code>	296
29.3.2 Static methods of <code>ArrayBuffer</code>	297
29.3.3 Properties of <code>ArrayBuffer.prototype</code>	297
<b>29.4 Typed Arrays</b>	<b>297</b>
29.4.1 Typed Arrays versus normal Arrays	297
29.4.2 Typed Arrays are iterable	298
29.4.3 Converting Typed Arrays to and from normal Arrays	298
29.4.4 The class hierarchy of Typed Arrays	298
29.4.5 Static methods of <code>TypedArray</code>	299
29.4.6 Properties of <code>TypedArray&lt;T&gt;.prototype</code>	300
29.4.7 <code>new «ElementType»Array()</code>	302
29.4.8 Static properties of <code>«ElementType»Array</code>	303
29.4.9 Properties of <code>«ElementType»Array.prototype</code>	303
29.4.10 Concatenating Typed Arrays	303
<b>29.5 <code>DataViews</code></b>	<b>304</b>

29.5.1 <code>new DataView()</code> . . . . .	304
29.5.2 Properties of <code>DataView.prototype</code> . . . . .	304
29.6 Further reading . . . . .	305

---

## 29.1 The basics of the API

Much data on the web is text: JSON files, HTML files, CSS files, JavaScript code, etc. JavaScript handles such data well, via its built-in strings.

However, before 2011, it did not handle binary data well. The Typed Array Specification 1.0<sup>1</sup> was introduced on 8 February 2011 and provides tools for working with binary data. With ECMAScript 6, Typed Arrays were added to the core language and gained methods that were previously only available for normal Arrays (`.map()`, `.filter()`, etc.).

### 29.1.1 Use cases for Typed Arrays

The main uses cases for Typed Arrays are:

- Processing binary data: manipulating image data in HTML Canvas elements, parsing binary files, handling binary network protocols, etc.
- Interacting with native APIs: Native APIs often receive and return data in a binary format, which you could neither store nor manipulate well in pre-ES6 JavaScript. That meant that, whenever you were communicating with such an API, data had to be converted from JavaScript to binary and back, for every call. Typed Arrays eliminate this bottleneck. One example of communicating with native APIs is WebGL, for which Typed Arrays were initially created. Section “History of Typed Arrays<sup>2</sup>” of the article “Typed Arrays: Binary Data in the Browser<sup>3</sup>” (by Ilmari Heikkinen for HTML5 Rocks) has more information.

### 29.1.2 Browser APIs that support Typed Arrays

The following browser APIs support Typed Arrays:

- File API<sup>4</sup>
- XMLHttpRequest<sup>5</sup>
- Fetch API<sup>6</sup>
- Canvas<sup>7</sup>
- WebSockets<sup>8</sup>
- WebGL<sup>9</sup>

---

<sup>1</sup><https://www.khronos.org/registry/typedarray/specs/1.0/>

<sup>2</sup>[http://www.html5rocks.com/en/tutorials/webgl/typed\\_arrays/#toc-history](http://www.html5rocks.com/en/tutorials/webgl/typed_arrays/#toc-history)

<sup>3</sup>[http://www.html5rocks.com/en/tutorials/webgl/typed\\_arrays/#toc-history](http://www.html5rocks.com/en/tutorials/webgl/typed_arrays/#toc-history)

<sup>4</sup><http://www.w3.org/TR/FileAPI/>

<sup>5</sup><http://www.w3.org/TR/XMLHttpRequest/>

<sup>6</sup><https://fetch.spec.whatwg.org/>

<sup>7</sup><http://www.w3.org/TR/2dcontext/>

<sup>8</sup><http://www.w3.org/TR/websockets/>

<sup>9</sup><https://www.khronos.org/registry/webgl/>

- Web Audio API<sup>10</sup>
- (And more)

### 29.1.3 The core classes: `ArrayBuffer`, `Typed Arrays`, `DataView`

The Typed Array API stores binary data in instances of `ArrayBuffer`:

```
const buf = new ArrayBuffer(4); // length in bytes
// buf is initialized with zeros
```

An `ArrayBuffer` itself is opaque. If you want to access its data, you must wrap it in another object – a *view object*. Two kinds of view objects are available:

- **Typed Arrays:** let you access the data as an indexed sequence of elements that all have the same type. Examples include:
  - `Uint8Array`: Elements are unsigned 8-bit integers. *Unsigned* means that their ranges start at zero.
  - `Int16Array`: Elements are signed 16-bit integers. *Signed* means that they have a sign and can be negative, zero, or positive.
  - `Float32Array`: Elements are 32-bit floating point numbers.
- **DataViews:** let you interpret the data as various types (`Uint8`, `Int16`, `Float32`, etc.) that you can read and write at any byte offset.

Fig. 29.1 shows a class diagram of the API.

### 29.1.4 Using Typed Arrays

Typed Arrays are used much like normal Arrays, with a few notable differences:

- Typed Arrays store their data in `ArrayBuffers`.
- All elements are initialized with zeros.
- All elements have the same type. Writing values to a Typed Array coerces them to that type. Reading values produces normal numbers.
- Once a Typed Array is created, its length never changes.
- Typed Arrays can't have holes.

This is an example of using a Typed Array:

```
const typedArray = new Uint8Array(2); // 2 elements
assert.equal(typedArray.length, 2);

// The wrapped ArrayBuffer
assert.deepEqual(
  typedArray.buffer, new ArrayBuffer(2)); // 2 bytes

// Getting and setting elements:
assert.equal(typedArray[1], 0); // initialized with 0
typedArray[1] = 72;
assert.equal(typedArray[1], 72);
```

---

<sup>10</sup><http://www.w3.org/TR/webaudio/>

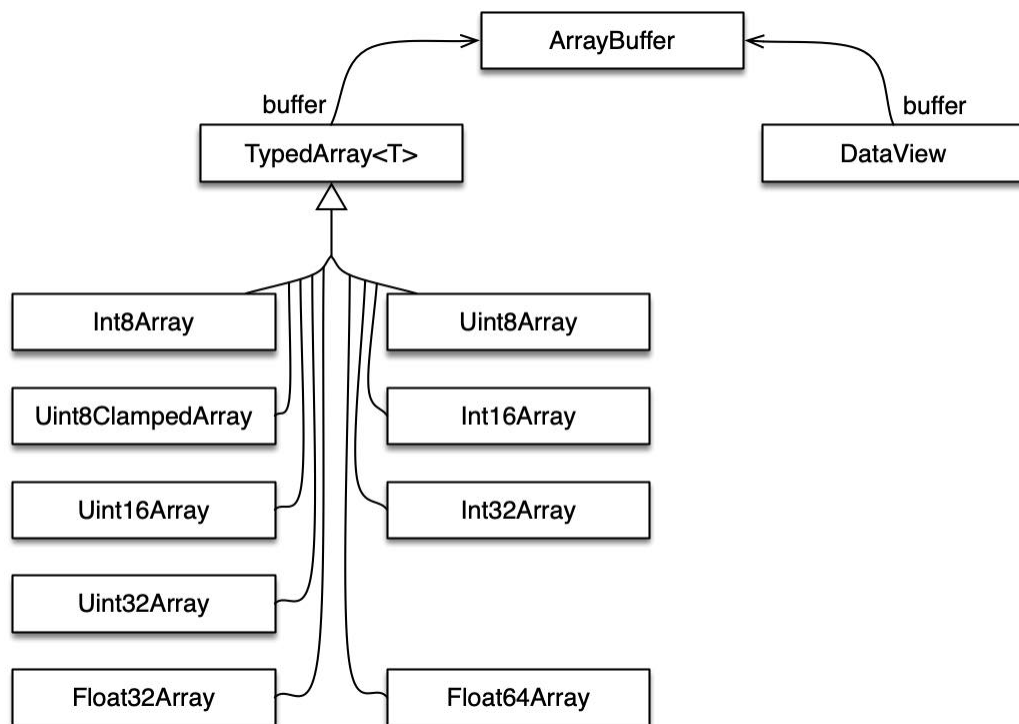


Figure 29.1: The classes of the Typed Array API.

Other ways of creating Typed Arrays:

```
const ta1 = new Uint8Array([5, 6]);
const ta2 = Uint8Array.of(5, 6);
assert.deepEqual(ta1, ta2);
```

### 29.1.5 Using DataViews

This is how DataViews are used:

```
const dataView = new DataView(new ArrayBuffer(4));
assert.equal(dataView.getUint16(0), 0);
assert.equal(dataView.getUint8(0), 0);
dataView.setUint8(0, 5);
```

## 29.2 Foundations of the Typed Array API

### 29.2.1 Element types

The following element types are supported by the API:

Element type	Bytes	Description	C type
Int8	1	8-bit signed integer	signed char
Uint8	1	8-bit unsigned integer	unsigned char
Uint8C	1	8-bit unsigned integer (clamped conversion)	unsigned char
Int16	2	16-bit signed integer	short
Uint16	2	16-bit unsigned integer	unsigned short
Int32	4	32-bit signed integer	int
Uint32	4	32-bit unsigned integer	unsigned int
Float32	4	32-bit floating point	float
Float64	8	64-bit floating point	double

The element type `Uint8C` is special: it is not supported by `DataView` and only exists to enable `Uint8ClampedArray`. This Typed Array is used by the canvas element (where it replaces `CanvasPixelArray`). The only difference between `Uint8C` and `Uint8` is how overflow and underflow are handled (as explained in the next section). It is recommended to avoid the former – quoting Brendan Eich<sup>11</sup>:

Just to be super-clear (and I was around when it was born), `Uint8ClampedArray` is *totally* a historical artifact (of the HTML5 canvas element). Avoid unless you really are doing canvas-y things.

<sup>11</sup><https://mail.mozilla.org/pipermail/es-discuss/2015-August/043902.html>

### 29.2.2 Handling overflow and underflow

Normally, when a value is out of the range of the element type, modulo arithmetic is used to convert it to a value within range. For signed and unsigned integers that means that:

- The highest value plus one is converted to the lowest value (0 for unsigned integers).
- The lowest value minus one is converted to the highest value.

The following functions helps illustrate how conversion works:

```
function setAndGet(typedArray, value) {
  typedArray[0] = value;
  return typedArray[0];
}
```

Modulo conversion for unsigned 8-bit integers:

```
const uint8 = new Uint8Array(1);

// Highest value of range
assert.equal(setAndGet(uint8, 255), 255);
// Overflow
assert.equal(setAndGet(uint8, 256), 0);

// Lowest value of range
assert.equal(setAndGet(uint8, 0), 0);
// Underflow
assert.equal(setAndGet(uint8, -1), 255);
```

Modulo conversion for signed 8-bit integers:

```
const int8 = new Int8Array(1);

// Highest value of range
assert.equal(setAndGet(int8, 127), 127);
// Overflow
assert.equal(setAndGet(int8, 128), -128);

// Lowest value of range
assert.equal(setAndGet(int8, -128), -128);
// Underflow
assert.equal(setAndGet(int8, -129), 127);
```

Clamped conversion is different:

- All underflowing values are converted to the lowest value.
- All overflowing values are converted to the highest value.

```
const uint8c = new Uint8ClampedArray(1);

// Highest value of range
assert.equal(setAndGet(uint8c, 255), 255);
// Overflow
assert.equal(setAndGet(uint8c, 256), 255);
```

```
// Lowest value of range
assert.equal(setAndGet(uint8c, 0), 0);
// Underflow
assert.equal(setAndGet(uint8c, -1), 0);
```

### 29.2.3 Endianness

Whenever a type (such as `Uint16`) is stored as a sequence of multiple bytes, *endianness* matters:

- Big endian: the most significant byte comes first. For example, the `Uint16` value 0x4321 is stored as two bytes – first 0x43, then 0x21.
- Little endian: the least significant byte comes first. For example, the `Uint16` value 0x4321 is stored as two bytes – first 0x21, then 0x43.

Endianness tends to be fixed per CPU architecture and consistent across native APIs. Typed Arrays are used to communicate with those APIs, which is why their endianness follows the endianness of the platform and can't be changed.

On the other hand, the endianness of protocols and binary files varies and is fixed across platforms. Therefore, we must be able to access data with either endianness. `DataViews` serve this use case and let you specify endianness when you get or set a value.

Quoting Wikipedia on Endianness<sup>12</sup>:

- Big-endian representation is the most common convention in data networking; fields in the protocols of the Internet protocol suite, such as IPv4, IPv6, TCP, and UDP, are transmitted in big-endian order. For this reason, big-endian byte order is also referred to as network byte order.
- Little-endian storage is popular for microprocessors in part due to significant historical influence on microprocessor designs by Intel Corporation.

You can use the following function to determine the endianness of a platform.

```
const BIG_ENDIAN = Symbol('BIG_ENDIAN');
const LITTLE_ENDIAN = Symbol('LITTLE_ENDIAN');
function getPlatformEndianness() {
  const arr32 = Uint32Array.of(0x87654321);
  const arr8 = new Uint8Array(arr32.buffer);
  if (compare(arr8, [0x87, 0x65, 0x43, 0x21])) {
    return BIG_ENDIAN;
  } else if (compare(arr8, [0x21, 0x43, 0x65, 0x87])) {
    return LITTLE_ENDIAN;
  } else {
    throw new Error('Unknown endianness');
  }
}
function compare(arr1, arr2) {
  if (arr1.length !== arr2.length) {
    return false;
  }
}
```

---

<sup>12</sup><https://en.wikipedia.org/wiki/Endianness>

```

for (let i=0; i<arr1.length; i++) {
  if (arr1[i] !== arr2[i]) {
    return false;
  }
}
return true;
}

```

Other orderings are also possible. Those are generically called *middle-endian* or *mixed-endian*.

### 29.2.4 Indices and offsets

For Typed Arrays, we distinguish:

- Indices for the bracket operator [ ]: You can only use non-negative indices (starting at 0).
- Indices for methods of ArrayBuffers, Typed Arrays and DataViews: Every index can be negative. If it is, it is added to the length of the entity, to produce the actual index. Therefore -1 refers to the last element, -2 to the second-last, etc. Methods of normal Arrays work the same way.

```

const ui8 = Uint8Array.of(0, 1, 2);
assert.deepEqual(ui8.slice(-1), Uint8Array.of(2));

```

- Offsets passed to methods of Typed Arrays and DataViews: must be non-negative. For example:

```

const dataView = new DataView(new ArrayBuffer(4));
assert.throws(
  () => dataView.getUint8(-1),
  {
    name: 'RangeError',
    message: 'Offset is outside the bounds of the DataView',
  });

```

## 29.3 ArrayBuffers

ArrayBuffers store binary data, which is meant to be accessed via Typed Arrays and DataViews.

### 29.3.1 new ArrayBuffer()

The type signature of the constructor is:

```
new ArrayBuffer(length: number)
```

Invoking this constructor via new creates an instance whose capacity is length bytes. Each of those bytes is initially 0.

You can't change the length of an ArrayBuffer, you can only create a new one with a different length.



### 29.3.2 Static methods of ArrayBuffer

- `ArrayBuffer.isView(arg: any)`  
Returns true if `arg` is an object and a view for an `ArrayBuffer` (a `Typed Array` or a `DataView`).

### 29.3.3 Properties of `ArrayBuffer.prototype`

- `get .byteLength(): number`  
Returns the capacity of this `ArrayBuffer` in bytes.
- `.slice(startIndex: number, endIndex=this.byteLength)`  
Creates a new `ArrayBuffer` that contains the bytes of this `ArrayBuffer` whose indices are greater than or equal to `startIndex` and less than `endIndex`. `start` and `endIndex` can be negative (see Sect. “[Indices and offsets](#)”).

## 29.4 Typed Arrays

The various kinds of Typed Arrays are only different w.r.t. the types of their elements:

- Typed Arrays whose elements are integers: `Int8Array`, `Uint8Array`, `Uint8ClampedArray`, `Int16Array`, `Uint16Array`, `Int32Array`, `Uint32Array`
- Typed Arrays whose elements are floats: `Float32Array`, `Float64Array`

### 29.4.1 Typed Arrays versus normal Arrays

Typed Arrays are much like normal Arrays: they have a `.length`, elements can be accessed via the bracket operator `[ ]` and they have most of the standard Array methods. They differ from normal Arrays in the following ways:

- Typed Arrays have buffers. The elements of a Typed Array `ta` are not stored in `ta`, they are stored in an associated `ArrayBuffer` that can be accessed via `ta.buffer`:

```
const ta = new Uint8Array(4); // 4 elements
assert.deepEqual(
  ta.buffer, new ArrayBuffer(4)); // 4 bytes
```

- Typed Arrays are initialized with zeros:
  - `new Array(4)` creates a normal Array without any elements. It only has 4 *holes* (indices less than the `.length` that have no associated elements).
  - `new Uint8Array(4)` creates a Typed Array whose 4 elements are all 0.

```
assert.deepEqual(new Uint8Array(4), Uint8Array.of(0, 0, 0, 0));
```

- All of the elements of a Typed Array have the same type:
  - Setting elements converts values to that type.

```
const ta = new Uint8Array(1);
```

```
ta[0] = 256;
assert.equal(ta[0], 0);
```

```
ta[0] = '2';
assert.equal(ta[0], 2);
```

- Getting elements returns numbers.

```
const ta = new Uint8Array(1);
assert.equal(ta[0], 0);
assert.equal(typeof ta[0], 'number');
```

- The `.length` of a Typed Array is derived from its `ArrayBuffer` and never changes (unless you switch to a different `ArrayBuffer`).
- Normal Arrays can have holes; Typed Arrays can't.

### 29.4.2 Typed Arrays are iterable

Typed Arrays are **iterable**. That means that you can use the `for-of` loop and similar mechanisms:

```
const ui8 = Uint8Array.of(0,1,2);
for (const byte of ui8) {
  console.log(byte);
}
// Output:
// 0
// 1
// 2
```

`ArrayBuffers` and `DataViews` are not iterable.

### 29.4.3 Converting Typed Arrays to and from normal Arrays

To convert a normal Array to a Typed Array, you pass it to a Typed Array constructor. For example:

```
const tarr = new Uint8Array([0,1,2]);
```

To convert a Typed Array to a normal Array, you can use spreading or `Array.from()` (because Typed Arrays are iterable):

```
assert.deepEqual([...tarr], [0,1,2]);
assert.deepEqual(Array.from(tarr), [0,1,2]);
```

### 29.4.4 The class hierarchy of Typed Arrays

The properties of the various Typed Array objects are introduced in two steps:

1. `TypedArray`: First, we look at the common superclass of all Typed Array classes (which was shown in the class diagram [at the beginning of this chapter](#)). I'm calling that superclass `TypedArray`, but it is not directly accessible from JavaScript. `TypedArray.prototype` houses all methods of Typed Arrays.
2. «ElementType»Array: The actual Typed Array classes are called `Uint8Array`, `Int16Array`, `Float32Array`, etc.

### 29.4.5 Static methods of `TypedArray`

Both static `TypedArray` methods are inherited by its subclasses (`Uint8Array` etc.).

#### 29.4.5.1 `TypedArray.of()`

This method has the type signature:

```
.of(...items: number[]): instanceof this
```

The notation of the return type is my invention: `.of()` returns an instance of `this` (the class on which `of()` was invoked). The elements of the instance are the arguments of `of()`.

You can think of `of()` as a custom literal for Typed Arrays:

```
const float32Arr = Float32Array.of(0.151, -8, 3.7);
const int16Arr = Int32Array.of(-10, 5, 7);
```

#### 29.4.5.2 `TypedArray.from()`

This method has the type signature:

```
TypedArray<T>.from<S>(
  source: Iterable<S>|ArrayLike<S>, mapfn?: S => T, thisArg?: any)
: instanceof this
```

It converts `source` into an instance of `this` (a Typed Array). Once again, the syntax `instanceof this` is my invention.

For example, normal Arrays are iterable and can be converted with this method:

```
assert.deepEqual(
  Uint16Array.from([0, 1, 2]),
  Uint16Array.of(0, 1, 2));
```

Typed Arrays are also iterable:

```
assert.deepEqual(
  Uint16Array.from(Uint8Array.of(0, 1, 2)),
  Uint16Array.of(0, 1, 2));
```

The source can also be an [Array-like object](#):

```
assert.deepEqual(
  Uint16Array.from({0:0, 1:1, 2:2, length: 3}),
  Uint16Array.of(0, 1, 2));
```

The optional `mapfn` lets you transform the elements of `source` before they become elements of the result. Why perform the two steps *mapping* and *conversion* in one go? Compared to mapping separately via `.map()`, there are two advantages:

1. No intermediate Array or Typed Array is needed.
2. When converting between Typed Arrays with different precisions, less can go wrong.

To illustrate the second advantage, let's first convert a Typed Array to a Typed Array with a higher precision. If we use `.from()` to `map`, the result is automatically correct. Otherwise, you must first convert and then `map`.

```
const typedArray = Int8Array.of(127, 126, 125);
assert.deepEqual(
  Int16Array.from(typedArray, x => x * 2),
  Int16Array.of(254, 252, 250));

assert.deepEqual(
  Int16Array.from(typedArray).map(x => x * 2),
  Int16Array.of(254, 252, 250)); // OK
assert.deepEqual(
  Int16Array.from(typedArray.map(x => x * 2)),
  Int16Array.of(-2, -4, -6)); // wrong
```

If we go from a Typed Array to a Typed Array with a lower precision, mapping via `.from()` produces the correct result. Otherwise, we must first `map` and then convert.

```
assert.deepEqual(
  Int8Array.from(Int16Array.of(254, 252, 250), x => x / 2),
  Int8Array.of(127, 126, 125));

assert.deepEqual(
  Int8Array.from(Int16Array.of(254, 252, 250).map(x => x / 2)),
  Int8Array.of(127, 126, 125)); // OK
assert.deepEqual(
  Int8Array.from(Int16Array.of(254, 252, 250)).map(x => x / 2),
  Int8Array.of(-1, -2, -3)); // wrong
```

The problem is that, if we `map` via `.map()`, then input type and output type are the same (if we work with Typed Arrays). In contrast, `.from()` goes from an arbitrary input type to an output type that you specify via its receiver.

According to Allen Wirfs-Brock<sup>13</sup>, mapping between Typed Arrays was what motivated the `mapfn` parameter of `.from()`.

#### 29.4.6 Properties of `TypedArray<T>.prototype`

Indices accepted by Typed Array methods can be negative (they work like traditional Array methods that way). Offsets must be non-negative. For details, see Sect. “[Indices and offsets](#)”.

<sup>13</sup><https://twitter.com/awbjs/status/585199958661472257>

### 29.4.6.1 Properties specific to Typed Arrays

The following properties are specific to Typed Arrays; normal Arrays don't have them:

- `get .buffer(): ArrayBuffer`  
Returns the buffer backing this Typed Array.
- `get .length(): number`  
Returns the length in elements of this Typed Array's buffer. Note that the length of normal Arrays is not a getter, it is a special property that instances have.
- `get .byteLength(): number`  
Returns the size in bytes of this Typed Array's buffer.
- `get .byteOffset(): number`  
Returns the offset where this Typed Array "starts" inside its ArrayBuffer.
- `.set(arrayLike: ArrayLike<number>, offset=0): void`
- `.set(typedArray: TypedArray, offset=0): void` Copies all elements of the first parameter to this Typed Array. The element at index 0 of the parameter is written to index `offset` of this Typed Array (etc.).
  - First parameter is `arrayLike`: its elements are converted to numbers, which are then converted to the element type `T` of this Typed Array.
  - First parameter is `typedArray`: each of its elements is converted directly to the appropriate type for this Typed Array. If both Typed Arrays have the same element type then faster, byte-wise copying is used.
- `.subarray(startIndex=0, end=this.length): TypedArray<T>`  
Returns a new Typed Array that has the same buffer as this Typed Array, but a (generally) smaller range. If `startIndex` is non-negative then the first element of the resulting Typed Array is `this[startIndex]`, the second `this[startIndex+1]` (etc.). If `startIndex` is negative, it is converted appropriately.

### 29.4.6.2 Array methods

The following methods are basically the same as the methods of normal Arrays:

- `.copyWithin(target: number, start: number, end=this.length): this` <sup>[W, ES6]</sup>
- `.entries(): Iterable<[number, T]>` <sup>[R, ES6]</sup>
- `.every(callback: (value: T, index: number, array: TypedArray<T>) => boolean, thisArg?: any): boolean` <sup>[R, ES5]</sup>
- `.fill(value: T, start=0, end=this.length): this` <sup>[W, ES6]</sup>
- `.filter(callback: (value: T, index: number, array: TypedArray<T>) => any, thisArg?: any): T[]` <sup>[R, ES5]</sup>
- `.find(predicate: (value: T, index: number, obj: T[]) => boolean, thisArg?: any): T | undefined` <sup>[R, ES6]</sup>
- `.findIndex(predicate: (value: T, index: number, obj: T[]) => boolean, thisArg?: any): number` <sup>[R, ES6]</sup>

- `.forEach(callback: (value: T, index: number, array: TypedArray<T>) => void, thisArg?: any): void` <sup>[R, ES5]</sup>
- `.includes(searchElement: T, fromIndex=0): boolean` <sup>[R, ES2016]</sup>
- `.indexOf(searchElement: T, fromIndex=0): number` <sup>[R, ES5]</sup>
- `.join(separator = ','): string` <sup>[R, ES1]</sup>
- `.keys(): Iterable<number>` <sup>[R, ES6]</sup>
- `.lastIndexOf(searchElement: T, fromIndex=this.length-1): number` <sup>[R, ES5]</sup>
- `.map<U>(mapFunc: (value: T, index: number, array: TypedArray<T>) => U, thisArg?: any): U[]` <sup>[R, ES5]</sup>
- `.reduce<U>(callback: (accumulator: U, element: T, index: number, array: T[]) => U, init?: U): U` <sup>[R, ES5]</sup>
- `.reduceRight<U>(callback: (accumulator: U, element: T, index: number, array: T[]) => U, init?: U): U` <sup>[R, ES5]</sup>
- `.reverse(): this` <sup>[W, ES1]</sup>
- `.slice(start=0, end=this.length): T[]` <sup>[R, ES3]</sup>
- `.some(callback: (value: T, index: number, array: TypedArray<T>) => boolean, thisArg?: any): boolean` <sup>[R, ES5]</sup>
- `.sort(compareFunc?: (a: T, b: T) => number): this` <sup>[W, ES1]</sup>
- `.toString(): string` <sup>[R, ES1]</sup>
- `.values(): Iterable<number>` <sup>[R, ES6]</sup>

For details on how these methods work, please consult [the chapter on normal Arrays](#).

#### 29.4.7 new «ElementType»Array()

Each Typed Array constructor has a name that follows the pattern «ElementType»Array, where «ElementType» is one of the element types in the table at the beginning. That means that there are 9 constructors for Typed Arrays: `Int8Array`, `Uint8Array`, `Uint8ClampedArray` (element type `Uint8C`), `Int16Array`, `Uint16Array`, `Int32Array`, `Uint32Array`, `Float32Array`, `Float64Array`.

Each constructor has four *overloaded* versions – it behaves differently depending on how many arguments it receives and what their types are:

- `new «ElementType»Array(buffer: ArrayBuffer, byteOffset=0, length=0)`  
Creates a new «ElementType»Array whose buffer is `buffer`. It starts accessing the buffer at the given `byteOffset` and will have the given `length`. Note that `length` counts elements of the Typed Array (with 1–4 bytes each), not bytes.
- `new «ElementType»Array(length=0)`  
Creates a new «ElementType»Array with the given `length` and the appropriate buffer (whose size in bytes is `length * «ElementType»Array.BYTES_PER_ELEMENT`).
- `new «ElementType»Array(source: TypedArray)`  
Creates a new instance of «ElementType»Array whose elements have the same values as the elements of `source`, but coerced to `ElementType`.
- `new «ElementType»Array(source: ArrayLike<number>)`  
Creates a new instance of «ElementType»Array whose elements have the same values as the elements of `source`, but coerced to `ElementType`. (For more information on Array-like objects, consult

the chapter on Arrays.)

The following code shows three different ways of creating the same Typed Array:

```
const ta1 = new Uint8Array([0, 1, 2]);

const ta2 = Uint8Array.of(0, 1, 2);

const ta3 = new Uint8Array(3);
ta3[0] = 0;
ta3[1] = 1;
ta3[2] = 2;

assert.deepEqual(ta1, ta2);
assert.deepEqual(ta1, ta3);
```

### 29.4.8 Static properties of «ElementType»Array

- «ElementType»Array.BYTES\_PER\_ELEMENT: number  
Counts how many bytes are needed to store a single element:

```
> Uint8Array.BYTES_PER_ELEMENT
1
> Int16Array.BYTES_PER_ELEMENT
2
> Float64Array.BYTES_PER_ELEMENT
8
```

### 29.4.9 Properties of «ElementType»Array.prototype

- .BYTES\_PER\_ELEMENT: number  
The same as «ElementType»Array.BYTES\_PER\_ELEMENT.

### 29.4.10 Concatenating Typed Arrays

Typed Arrays don't have a method `.concat()`, like normal Arrays do. The work-around is to use the method

```
.set(typedArray: TypedArray, offset=0): void
```

That method copies an existing Typed Array into `typedArray` at index `offset`. Then you only have to make sure that `typedArray` is big enough to hold all (Typed) Arrays you want to concatenate:

```
function concatenate(resultConstructor, ...arrays) {
  let totalLength = 0;
  for (const arr of arrays) {
    totalLength += arr.length;
  }
}
```

```

const result = new resultConstructor(totalLength);
let offset = 0;
for (const arr of arrays) {
  result.set(arr, offset);
  offset += arr.length;
}
return result;
}
assert.deepEqual(
  concatenate(UInt8Array,
    UInt8Array.of(1, 2), [3, 4]),
    UInt8Array.of(1, 2, 3, 4)
);

```

## 29.5 DataViews

### 29.5.1 new DataView()

- `new DataView(buffer: ArrayBuffer, byteOffset=0, byteLength=buffer.byteLength-byteOffset)`

Creates a new `DataView` whose data is stored in the `ArrayBuffer` `buffer`. By default, the new `DataView` can access all of `buffer`, the last two parameters allow you to change that.

### 29.5.2 Properties of `DataView.prototype`

«`ElementType`» can be: `Float32`, `Float64`, `Int8`, `Int16`, `Int32`, `Uint8`, `Uint16`, `Uint32`.

- `get .buffer()`  
Returns the `ArrayBuffer` of this `DataView`.
- `get .byteLength()`  
Returns how many bytes can be accessed by this `DataView`.
- `get .byteOffset()`  
Returns at which offset this `DataView` starts accessing the bytes in its buffer.
- `.get«ElementType»(byteOffset: number, littleEndian=false)`  
Reads a value from the buffer of this `DataView`.
- `.set«ElementType»(byteOffset: number, value: number, littleEndian=false)`  
Writes `value` to the buffer of this `DataView`.



## 29.6 Further reading

The chapter on Typed Arrays<sup>14</sup> in “Exploring ES6” has some additional content:

- More details on browser APIs that support Typed Arrays
- A real-world example
- And a few more technical details.

---

<sup>14</sup>[http://exploringjs.com/es6/ch\\_typed-arrays.html](http://exploringjs.com/es6/ch_typed-arrays.html)



## Chapter 30

# Maps (Map)

### Contents

---

<b>30.1 Using Maps</b>	<b>308</b>
30.1.1 Creating Maps	308
30.1.2 Working with single entries	308
30.1.3 Determining the size of a Map and clearing it	309
30.1.4 Getting the keys and values of a Map	309
30.1.5 Getting the entries of a Map	309
<b>30.2 Example: Counting characters</b>	<b>310</b>
<b>30.3 A few more details about the keys of Maps (advanced)</b>	<b>310</b>
30.3.1 What keys are considered equal?	311
<b>30.4 Missing Map operations</b>	<b>311</b>
30.4.1 Mapping and filtering Maps	311
30.4.2 Combining Maps	312
<b>30.5 Quick reference: Map&lt;K, V&gt;</b>	<b>313</b>
30.5.1 Constructor	313
30.5.2 Handling single entries	313
30.5.3 Handling all entries	314
30.5.4 Iterating and looping	314
30.5.5 Sources	315
<b>30.6 FAQ</b>	<b>316</b>
30.6.1 When should I use a Map, when an object?	316
30.6.2 When would I use an object as a key in a Map?	316

---

Before ES6, JavaScript didn't have a data structure for dictionaries and (ab)used objects as dictionaries from strings to arbitrary values. ES6 brought Maps, which are dictionaries from arbitrary values to arbitrary values.

## 30.1 Using Maps

An instance of `Map` maps keys to values. A single key-value mapping is called an *entry*. Maps record in which order entries were created and honor that order when returning, e.g., keys or entries.

### 30.1.1 Creating Maps

There are three common ways of creating Maps.

First, you can use the constructor without any parameters to create an empty Map:

```
const emptyMap = new Map();
assert.equal(emptyMap.size, 0);
```

Second, you can pass an iterable (e.g. an Array) over key-value “pairs” (Arrays with 2 elements) to the constructor:

```
const map = new Map([
  [ 1, 'one' ],
  [ 2, 'two' ],
  [ 3, 'three' ], // trailing comma is ignored
]);
```

Third, the `.set()` method adds entries to a Map and is chainable:

```
const map = new Map()
  .set(1, 'one')
  .set(2, 'two')
  .set(3, 'three');
```

### 30.1.2 Working with single entries

`.set()` and `.get()` are for writing and reading values (given keys).

```
const map = new Map();

map.set('foo', 123);

assert.equal(map.get('foo'), 123);
// Unknown key:
assert.equal(map.get('bar'), undefined);
// Use the default value '' if an entry is missing:
assert.equal(map.get('bar') || '', '');
```

`.has()` checks if a Map has an entry with a given key. `.delete()` removes entries.

```
const map = new Map([['foo', 123]]);

assert.equal(map.has('foo'), true);
assert.equal(map.delete('foo'), true)
assert.equal(map.has('foo'), false)
```

### 30.1.3 Determining the size of a Map and clearing it

`.size` contains the number of entries in a Map. `.clear()` removes all entries of a Map.

```
const map = new Map()
  .set('foo', true)
  .set('bar', false)
;

assert.equal(map.size, 2)
map.clear();
assert.equal(map.size, 0)
```

### 30.1.4 Getting the keys and values of a Map

`.keys()` returns an iterable over the keys of a Map:

```
const map = new Map()
  .set(false, 'no')
  .set(true, 'yes')
;

for (const key of map.keys()) {
  console.log(key);
}
// Output:
// false
// true
```

We can use `spreading (...)` to convert the iterable returned by `.keys()` to an Array:

```
assert.deepEqual(
  [...map.keys()],
  [false, true]);
```

`.values()` works like `.keys()`, but for values instead of keys.

### 30.1.5 Getting the entries of a Map

`.entries()` returns an iterable over the entries of a Map:

```
const map = new Map()
  .set(false, 'no')
  .set(true, 'yes')
;

for (const entry of map.entries()) {
  console.log(entry);
}
// Output:
```

```
// [false, 'no']
// [true, 'yes']
```

Spreading (...) converts the iterable returned by `.entries()` to an Array:

```
assert.deepEqual(
  [...map.entries()],
  [[false, 'no'], [true, 'yes']]);
```

Map instances are also iterables over entries. In the following code, we use destructuring to access the keys and values of map:

```
for (const [key, value] of map) {
  console.log(key, value);
}
// Output:
// false, 'no'
// true, 'yes'
```

## 30.2 Example: Counting characters

`countChars()` returns a Map that maps characters to numbers of occurrences.

```
function countChars(chars) {
  const charCounts = new Map();
  for (let ch of chars) {
    ch = ch.toLowerCase();
    const prevCount = charCounts.get(ch) || 0;
    charCounts.set(ch, prevCount+1);
  }
  return charCounts;
}

const result = countChars('AaBccc');
assert.deepEqual(
  [...result],
  [
    ['a', 2],
    ['b', 1],
    ['c', 3],
  ]
);
```

## 30.3 A few more details about the keys of Maps (advanced)

Any value can be a key, even an object:

```
const map = new Map();
```

```
const KEY1 = {};  
const KEY2 = {};  
  
map.set(KEY1, 'hello');  
map.set(KEY2, 'world');  
  
assert.equal(map.get(KEY1), 'hello');  
assert.equal(map.get(KEY2), 'world');
```

### 30.3.1 What keys are considered equal?

Most Map operations need to check whether a value is equal to one of the keys. They do so via the internal operation `SameValueZero`<sup>1</sup>, which works like `===`, but considers `NaN` to be equal to itself.

As a consequence, you can use `NaN` as a key in Maps, just like any other value:

```
> const map = new Map();  
  
> map.set(NaN, 123);  
> map.get(NaN)  
123
```

Different objects are always considered to be different. That is something that can't be configured (yet – TC39 is aware that this is important functionality).

```
> new Map().set({}, 1).set({}, 2).size  
2
```

## 30.4 Missing Map operations

### 30.4.1 Mapping and filtering Maps

You can `.map()` and `.filter()` Arrays, but there are no such operations for Maps. The solution is:

1. Convert the Map into an Array of [key,value] pairs.
2. Map or filter the Array.
3. Convert the result back to a Map.

I'll use the following Map to demonstrate how that works.

```
const originalMap = new Map()  
.set(1, 'a')  
.set(2, 'b')  
.set(3, 'c');
```

Mapping originalMap:

```
const mappedMap = new Map( // step 3  
  [...originalMap] // step 1
```

---

<sup>1</sup><http://www.ecma-international.org/ecma-262/6.0/#sec-samevaluezero>

```

    .map(([k, v]) => [k * 2, '_' + v]) // step 2
  );
  assert.deepEqual([...mappedMap],
    [[2, '_a'], [4, '_b'], [6, '_c']]);

```

Filtering originalMap:

```

const filteredMap = new Map( // step 3
  [...originalMap] // step 1
  .filter(([k, v]) => k < 3) // step 2
);
assert.deepEqual([...filteredMap],
  [[1, 'a'], [2, 'b']]);

```

Step 1 is performed by the spread operator (...).

### 30.4.2 Combining Maps

There are no methods for combining Maps, which is why the approach from the previous section must be used to do so.

Let's combine the following two Maps:

```

const map1 = new Map()
  .set(1, '1a')
  .set(2, '1b')
  .set(3, '1c')
;

const map2 = new Map()
  .set(2, '2b')
  .set(3, '2c')
  .set(4, '2d')
;

```

To combine map1 and map2, we turn them into Arrays via the spread operator (...) and concatenate those Arrays. Afterwards, we convert the result back to a Map. All of that is done in line A.

```

const combinedMap = new Map([...map1, ...map2]); // (A)
assert.deepEqual(
  [...combinedMap], // convert to Array for comparison
  [
    [ 1, '1a' ],
    [ 2, '2b' ],
    [ 3, '2c' ],
    [ 4, '2d' ]
  ]
);

```



#### Exercise: Combining two Maps

exercises/maps-sets/combine\_maps\_test.js



## 30.5 Quick reference: Map<K, V>

Note: For the sake of conciseness, I'm pretending that all keys have the same type K and that all values have the same type V.

### 30.5.1 Constructor

- `new Map<K, V>(entries?: Iterable<[K, V]>)` <sup>[ES6]</sup>

If you don't provide the parameter `entries` then an empty Map is created. If you do provide an iterable over [key, value] pairs then those pairs are used to add entries to the Map. For example:

```
const map = new Map([
  [ 1, 'one' ],
  [ 2, 'two' ],
  [ 3, 'three' ], // trailing comma is ignored
]);
```

### 30.5.2 Handling single entries

Map.prototype:

- `.get(key: K): V` <sup>[ES6]</sup>

Returns the value that key is mapped to in this Map. If there is no key key in this Map, undefined is returned.

```
const map = new Map([[1, 'one'], [2, 'two']]);
assert.equal(map.get(1), 'one');
assert.equal(map.get(5), undefined);
```

- `.set(key: K, value: V): this` <sup>[ES6]</sup>

Maps the given key to the given value. If there is already an entry whose key is key, it is updated. Otherwise, a new entry is created. This method returns this, which means that you can chain it.

```
const map = new Map([[1, 'one'], [2, 'two']]);
map.set(1, 'ONE!');
map.set(3, 'THREE!');
assert.deepEqual(
  [...map.entries()],
  [[1, 'ONE!'], [2, 'two'], [3, 'THREE!']]);
```

- `.has(key: K): boolean` <sup>[ES6]</sup>

Returns whether the given key exists in this Map.

```
const map = new Map([[1, 'one'], [2, 'two']]);
assert.equal(map.has(1), true); // key exists
assert.equal(map.has(5), false); // key does not exist
```

- `.delete(key: K): boolean` <sup>[ES6]</sup>

If there is an entry whose key is `key`, it is removed and `true` is returned. Otherwise, nothing happens and `false` is returned.

```
const map = new Map([[1, 'one'], [2, 'two']]);
assert.equal(map.delete(1), true);
assert.equal(map.delete(5), false); // nothing happens
assert.deepEqual(
  [...map.entries()],
  [[2, 'two']]);
```

### 30.5.3 Handling all entries

`Map.prototype`:

- `get .size: number` <sup>[ES6]</sup>

Returns how many entries there are in this `Map`.

```
const map = new Map([[1, 'one'], [2, 'two']]);
assert.equal(map.size, 2);
```

In JavaScript indexable sequences (such as `Arrays`) have a `.length`, while mainly unordered collections (such as `Maps`) have a `.size`.

- `.clear(): void` <sup>[ES6]</sup>

Removes all entries from this `Map`.

```
const map = new Map([[1, 'one'], [2, 'two']]);
assert.equal(map.size, 2);
map.clear();
assert.equal(map.size, 0);
```

### 30.5.4 Iterating and looping

Both iterating and looping happen in the order in which entries were added to a `Map`.

`Map.prototype`:

- `.entries(): Iterable<[K,V]>` <sup>[ES6]</sup>

Returns an iterable with one `[key,value]` pair for each entry in this `Map`. The pairs are `Arrays` of length 2.

```
const map = new Map([[1, 'one'], [2, 'two']]);
for (const entry of map.entries()) {
  console.log(entry);
}
// Output:
// [1, 'one']
// [2, 'two']
```

- `.forEach(callback: (value: V, key: K, theMap: Map<K,V>) => void, thisArg?: any): void` <sup>[ES6]</sup>

The first parameter is a callback that is invoked once for each entry in this Map. If `thisArg` is provided, this is set to it for each invocation. Otherwise, this is set to undefined.

```
const map = new Map([[1, 'one'], [2, 'two']]);
map.forEach((value, key) => console.log(value, key));
// Output:
// 'one', 1
// 'two', 2
```

- `.keys(): Iterable<K>` <sup>[ES6]</sup>

Returns an iterable over all keys in this Map.

```
const map = new Map([[1, 'one'], [2, 'two']]);
for (const key of map.keys()) {
  console.log(key);
}
// Output:
// 1
// 2
```

- `.values(): Iterable<V>` <sup>[ES6]</sup>

Returns an iterable over all values in this Map.

```
const map = new Map([[1, 'one'], [2, 'two']]);
for (const value of map.values()) {
  console.log(value);
}
// Output:
// 'one'
// 'two'
```

- `[Symbol.iterator](): Iterable<[K,V]>` <sup>[ES6]</sup>

The default way of iterating over Maps. Same as `.entries()`.

```
const map = new Map([[1, 'one'], [2, 'two']]);
for (const [key, value] of map) {
  console.log(key, value);
}
// Output:
// 1, 'one'
// 2, 'two'
```

### 30.5.5 Sources

- TypeScript's built-in typings<sup>2</sup>

---

<sup>2</sup><https://github.com/Microsoft/TypeScript/blob/master/lib/>

## 30.6 FAQ

### 30.6.1 When should I use a Map, when an object?

If you map anything other than strings to any kind of data, you have no choice: you must use a Map.

If, however, you are mapping strings to arbitrary data, you must decide whether or not to use an object. A rough general guideline is:

- Is there a fixed set of keys (known at development time)?  
Then use an object and access the values via fixed keys: `obj . key`
- Can the set of keys change at runtime?  
Then use a Map and access the values via keys stored in variables: `map . get ( theKey )`

### 30.6.2 When would I use an object as a key in a Map?

Map keys mainly make sense if they are compared by value (the same “content” means that two values are considered equal, not the same identity). That excludes objects. There is one use case – externally attaching data to objects, but that use case is better served by WeakMaps where entries don’t prevent garbage collection (for details, consult [the next chapter](#)).

## Chapter 31

# WeakMaps (WeakMap)

### Contents

---

<b>31.1 Attaching values to objects via WeakMaps</b> . . . . .	<b>317</b>
31.1.1 The keys of a WeakMap are weakly held . . . . .	318
<b>31.2 WeakMaps as black boxes</b> . . . . .	<b>318</b>
<b>31.3 Examples</b> . . . . .	<b>318</b>
31.3.1 Caching computed results via WeakMaps . . . . .	318
31.3.2 Keeping private data via WeakMaps . . . . .	319
<b>31.4 WeakMap API</b> . . . . .	<b>320</b>

---

WeakMaps are similar to Maps, with the following differences:

- They can be used to attach data to objects, without preventing those objects from being garbage-collected.
- They are black boxes where a value can only be accessed if you have both the WeakMap and the key.

The next two sections examine in more detail what that means.

### 31.1 Attaching values to objects via WeakMaps

This is how you attach a value to an object via a WeakMap:

```
const wm = new WeakMap();
{
  const obj = {};
  wm.set(obj, 'attachedValue'); // (A)
}
// (B)
```

In line A, we attach a value to obj. In line B, obj can be garbage-collected. The notable feature of WeakMaps is that wm does not prevent obj from being garbage-collected. This technique of attaching

a value to an object is equivalent to storing a property of that object externally. If `wm` were a property, the previous code would look as follows.

```
{
  const obj = {};
  obj.wm = 'attachedValue';
}
```

### 31.1.1 The keys of a WeakMap are weakly held

The keys of a WeakMap are said to be *weakly held*: Normally, references to an object prevent the object from being garbage-collected. However, WeakMap keys don't. Additionally, WeakMap entries, whose keys were garbage-collected, are also (eventually) garbage-collected.

Weakly held keys only make sense for objects. Thus, you can only use objects as keys:

```
> const wm = new WeakMap();
> wm.set(123, 'test')
TypeError: Invalid value used as weak map key
```

## 31.2 WeakMaps as black boxes

It is impossible to inspect what's inside a WeakMap:

- For example, you can't iterate or loop over keys, values or entries. And you can't compute the size.
- Additionally, you can't clear a WeakMap, either – you have to create a fresh instance.

These restrictions enable a security property. Quoting Mark Miller<sup>1</sup>: “The mapping from weakmap/key pair value can only be observed or affected by someone who has both the weakmap and the key. With `clear()`, someone with only the WeakMap would've been able to affect the WeakMap-and-key-to-value mapping.”

## 31.3 Examples

### 31.3.1 Caching computed results via WeakMaps

With WeakMaps, you can associate previously computed results with objects, without having to worry about memory management. The following function `countOwnKeys()` is an example: it caches previous results in the WeakMap cache.

```
const cache = new WeakMap();
function countOwnKeys(obj) {
  if (cache.has(obj)) {
    return [cache.get(obj), 'cached'];
  } else {
```

---

<sup>1</sup><https://github.com/rwaldron/tc39-notes/blob/master/es6/2014-11/nov-19.md#412-should-weakmapweakset-have-a-clear-method-markm>

```

    const count = Object.keys(obj).length;
    cache.set(obj, count);
    return [count, 'computed'];
  }
}

```

If we use this function with an object `obj`, you can see that the result is only computed for the first invocation, while a cached value is used for the second invocation:

```

> const obj = { foo: 1, bar: 2};

> countOwnKeys(obj)
[2, 'computed']
> countOwnKeys(obj)
[2, 'cached']

```

### 31.3.2 Keeping private data via WeakMaps

In the following code, the WeakMaps `_counter` and `_action` are used to store the data of virtual properties of instances of `Countdown`:

```

let _counter = new WeakMap();
let _action = new WeakMap();

class Countdown {
  constructor(counter, action) {
    _counter.set(this, counter);
    _action.set(this, action);
  }
  dec() {
    let counter = _counter.get(this);
    if (counter < 1) return;
    counter--;
    _counter.set(this, counter);
    if (counter === 0) {
      _action.get(this)();
    }
  }
}

```

```

// The two pseudo-properties are truly private:
assert.deepEqual(
  Reflect.ownKeys(new Countdown()),
  []);

```



#### Exercise: WeakMaps for private data

exercises/maps-sets/weakmaps\_private\_data\_test.js

## 31.4 WeakMap API

The constructor and the four methods of `WeakMap` work the same as **their Map equivalents**:

- `new WeakMap<K, V>(entries?: Iterable<[K, V]>) [ES6]`
- `.delete(key: K) : boolean [ES6]`
- `.get(key: K) : V [ES6]`
- `.has(key: K) : boolean [ES6]`
- `.set(key: K, value: V) : this [ES6]`



# Chapter 32

## Sets (Set)

### Contents

---

<b>32.1 Using Sets</b>	<b>322</b>
32.1.1 Creating Sets	322
32.1.2 Adding, removing, checking inclusion	322
32.1.3 Determining the size of a Set and clearing it	322
32.1.4 Iterating over a set	323
<b>32.2 Comparing Set elements</b>	<b>323</b>
<b>32.3 Missing Set operations</b>	<b>324</b>
32.3.1 Mapping	324
32.3.2 Filtering	324
32.3.3 Union	324
32.3.4 Intersection	325
32.3.5 Difference	325
<b>32.4 Quick reference: Set&lt;T&gt;</b>	<b>325</b>
32.4.1 Constructor	325
32.4.2 Single Set elements	325
32.4.3 All Set elements	326
32.4.4 Iterating and looping	326
32.4.5 Symmetry with Map	327

---

Before ES6, JavaScript didn't have a data structure for sets. Instead, two work-around were used:

- The keys of objects as sets of strings.
- Arrays as sets of arbitrary values (e.g., checking if an element is in a set via `.includes()`).

ECMAScript 6 has the data structure `Set`, which works for arbitrary values.

## 32.1 Using Sets

### 32.1.1 Creating Sets

There are three common ways of creating Sets.

First, you can use the constructor without any parameters to create an empty Set:

```
const emptySet = new Set();
assert.equal(emptySet.size, 0);
```

Second, you can pass an iterable (e.g. an Array) to the constructor. The iterated values become elements of the new Set:

```
const set = new Set(['red', 'green', 'blue']);
```

Third, the `.add()` method adds elements to a Set and is chainable:

```
const set = new Set()
  .add('red')
  .add('green')
  .add('blue');
```

### 32.1.2 Adding, removing, checking inclusion

`.add()` adds an element to a Set. `.has()` checks if an element is included in a Set. `.delete()` removes an element from a Set.

```
const set = new Set();

set.add('red');
assert.equal(set.has('red'), true);

assert.equal(set.delete('red'), true); // there was a deletion
assert.equal(set.has('red'), false);
```

### 32.1.3 Determining the size of a Set and clearing it

`.size` contains the number of elements in a Set. `.clear()` removes all elements of a Set.

```
const set = new Set()
  .add('foo')
  .add('bar')
;

assert.equal(set.size, 2)
set.clear();
assert.equal(set.size, 0)
```

### 32.1.4 Iterating over a set

Sets are iterable and the for-of loop works as you'd expect:

```
const set = new Set(['red', 'green', 'blue']);
for (const x of set) {
  console.log(x);
}
// Output:
// 'red'
// 'green'
// 'blue'
```

As you can see, Sets preserve iteration order. That is, elements are always iterated over in the order in which they were added.

Spreading (...) into Arrays works with iterables and thus lets you convert a Set to an Array:

```
const set = new Set(['red', 'green', 'blue']);
const arr = [...set]; // ['red', 'green', 'blue']
```

#### 32.1.4.1 Removing duplicates from Arrays and strings

Converting an Array to a Set and back, removes duplicates from the Array:

```
assert.deepEqual(
  [...new Set([1, 2, 1, 2, 3, 3, 3])],
  [1, 2, 3]);
```

Strings are also iterable and therefore can be used as parameters for new Set():

```
assert.equal(
  [...new Set('ababccc')].join(''),
  'abc');
```

## 32.2 Comparing Set elements

As with Maps, elements are compared similarly to ===, with the exception of NaN being equal to itself.

```
> const set = new Set([NaN]);
> set.size
1
> set.has(NaN)
true
```

Adding an element a second time has no effect:

```
> const set = new Set();

> set.add('foo');
> set.size
1
```

```
> set.add('foo');
> set.size
1
```

Similarly to `===`, two different objects are never considered equal (which can't currently be customized):

```
> const set = new Set();
```

```
> set.add({});
> set.size
1
```

```
> set.add({});
> set.size
2
```

## 32.3 Missing Set operations

Sets are missing several common operations. They can usually be implemented by:

- Converting Sets to Arrays (via `spreading (...)`).
- Performing the operation.
- Converting the result back to a Set.

### 32.3.1 Mapping

```
const set = new Set([1, 2, 3]);
const mappedSet = new Set([...set].map(x => x * 2));

// Convert mappedSet to an Array to check what's inside it
assert.deepEqual([...mappedSet], [2, 4, 6]);
```

### 32.3.2 Filtering

```
const set = new Set([1, 2, 3, 4, 5]);
const filteredSet = new Set([...set].filter(x => (x % 2) === 0));

assert.deepEqual([...filteredSet], [2, 4]);
```

### 32.3.3 Union

Union ( $a \cup b$ ): create a Set that contains the elements of both Set a and Set b.

```
const a = new Set([1,2,3]);
const b = new Set([4,3,2]);
// Use spreading to concatenate two iterables
```

```
const union = new Set([...a, ...b]);

assert.deepEqual([...union], [1, 2, 3, 4]);
```

### 32.3.4 Intersection

Intersection ( $a \cap b$ ): create a Set that contains those elements of Set *a* that are also in Set *b*.

```
const a = new Set([1,2,3]);
const b = new Set([4,3,2]);
const intersection = new Set(
  [...a].filter(x => b.has(x)));

assert.deepEqual([...intersection], [2, 3]);
```

### 32.3.5 Difference

Difference ( $a \setminus b$ ): create a Set that contains those elements of Set *a* that are not in Set *b*. This operation is also sometimes called *minus* (-).

```
const a = new Set([1,2,3]);
const b = new Set([4,3,2]);
const difference = new Set(
  [...a].filter(x => !b.has(x)));

assert.deepEqual([...difference], [1]);
```

## 32.4 Quick reference: Set<T>

### 32.4.1 Constructor

- `new Set<T>(values?: Iterable<T>)` <sup>[ES6]</sup>

If you don't provide the parameter *values* then an empty Set is created. If you do then the iterated values are added as elements to the Set. For example:

```
const set = new Set(['red', 'green', 'blue']);
```

### 32.4.2 Single Set elements

Set.prototype:

- `.add(value: T): this` <sup>[ES6]</sup>

Adds value to this Set. This method returns *this*, which means that it can be chained.

```
const set = new Set(['red', 'green']);
set.add('blue');
assert.deepEqual([...set], ['red', 'green', 'blue']);
```

- `.delete(value: T): boolean` <sup>[ES6]</sup>

Removes value from this Set.

```
const set = new Set(['red', 'green', 'blue']);
assert.equal(set.delete('red'), true); // there was a deletion
assert.deepEqual([...set], ['green', 'blue']);
```

- `.has(value: T): boolean` <sup>[ES6]</sup>

Checks whether value is in this Set.

```
const set = new Set(['red', 'green']);
assert.equal(set.has('red'), true);
assert.equal(set.has('blue'), false);
```

### 32.4.3 All Set elements

Set.prototype:

- `get .size: number` <sup>[ES6]</sup>

Returns how many elements there are in this Set.

```
const set = new Set(['red', 'green', 'blue']);
assert.equal(set.size, 3);
```

- `.clear(): void` <sup>[ES6]</sup>

Removes all elements from this Set.

```
const set = new Set(['red', 'green', 'blue']);
assert.equal(set.size, 3);
set.clear();
assert.equal(set.size, 0);
```

### 32.4.4 Iterating and looping

Set.prototype:

- `.values(): Iterable<T>` <sup>[ES6]</sup>

Returns an iterable over all elements of this Set.

```
const set = new Set(['red', 'green']);
for (const x of set.values()) {
  console.log(x);
}
// Output:
// 'red'
// 'green'
```

- `[Symbol.iterator](): Iterable<T>` <sup>[ES6]</sup>

The default way of iterating over Sets. Same as `.values()`.

```
const set = new Set(['red', 'green']);
for (const x of set) {
  console.log(x);
}
// Output:
// 'red'
// 'green'
```

- `.forEach(callback: (value: T, value2: T, theSet: Set<T>) => void, thisArg?: any): void` <sup>[ES6]</sup>

Loops over the elements of this Set and invokes the callback (first parameter) for each one. `value` and `key` are both set to the element, so that this method works similarly to `Map.prototype.forEach`. If `thisArg` is provided, this is set to it for each call. Otherwise, this is set to `undefined`.

```
const set = new Set(['red', 'green']);
set.forEach(x => console.log(x));
// Output:
// 'red'
// 'green'
```

### 32.4.5 Symmetry with Map

The following two methods only exist so that the interface of Sets is similar to the interface of Maps. Each Set element is handled as if it were a Map entry whose key and value are the element.

- `Set.prototype.entries(): Iterable<[T,T]>` <sup>[ES6]</sup>
- `Set.prototype.keys(): Iterable<T>` <sup>[ES6]</sup>

`.entries()` enables you to convert a Set to a Map:

```
const set = new Set(['a', 'b', 'c']);
const map = new Map(set.entries());
assert.deepEqual(
  [...map.entries()],
  [['a', 'a'], ['b', 'b'], ['c', 'c']]);
```





## Chapter 33

# WeakSets (WeakSet)

### Contents

33.1 Example: Marking objects as safe to use with a method . . . . .	329
33.2 WeakSet API . . . . .	330

WeakSets are similar to Sets, with the following differences:

- They can hold objects, without preventing those objects from being garbage-collected.
- They are black boxes: You only get any data out of a WeakSet if you have both the WeakSet and a value. The only methods that are supported are `.add()`, `.delete()`, `.has()`. Consult the section on [WeakMaps as black boxes](#) for an explanation of why WeakSets don't allow iteration, looping and clearing.

Given that you can't iterate over their elements, there are not that many use cases for WeakSets. They do enable you to mark objects.

### 33.1 Example: Marking objects as safe to use with a method

Domenic Denicola shows<sup>1</sup> how a class `Foo` can ensure that its methods are only applied to instances that were created by it:

```
const foos = new WeakSet();

class Foo {
  constructor() {
    foos.add(this);
  }

  method() {
    if (!foos.has(this)) {
      throw new TypeError('Incompatible object!');
    }
  }
}
```

<sup>1</sup><https://mail.mozilla.org/pipermail/es-discuss/2015-June/043027.html>

```
    }  
  }  
}  
  
const foo = new Foo();  
foo.method(); // works  
  
assert.throws(  
  () => {  
    const obj = {};  
    Foo.prototype.method.call(obj); // throws an exception  
  },  
  TypeError  
);
```

## 33.2 WeakSet API

The constructor and the three methods of `WeakSet` work the same as **their `Set` equivalents**:

- `new WeakSet<T>(values?: Iterable<T>)` <sup>[ES6]</sup>
- `.add(value: T): this` <sup>[ES6]</sup>
- `.delete(value: T): boolean` <sup>[ES6]</sup>
- `.has(value: T): boolean` <sup>[ES6]</sup>

## Chapter 34

# Destructuring

### Contents

---

<b>34.1 A first taste of destructuring</b>	<b>331</b>
<b>34.2 Constructing vs. extracting</b>	<b>332</b>
<b>34.3 Where can we destructure?</b>	<b>333</b>
<b>34.4 Object-destructuring</b>	<b>333</b>
34.4.1 Property value shorthands	334
34.4.2 Rest properties	334
34.4.3 Syntax pitfall: assigning	334
<b>34.5 Array-destructuring</b>	<b>335</b>
34.5.1 Rest elements	336
34.5.2 Array-destructuring works with any iterable	336
34.5.3 You can only Array-destructure iterable values	336
<b>34.6 Destructuring use case: multiple return values</b>	<b>336</b>
<b>34.7 Not finding a match</b>	<b>337</b>
34.7.1 Object-destructuring and missing properties	338
34.7.2 Array-destructuring and missing elements	338
<b>34.8 What values can't be destructured?</b>	<b>338</b>
34.8.1 You can't object-destructure undefined and null	338
<b>34.9 (Advanced)</b>	<b>338</b>
<b>34.10 Default values</b>	<b>339</b>
34.10.1 Default values in Array-destructuring	339
34.10.2 Default values in object-destructuring	339
<b>34.11 Parameter definitions are similar to destructuring</b>	<b>339</b>
<b>34.12 Nesting</b>	<b>340</b>
<b>34.13 Further reading</b>	<b>340</b>

---

### 34.1 A first taste of destructuring

With normal assignment, you extract one piece of data at a time. For example, via:

```
x = arr[1];
```

With destructuring, you can extract multiple pieces of data at the same time, via patterns in locations that receive data. The left-hand side of `=` in the previous code is one such location. In the following code, the square brackets in line A are a destructuring pattern. It extracts the values of the Array elements at index 0 and index 1:

```
const arr = ['a', 'b', 'c'];
const [x, y] = arr; // (A)
assert.equal(x, 'a');
assert.equal(y, 'b');
```

Note that the pattern is “smaller” than the data: we are only extracting what we need.

## 34.2 Constructing vs. extracting

In order to understand what destructuring is, consider that JavaScript has two kinds of operations that are opposites:

- On one hand, you can construct data via literals and similar operations. For example:

```
const jane = {};
jane.first = 'Jane';
jane.last = 'Doe';
```

- On the other hand, you can extract data, by accessing properties or Array elements:

```
const f = jane.first;
const l = jane.last;

assert.equal(f, 'Jane');
assert.equal(l, 'Doe');
```

So far, we have constructed and extract data, one value at a time. Object and Array literals additionally let you construct data in “batch mode” – involving multiple values at the same time:

```
const jane = {
  first: 'Jane',
  last: 'Doe',
};
```

Destructuring offers the same kind of batch mode for extracting data:

```
const {first: f, last: l} = jane;

assert.equal(f, 'Jane');
assert.equal(l, 'Doe');
```

We are setting the new variable `f` to `jane.first` and the new variable `l` to `jane.last`.

### 34.3 Were can we destructure?

Destructuring patterns can be used at “assignment locations” such as:

- Variable declarations:

```
const [a] = ['x'];
assert.equal(a, 'x');

let [b] = ['y'];
assert.equal(b, 'y');
```

- Assignments:

```
let b;
[b] = ['z'];
assert.equal(b, 'z');
```

- Parameter definitions:

```
const f = ([x]) => x;
assert.equal(f(['a']), 'a');
```

Note that variable declarations include `const` and `let` declarations in `for-of` loops:

```
const arr = ['a', 'b'];
for (const [index, element] of arr.entries()) {
  console.log(index, element);
}
// Output:
// 0, 'a'
// 1, 'b'
```

Next, we’ll look deeper into the two kinds of destructuring: object-destructuring and Array-destructuring.

### 34.4 Object-destructuring

*Object-destructuring* lets you batch-extract values of properties, via patterns that look like object literals:

```
const address = {
  street: 'Evergreen Terrace',
  number: '742',
  city: 'Springfield',
  state: 'NT',
  zip: '49007',
};

const { street: s, city: c } = address;
assert.equal(s, 'Evergreen Terrace');
assert.equal(c, 'Springfield');
```

You can think of the pattern as a transparent sheet that you place over the data: The pattern key `'street'` has a match in the data. Therefore, the data value `'Evergreen Terrace'` is assigned to the pattern variable `s`.

You can also object-destructure primitive values:

```
const {length: len} = 'abc';
assert.equal(len, 3);
```

And you can object-destructure Arrays (remember that Array indices are also properties):

```
const {0:x, 2:y} = ['a', 'b', 'c'];
assert.equal(x, 'a');
assert.equal(y, 'c');
```

### 34.4.1 Property value shorthands

Object literals support property value shorthands and so do object patterns:

```
const { street, city } = address;
assert.equal(street, 'Evergreen Terrace');
assert.equal(city, 'Springfield');
```

### 34.4.2 Rest properties

In object literals, you can have spread properties. In object patterns, you can have rest properties (which must come last):

```
const obj = { a: 1, b: 2, c: 3 };
const { a: propValue, ...remaining } = obj; // (A)

assert.equal(propValue, 1);
assert.deepEqual(remaining, {b:2, c:3});
```

A rest property variable, such as `remaining` (line A), is assigned an object with all data properties whose keys are not mentioned in the pattern.

### 34.4.3 Syntax pitfall: assigning

If we object-destructure in an assignment, we run into the old problem that you can't start a statement with a curly brace, because then JavaScript thinks you are starting a block:

```
let value;
assert.throws(
  () => eval("{prop: value} = { prop: 'hello' };"),
  {
    name: 'SyntaxError',
    message: 'Unexpected token =',
  });
```

The `eval()` in the previous example is necessary so that the code can run and isn't rejected right away.

The work-around is to put the whole assignment in parentheses:

```
let value;
({prop: value} = { prop: 'hello' });
assert.equal(value, 'hello');
```



#### Exercise: Object-destructuring

`exercises/destructuring/object_destructuring_exrc.js`

## 34.5 Array-destructuring

*Array-destructuring* lets you batch-extract values of Array elements, via patterns that look like Array literals:

```
const [x, y] = ['a', 'b'];
assert.equal(x, 'a');
assert.equal(y, 'b');
```

You can skip elements by mentioning holes inside Array patterns:

```
const [, x, y] = ['a', 'b', 'c']; // (A)
assert.equal(x, 'b');
assert.equal(y, 'c');
```

The first element of the Array pattern in line A is a hole, which is why the Array element at index 0 is ignored.

Array-destructuring is useful when operations return Arrays. As does, for example, the regular expression method `.exec()`:

```
// Skip the element at index 0 (the whole match):
const [, year, month, day] =
  /^(?![0-9]{4})-(?![0-9]{2})-(?![0-9]{2})$/
  .exec('2999-12-31');

assert.equal(year, '2999');
assert.equal(month, '12');
assert.equal(day, '31');
```

You can also use destructuring to swap the values of two variables, without needing a temporary variable:

```
let x = 'a';
let y = 'b';

[x, y] = [y, x]; // swap

assert.equal(x, 'b');
assert.equal(y, 'a');
```

### 34.5.1 Rest elements

In Array literals, you can have spread elements. In Array patterns, you can have rest elements (which must come last):

```
const [x, y, ...remaining] = ['a', 'b', 'c', 'd']; // (A)

assert.equal(x, 'a');
assert.equal(y, 'b');
assert.deepEqual(remaining, ['c', 'd']);
```

A rest element variable such as `remaining` (line A), is assigned an Array with all elements of the destructured value that were not mentioned, yet.

### 34.5.2 Array-destructuring works with any iterable

Array-destructuring can be applied to any value that is iterable, not just to Arrays:

```
// Sets are iterable
const mySet = new Set().add('a').add('b').add('c');
const [first, second] = mySet;
assert.equal(first, 'a');
assert.equal(second, 'b');

// Strings are iterable
const [a, b] = 'xyz';
assert.equal(a, 'x');
assert.equal(b, 'y');
```

### 34.5.3 You can only Array-destructure iterable values

Array-destructuring demands that the destructured value be iterable. Therefore, you can't Array-destructure `undefined` and `null`. But you can't Array-destructure non-iterable objects, either:

```
assert.throws(
  () => { const [x] = {}; },
  {
    name: 'TypeError',
    message: '{} is not iterable',
  }
);
```

## 34.6 Destructuring use case: multiple return values

Destructuring is very useful if a function returns multiple values – either packaged as an Array or packaged as an object.



Consider a function `findElement()` that finds elements in an Array: Its parameter is a function that receives the value and index of an element and returns a boolean indicating if this is the element the caller is looking for. We are now faced with a dilemma: Should `findElement()` return the value of the element it found or the index? One solution would be to create two separate functions, but that would result in duplicated code, because both functions would be very similar.

The following implementation avoids duplication by returning an object that contains both index and value of the element that is found:

```
function findElement(arr, predicate) {
  for (let index=0; index < arr.length; index++) {
    const element = arr[index];
    if (predicate(element)) {
      // We found something:
      return { element, index };
    }
  }
  // We didn't find anything:
  return { element: undefined, index: -1 };
}
```

Destructuring helps us with processing the result of `findElement()`:

```
const arr = [7, 8, 6];

const {element, index} = findElement(arr, x => x % 2 === 0);
assert.equal(element, 8);
assert.equal(index, 1);
```

As we are working with property keys, the order in which we mention `element` and `index` doesn't matter:

```
const {index, element} = findElement(arr, x => x % 2 === 0);
```

The kicker is that destructuring also serves us well if we are only interested in one of the two results:

```
const arr = [7, 8, 6];

const {element} = findElement(arr, x => x % 2 === 0);
assert.equal(element, 8);

const {index} = findElement(arr, x => x % 2 === 0);
assert.equal(index, 1);
```

All of these conveniences combined make this way of handling multiple return values quite versatile.

## 34.7 Not finding a match

What happens if there is no match for part of a pattern? The same thing that happens if you use non-batch operators: you get `undefined`.

### 34.7.1 Object-destructuring and missing properties

If a property in an object pattern has no match on the right-hand side, you get `undefined`:

```
const {prop: p} = {};  
assert.equal(p, undefined);
```

### 34.7.2 Array-destructuring and missing elements

If an element in an Array pattern has no match on the right-hand side, you get `undefined`:

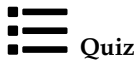
```
const [x] = [];  
assert.equal(x, undefined);
```

## 34.8 What values can't be destructured?

### 34.8.1 You can't object-destructure `undefined` and `null`

Object-destructuring only fails if the value to be destructured is either `undefined` or `null`. That is, it fails whenever accessing a property via the dot operator would fail, too.

```
assert.throws(  
  () => { const {prop} = undefined; },  
  {  
    name: 'TypeError',  
    message: "Cannot destructure property `prop` of 'undefined' or 'null'.",  
  }  
);  
assert.throws(  
  () => { const {prop} = null; },  
  {  
    name: 'TypeError',  
    message: "Cannot destructure property `prop` of 'undefined' or 'null'.",  
  }  
);
```



Quiz

See quiz app.

## 34.9 (Advanced)

All of the remaining sections are advanced.

## 34.10 Default values

Normally, if a pattern has no match, the corresponding variable is set to `undefined`:

```
const {prop: p} = {};  
assert.equal(p, undefined);
```

With default values, you can specify a value other than `undefined`, that should be used in such a case:

```
const {prop: p = 123} = {}; // (A)  
assert.equal(p, 123);
```

In line A, we specify the default value for `p` to be 123. That default is used, because the data that we are destructuring has no property named `prop`.

### 34.10.1 Default values in Array-destructuring

Here, we have two default values that are assigned to the variables `x` and `y`, because the corresponding elements don't exist in the Array that is destructured.

```
const [x=1, y=2] = [];  
  
assert.equal(x, 1);  
assert.equal(y, 2);
```

The default value for the first element of the Array pattern is 1, the default value for the second element is 2.

### 34.10.2 Default values in object-destructuring

You can also specify default values for object-destructuring:

```
const {first: f='', last: l='' } = {};  
assert.equal(f, '');  
assert.equal(l, '');
```

Neither property key `first` nor property key `last` exist in the object that is destructured. Therefore, the default values are used.

With property value shorthands, this code becomes simpler:

```
const {first='', last='' } = {};  
assert.equal(first, '');  
assert.equal(last, '');
```

## 34.11 Parameter definitions are similar to destructuring

Considering what we have learned in this chapter, parameter definitions have much in common with an Array pattern (rest elements, default values, etc.). In fact, the following two function declarations are equivalent:

```
function f1(pattern1, pattern2 /* etc. */) {  
  // ...  
}  
  
function f2(...args) {  
  const [pattern1, pattern2 /* etc. */] = args;  
  // ...  
}
```

## 34.12 Nesting

Until now, we have only used variables as assignment targets inside destructuring patterns. But you can also use patterns as assignment targets, which enables you to nest patterns to arbitrary depths:

```
const arr = [  
  { first: 'Jane', last: 'Bond' },  
  { first: 'Lars', last: 'Croft' },  
];  
const [, {first}] = arr;  
assert.equal(first, 'Lars');
```

Inside the Array pattern in line A, there is a nested object pattern, at index 1.

## 34.13 Further reading

- “Exploring ES6<sup>1</sup>” goes into more depth for destructuring.



Quiz

See quiz app.

---

<sup>1</sup>[http://exploringjs.com/es6/ch\\_destructuring.html](http://exploringjs.com/es6/ch_destructuring.html)

## Chapter 35

# Synchronous generators (advanced)

### Contents

---

<b>35.1 What are synchronous generators?</b>	<b>341</b>
35.1.1 Generator functions return iterables and fill them via <code>yield</code>	342
35.1.2 <code>yield</code> pauses a generator function	342
35.1.3 Why does <code>yield</code> pause execution?	343
35.1.4 Example: Mapping over iterables	344
<b>35.2 Calling generators from generators (advanced)</b>	<b>345</b>
35.2.1 Calling generators via <code>yield*</code>	345
35.2.2 Example: Iterating over a tree	346
<b>35.3 Example: Reusing loops</b>	<b>347</b>
35.3.1 The loop to reuse	347
35.3.2 Internal iteration (push)	347
35.3.3 External iteration (pull)	348
<b>35.4 Advanced features of generators</b>	<b>348</b>

---

## 35.1 What are synchronous generators?

Synchronous generators are special versions of function definitions and method definitions that always return synchronous iterables:

```
// Generator function declaration
function* genFunc1() { /*...*/ }

// Generator function expression
const genFunc2 = function* () { /*...*/ };

// Generator method definition in an object literal
const obj = {
  * generatorMethod() {
```

```

    // ...
  }
};

// Generator method definition in a class definition
// (class declaration or class expression)
class MyClass {
  * generatorMethod() {
    // ...
  }
}

```

The asterisks mark functions and methods as generators:

- The pseudo-keyword `function*` is a combination of an asterisk and the keyword `function`.
- The pseudo-keyword `*` in front of method definitions, is a combination of an imaginary empty keyword (think `method`, but with zero characters) and an asterisk.

### 35.1.1 Generator functions return iterables and fill them via `yield`

If you call a generator function, it returns an iterable. The generator fills that iterable via the `yield` operator:

```

function* genFunc1() {
  yield 'a';
  yield 'b';
}

// Convert the iterable to an Array, to check what's inside it:
const iterable = genFunc1();
assert.deepEqual([...iterable], ['a', 'b']);

// You can also use a for-of loop
for (const x of genFunc1()) {
  console.log(x);
}

// Output:
// 'a'
// 'b'

```

### 35.1.2 `yield` pauses a generator function

So far, `yield` looks like a simple way of adding values to an iterable. However, it does much more than that – it also pauses and exits the generator function:

- Like `return`, a `yield` exits the body of the function.
- Unlike `return`, if you invoke the function again, execution resumes directly after the `yield`.

Let's examine what that means via the following generator function.

```

let location = 0;
function* genFunc2() {
  location = 1;
  yield 'a';
  location = 2;
  yield 'b';
  location = 3;
}

```

The result of a generator function is called a *generator object*. It is more than just an iterable, but that is beyond the scope of this book (consult “Exploring ES6”<sup>1</sup> if you are interested in further details).

In order to use `genFunc2()`, we must first create the generator object `genObj`. `genFunc2()` is now paused “before” its body.

```

const genObj = genFunc2();
// genFunc2() is now paused “before” its body:
assert.equal(location, 0);

```

`genObj` implements the *iteration protocol*. Therefore, we control the execution of `genFunc2()` via `genObj.next()`. Calling that method, resumes the paused `genFunc2()` and executes it until there is a `yield`. Then execution pauses and `.next()` returns the operand of the `yield`:

```

assert.deepEqual(
  genObj.next(), {value: 'a', done: false});
// genFunc2() is now paused directly after the first `yield`:
assert.equal(location, 1);

```

Note that the yielded value ‘a’ is wrapped in an object, which is how iterables always deliver their values.

We call `genObj.next()` again and execution continues where we previously paused. Once we encounter the second `yield`, `genFunc2()` is paused and `.next()` returns the yielded value ‘b’.

```

assert.deepEqual(
  genObj.next(), {value: 'b', done: false});
// genFunc2() is now paused directly after the second `yield`:
assert.equal(location, 2);

```

We call `genObj.next()` one more time and execution continues until it leaves the body of `genFunc2()`:

```

assert.deepEqual(
  genObj.next(), {value: undefined, done: true});
// We have reached the end of genFunc2():
assert.equal(location, 3);

```

This time, property `.done` of the result of `.next()` is `true`, which means that the iterable is finished.

### 35.1.3 Why does `yield` pause execution?

What are the benefits of `yield` pausing execution? Why doesn’t it simply work like the `Array` method `.push()` and fill the iterable with values – without pausing?

---

<sup>1</sup>[http://exploringjs.com/es6/ch\\_generators.html](http://exploringjs.com/es6/ch_generators.html)

Due to pausing, generators provide many of the features of *coroutines* (think processes that are multi-tasked cooperatively). For example, when you ask for the next value of an iterable, that value is computed *lazily* (on demand). The following two generator functions demonstrate what that means.

```
/**
 * Returns an iterable over lines
 */
function* genLines() {
  yield 'A line';
  yield 'Another line';
  yield 'Last line';
}

/**
 * Input: iterable over lines
 * Output: iterable over numbered lines
 */
function* numberLines(lineIterable) {
  let lineNumber = 1;
  for (const line of lineIterable) { // input
    yield lineNumber + ': ' + line; // output
    lineNumber++;
  }
}
```

Note that the `yield` inside `numberLines()` appears inside a `for-of` loop. `yield` can be used inside loops, but not inside callbacks (more on that later).

Let's combine both generators to produce the iterable `numberedLines`:

```
const numberedLines = numberLines(genLines());
assert.deepEqual(
  numberedLines.next(), {value: '1: A line', done: false});
assert.deepEqual(
  numberedLines.next(), {value: '2: Another line', done: false});
```

Every time we ask `numberedLines` for another value via `.next()`, `numberLines()` only asks `genLines()` for a single line and numbers it. If `genLines()` were to synchronously read its lines from a large file, we would be able to retrieve the first numbered line as soon as it is read from the file. If `yield` didn't pause, we'd have to wait until `genLines()` is completely finished with reading.

### 35.1.4 Example: Mapping over iterables

The following function `mapIter()` is similar to `Array.from()`, but it returns an iterable, not an `Array` and produces its results on demand.

```
function* mapIter(iterable, func) {
  let index = 0;
  for (const x of iterable) {
    yield func(x, index);
    index++;
  }
}
```



```

    }
  }

  const iterable = mapIter(['a', 'b'], x => x + x);
  assert.deepEqual([...iterable], ['aa', 'bb']);

```



### Exercise: Filtering iterables

exercises/generators/filter\_iter\_gen\_test.js

## 35.2 Calling generators from generators (advanced)

### 35.2.1 Calling generators via `yield*`

`yield` only works directly inside generators – so far we haven’t seen a way of delegating yielding to another function or method.

Let’s first examine what does *not* work: In the following example, we’d like `foo()` to call `bar()`, so that the latter yields two values for the former. Alas, a naive approach fails:

```

function* foo() {
  // Nothing happens if we call `bar()`:
  bar();
}

function* bar() {
  yield 'a';
  yield 'b';
}

assert.deepEqual(
  [...foo()], []);

```

Why doesn’t this work? The function call `bar()` returns an iterable, which we ignore.

What we want is for `foo()` to yield everything that is yielded by `bar()`. That’s what the `yield*` operator does:

```

function* foo() {
  yield* bar();
}

function* bar() {
  yield 'a';
  yield 'b';
}

assert.deepEqual(
  [...foo()], ['a', 'b']);

```

In other words, the previous `foo()` is roughly equivalent to:

```

function* foo() {
  for (const x of bar()) {

```

```

    yield bar();
  }
}

```

Note that `yield*` works with any iterable:

```

function* gen() {
  yield* [1, 2];
}
assert.deepEqual(
  [...gen()], [1, 2]);

```

### 35.2.2 Example: Iterating over a tree

`yield*` lets us make recursive calls in generators, which is useful when iterating over recursive data structures such as trees. Take, for example, the following data structure for binary trees.

```

class BinaryTree {
  constructor(value, left=null, right=null) {
    this.value = value;
    this.left = left;
    this.right = right;
  }

  /** Prefix iteration: parent before children */
  * [Symbol.iterator]() {
    yield this.value;
    if (this.left) {
      // Same as yield* this.left[Symbol.iterator]()
      yield* this.left;
    }
    if (this.right) {
      yield* this.right;
    }
  }
}

```

Method `[Symbol.iterator]()` adds support for the iteration protocol, which means that we can use a for-of loop to iterate over an instance of `BinaryTree`:

```

const tree = new BinaryTree('a',
  new BinaryTree('b',
    new BinaryTree('c'),
    new BinaryTree('d')),
  new BinaryTree('e'));

for (const x of tree) {
  console.log(x);
}
// Output:
// 'a'

```

```
// 'b'
// 'c'
// 'd'
// 'e'
```



#### Exercise: Iterating over a nested Array

exercises/generators/iter\_nested\_arrays\_test.js

## 35.3 Example: Reusing loops

One important use case for generators is extracting and reusing loop functionality.

### 35.3.1 The loop to reuse

As an example, consider the following function that iterates over a tree of files and logs their paths (it uses the Node.js API<sup>2</sup> for doing so):

```
function logFiles(dir) {
  for (const fileName of fs.readdirSync(dir)) {
    const filePath = path.resolve(dir, fileName);
    console.log(filePath);
    const stats = fs.statSync(filePath);
    if (stats.isDirectory()) {
      logFiles(filePath); // recursive call
    }
  }
}
const rootDir = process.argv[2];
logFiles(rootDir);
```

How can we reuse this loop, to do something other than logging paths?

### 35.3.2 Internal iteration (push)

One way of reusing iteration code is via *internal iteration*: Each iterated value is passed to a callback (line A).

```
function iterFiles(dir, callback) {
  for (const fileName of fs.readdirSync(dir)) {
    const filePath = path.resolve(dir, fileName);
    callback(filePath); // (A)
    const stats = fs.statSync(filePath);
    if (stats.isDirectory()) {
      iterFiles(filePath, callback);
    }
  }
}
```

---

<sup>2</sup><https://nodejs.org/en/docs/>

```

    }
  }
}
const rootDir = process.argv[2];
const paths = [];
iterFiles(rootDir, p => paths.push(p));

```

### 35.3.3 External iteration (pull)

Another way of reusing iteration code is via *external iteration*: We can write a generator that yields all iterated values.

```

function* iterFiles(dir) {
  for (const fileName of fs.readdirSync(dir)) {
    const filePath = path.resolve(dir, fileName);
    yield filePath; // (A)
    const stats = fs.statSync(filePath);
    if (stats.isDirectory()) {
      yield* iterFiles(filePath);
    }
  }
}

const rootDir = process.argv[2];
const paths = [...iterFiles(rootDir)];

```



#### Exercise: Turning a normal function into a generator

exercises/generators/fib\_seq\_test.js

## 35.4 Advanced features of generators

- `yield` can also *receive* data via `.next()`. For details, consult “Exploring ES6<sup>3</sup>”.
- The result of `yield*` is whatever is returned by its operand. For details, consult “Exploring ES6<sup>4</sup>”.

<sup>3</sup>[http://exploringjs.com/es6/ch\\_generators.html#sec\\_generators-as-observers](http://exploringjs.com/es6/ch_generators.html#sec_generators-as-observers)

<sup>4</sup>[http://exploringjs.com/es6/ch\\_generators.html#\\_recursion-via-yield](http://exploringjs.com/es6/ch_generators.html#_recursion-via-yield)

## **Part VIII**

# **Asynchronicity**



## Chapter 36

# Asynchronous programming in JavaScript

### Contents

---

<b>36.1 The call stack</b>	<b>351</b>
<b>36.2 The event loop</b>	<b>352</b>
<b>36.3 How to avoid blocking the JavaScript process</b>	<b>353</b>
36.3.1 The user interface of the browser can be blocked	353
36.3.2 How can we avoid blocking the browser?	354
36.3.3 Taking breaks	354
36.3.4 Run-to-completion semantics	355
<b>36.4 Patterns for delivering asynchronous results</b>	<b>355</b>
36.4.1 Delivering asynchronous results via events	355
36.4.2 Delivering asynchronous results via callbacks	357
<b>36.5 Asynchronous code: the downsides</b>	<b>358</b>
<b>36.6 Resources</b>	<b>359</b>

---

This chapter explains the foundations of asynchronous programming in JavaScript.

### 36.1 The call stack

Whenever a function calls another function, we need to remember where to return to, after the latter function is finished. That is typically done via a stack, the *call stack*: The caller pushes onto it the location to return to, and the callee jumps to that location after it is done.

This is an example where several calls happen:

```
1 function h(z) {  
2   const error = new Error();  
3   console.log(error.stack);  
4 }
```

```

5  function g(y) {
6    h(y + 1);
7  }
8  function f(x) {
9    g(x + 1);
10 }
11 f(3);
12 // done

```

Initially, before running this piece of code, the call stack is empty. After the function call `f(3)` in line 11, the stack has one entry:

- Line 12 (location in top-level scope)

After the function call `g(x + 1)` in line 9, the stack has two entries:

- Line 10 (location in `f()`)
- Line 12 (location in top-level scope)

After the function call `h(y + 1)` in line 6, the stack has three entries:

- Line 7 (location in `g()`)
- Line 10 (location in `f()`)
- Line 12 (location in top-level scope)

Creating the exception in line 2 is yet another call. That's why the call stack that is recorded inside the exception error includes a location inside `h()`. Logging error produces the following output (note that stack traces record where calls are made, not return locations):

Error

```

at h (demos/async-js/stack_trace.js:2:17)
at g (demos/async-js/stack_trace.js:6:3)
at f (demos/async-js/stack_trace.js:9:3)
at <top level> (demos/async-js/stack_trace.js:11:1)

```

Afterwards, each of the functions terminates and each time the top entry is removed from the stack. After function `f` is done, we are back in top-level scope and the call stack is empty. When the code fragment ends then that is like an implicit return. If we consider the code fragment to be a task that is executed, then returning with an empty call stack ends the task.

## 36.2 The event loop

By default, JavaScript runs in a single process – in both web browsers and Node.js. The so-called *event loop* sequentially executes *tasks* (pieces of code) inside that process. The event loop is depicted in fig. 36.1.

Two parties access the task queue:

- *Task sources* add tasks to the queue. Some of those sources run concurrently to the JavaScript process. For example, one task source takes care of user interface events: If a user clicks somewhere and that triggers JavaScript code, then that code is added to the task queue.
- The *event loop* runs continuously, inside the JavaScript process. It takes a task out of the queue and executes it. Once the call stack is empty and there is a return, the current task is finished. Control



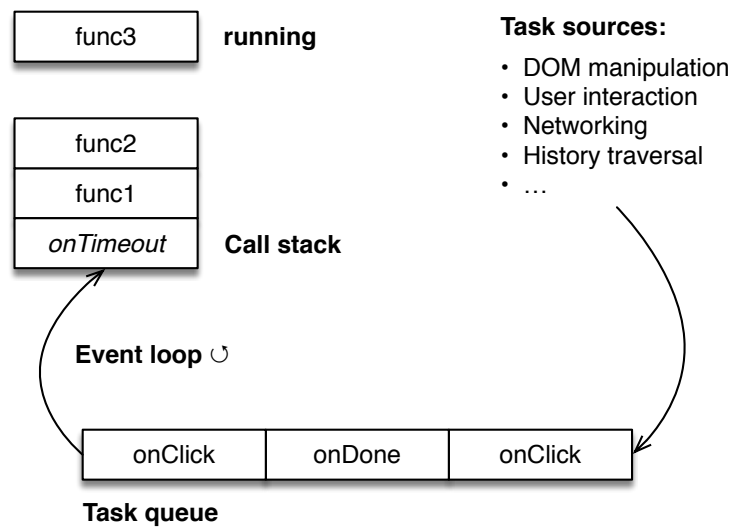


Figure 36.1: Task sources add code to run to the *task queue*, which is emptied by the *event loop*.

goes back to the event loop, which then retrieves the next task from the queue and executes it. Etc.

The following JavaScript code is an approximation of the event loop:

```
while (true) {
  const nextTask = taskQueue.deq();
  execute(nextTask);
}
```

## 36.3 How to avoid blocking the JavaScript process

### 36.3.1 The user interface of the browser can be blocked

Many of the user interface mechanisms of browsers also run in the JavaScript process (as tasks). Therefore, long-running JavaScript code can block the user interface. Let's look at a web page that demonstrates that. There are two ways in which you can try out that page:

- You can run it online<sup>1</sup>.
- You can open the following file inside the repository with the exercises: `demos/async-js/blocking.html`

The following HTML is the page's user interface:

```
<a id="block" href="#">Block</a>
<div id="statusMessage"></div>
<button>Click me!</button>
```

The idea is that you click "Block" and a long-running loop is executed via JavaScript. During that loop, you can't click the button, because the browser/JavaScript process is blocked.

<sup>1</sup><http://rauschma.github.io/async-examples/blocking.html>

A simplified version of the JavaScript code looks like this:

```
document.getElementById('block')
  .addEventListener('click', doBlock); // (A)

function doBlock(event) {
  // ...
  setStatus('Blocking...');
  // ...
  sleep(5000); // (B)
  setStatus('Done');
}

function sleep(milliseconds) {
  const start = Date.now();
  while ((Date.now() - start) < milliseconds);
}

function setStatus(status) {
  document.getElementById('statusMessage')
    .textContent = status;
}
```

These are the key parts of the code:

- Line A: We tell the browser to call `doBlock()` whenever the HTML element is clicked whose ID is `block`.
- `doBlock()` runs a loop for 5000 milliseconds (line B).
- `sleep()` does the actual looping.
- `setStatus()` displays status messages inside the `<div>` whose ID is `statusMessage`.

### 36.3.2 How can we avoid blocking the browser?

There are several ways in which you can prevent a long-running operation from blocking the browser:

- The operation can deliver its result *asynchronously*: Some operations, such as downloads, can be performed concurrently to the JavaScript process. The JavaScript code triggering such an operation registers a callback, which is invoked with the result once the operation is finished. The invocation is handled via the task queue. This style of delivering a result is called *asynchronous*, because the caller doesn't wait until the results are ready. Normal function calls deliver their results synchronously.
- Perform long computations in separate processes: This can be done via so-called *Web Workers*<sup>2</sup> (which are not described in this book).
- Take breaks during long computations. The next section explains how.

### 36.3.3 Taking breaks

The following global function executes its parameter `callback` after a delay of `ms` milliseconds:

<sup>2</sup>[https://developer.mozilla.org/en-US/docs/Web/API/Web\\_Workers\\_API](https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API)

```
setTimeout(callback: () => void, ms: number): void
```

(Note that the type definition is simplified; `setTimeout()` has more features.)

`setTimeout()` is available on both browsers and Node.js. It allows code to take a break and resume execution later, via `callback`. The following code shows how `setTimeout()` and its parameter are executed:

```
console.log('start');
setTimeout(() => {
  console.log('callback');
}, 0);
console.log('end');
```

```
// Output:
// 'start'
// 'end'
// 'callback'
```

`setTimeout()` puts the invocation of its parameter into the task queue. The parameter is therefore executed after the current piece of code (task) is completely finished.

The parameter `ms` only specifies when the invocation is put into the task queue, it does not specify when it is run. It may even never run, if there is a task before it in the queue that never terminates. That explains, why the previous code logs 'end' before 'delayed', even though the delay is zero milliseconds.

### 36.3.4 Run-to-completion semantics

JavaScript makes the following guarantee for tasks:

Each task is always finished (“run to completion”) before the next task is executed.

That means that tasks don’t have to worry about their data being changed while they are working on it (*concurrent modification*). That simplifies programming in JavaScript.

## 36.4 Patterns for delivering asynchronous results

These are three popular patterns for delivering results asynchronously in JavaScript:

- Events
- Callbacks
- Promises

The first two patterns are explained next. Promises are explained in the next chapter.

### 36.4.1 Delivering asynchronous results via events

Events as a pattern work as follows:

- They are used to deliver values asynchronously.
- They may do so multiple times.
- There are three roles in this pattern:

- The *event* object carries the data to be delivered.
- The *event listener* is a function that receives events via a parameter.
- The *event source* sends events and lets you register event listeners.

Multiple variations of this pattern exist in the world of JavaScript. We'll look at three examples next.

#### 36.4.1.1 Events: IndexedDB

IndexedDB is a database that is built into web browsers. This is an example of using it:

```
const openRequest = indexedDB.open('MyDatabase', 1); // (A)

openRequest.onsuccess = (event) => {
  const db = event.target.result;
  // ...
};

openRequest.onerror = (error) => {
  console.log(error);
};
```

indexedDB has an unusual way of invoking operations:

- Each operation has an associated method for creating *request objects*. For example, in line A, the operation is “open”, the method is `.open()` and the request object is `openRequest`.
- The parameters for the operation are provided via the request object, not via parameters of the method. For example, the event listeners (functions) are stored in the properties `.onsuccess` and `.onerror`.
- The invocation of the operation is added to the task queue via the method (in line A). That is, we configure the operation *after* its invocation has already been added to the queue. Only run-to-completion semantics save us from race conditions here and ensure that the operation runs after the current code fragment is finished.

#### 36.4.1.2 Events: XMLHttpRequest

The XMLHttpRequest API lets you make downloads from within a web browser. This is how you download the file `http://example.com/textfile.txt`:

```
const xhr = new XMLHttpRequest(); // (A)
xhr.open('GET', 'http://example.com/textfile.txt'); // (B)
xhr.onload = () => { // (C)
  if (xhr.status == 200) {
    processData(xhr.responseText);
  } else {
    assert.fail(new Error(xhr.statusText));
  }
};
xhr.onerror = () => { // (D)
  assert.fail(new Error('Network error'));
```

```
};
xhr.send(); // (E)

function processData(str) {
  assert.equal(str, 'Content of textfile.txt\n');
}
```

With this API, you first create a request object (line A), then configure it, then send it (line E). The configuration consists of:

- Specifying which HTTP request method to use (line B): GET, POST, PUT, etc.
- Registering a listener (line C) that is notified if something could be downloaded. Inside the listener, you still need to determine if the download contains what you requested or informs you of an error. Note that some of the result data is delivered via `xhr`. I'm not a fan of this kind of mixing of input and output data.
- Registering a listener (line D) that is notified if there was a network error.

### 36.4.1.3 Events: DOM

We have already seen DOM events in action. Let's revisit the previous example, in a slight variation:

```
const element = document.getElementById('my-link'); // (A)
element.addEventListener('click', clickListener); // (B)

function clickListener(event) {
  event.preventDefault(); // (C)
  console.log(event.shiftKey); // (D)
}
```

We first ask the browser to retrieve the HTML element whose ID is `my-link` (line A). Then we add a listener for all `click` events (line B). In the listener, we first tell the browser not to perform its default action – going to the target of the link (line C). Then we log to the console if the shift key is currently pressed (line D).

## 36.4.2 Delivering asynchronous results via callbacks

Callbacks are another pattern for handling asynchronous results. They are only used for one-off results and have the advantage of being less verbose than events.

As an example, consider a function `readFile()` that reads a text file and returns its contents asynchronously. We'll first explore how that is done via callbacks in Node.js and then a callback style that is popular in functional programming.

### 36.4.2.1 Node.js callback pattern

This is how you call `readFile()` if it uses the Node.js callback pattern:

```
readFile('some-file.txt', {encoding: 'utf8'},
  (error, data) => {
    if (error) {
```

```

    console.error(error);
    return;
  }
  assert.equal(data, 'The content of some-file.txt\n');
});

```

There is a single callback that handles both success and failure. If the first parameter is not `null` then an error happened. Otherwise, the result can be found in the second parameter.



#### Exercises: Callback-based code

The following exercises use tests for asynchronous code, which are different from tests for synchronous code. Consult [the section on asynchronous tests in mocha](#) (in the chapter on tests) for more information.

- From synchronous to callback-based code: `exercises/async-js/read_file_cb_exrc.js`
- Implementing a callback-based version of `.map()`: `exercises/async-js/map_cb_test.js`

#### 36.4.2.2 Continuation-passing style

The following style of the callback pattern is popular in functional programming languages:

```

readFile('some-file.txt', {encoding: 'utf8'},
  (data) => { // success
    assert.equal(data, 'The content of some-file.txt\n');
  },
  (error) => { // failure
    assert.fail(error);
  });

```

This pattern is often called *continuation-passing style*, because execution *continues* in either of the two callbacks. The callbacks are called *continuations*.

One benefit of this pattern compared to Node.js-style callbacks is that you can reuse error handling code – simply use the same error callback multiple times. I also prefer not having to check for errors via `if`.

## 36.5 Asynchronous code: the downsides

In many situations, on either browsers or Node.js, you have no choice: You must use asynchronous code. In this chapter, we have seen several patterns that such code can use. All of them have two disadvantages:

- Asynchronous code is more verbose than synchronous code.
- If you call asynchronous code, your code must become asynchronous, too. That's because you can't wait synchronously for an asynchronous result. Asynchronous code has an infectious quality.

The first disadvantage becomes less severe with Promises (covered in the next chapter) and mostly disappears with async functions (covered in the chapter after the next one).

Alas, the infectiousness of async code does not go away. But it is mitigated by the fact that switching between sync and async is easy with async functions.

## 36.6 Resources

- “Help, I’m stuck in an event-loop<sup>3</sup>” by Philip Roberts (video).
- “Event loops<sup>4</sup>”, section in HTML5 spec.

---

<sup>3</sup><https://vimeo.com/96425312>

<sup>4</sup><https://www.w3.org/TR/html5/webappapis.html#event-loops>





## Chapter 37

# Promises for asynchronous programming

### Contents

---

<b>37.1 Overview</b>	<b>362</b>
37.1.1 Asynchronous callback-based code	362
37.1.2 The same functionality via Promises	363
37.1.3 What is a Promise?	364
<b>37.2 The basics of using Promises</b>	<b>364</b>
37.2.1 States of promises	364
37.2.2 <code>Promise.resolve()</code> : create a Promise fulfilled with a given value	365
37.2.3 <code>Promise.reject()</code> : create a Promise rejected with a given value	365
37.2.4 Returning and throwing in <code>.then()</code> callbacks	366
37.2.5 <code>.catch()</code> and its callback	366
37.2.6 Chaining method calls	367
37.2.7 Advantages of promises	367
<b>37.3 Examples</b>	<b>368</b>
37.3.1 Browsers: Promisifying <code>XMLHttpRequest</code>	368
37.3.2 Node.js: <code>util.promisify()</code>	369
37.3.3 Browsers: Fetch API	370
<b>37.4 Promise-based functions start synchronously, settle asynchronously</b>	<b>370</b>
<b>37.5 Error handling: don't mix rejections and exceptions</b>	<b>371</b>
<b>37.6 <code>Promise.all()</code>: working with Arrays of Promises</b>	<b>372</b>
37.6.1 Sequential execution vs. concurrent execution	372
37.6.2 Focus on when computations are started	373
37.6.3 <code>Promise.all()</code> is fork-join	374
37.6.4 Asynchronous <code>.map()</code>	374
<b>37.7 Tips for using Promises</b>	<b>375</b>
37.7.1 Chaining mistake: losing the tail	375
37.7.2 Chaining mistake: nesting	376
37.7.3 Chaining mistake: more nesting than necessary	376

37.7.4 Nesting per se is not evil . . . . .	377
37.7.5 Chaining mistake: creating Promises instead of chaining . . . . .	377
37.8 Further reading . . . . .	378

In this chapter, we explore Promises, yet another pattern for delivering asynchronous results.

This chapter builds on the previous chapter, with background on asynchronous programming in JavaScript.

## 37.1 Overview

Promises are a pattern for delivering asynchronous results. The following code is an example of using the Promise-based function `addAsync()` (whose implementation is shown later):

```
addAsync(3, 4)
  .then(result => { // success
    assert.equal(result, 7);
  })
  .catch(error => { // failure
    assert.fail(error);
  });
```

Promises combine aspects of [the event pattern](#) and [the callback pattern](#):

- Like the callback pattern, Promises specialize in delivering one-off values.
- Like callback-based functions, Promise-based functions (and methods) are transformed versions of synchronous functions (and methods). They differ in how the result is delivered: With the callback pattern, it is delivered via 1–2 callbacks, passed as parameters. With the Promise pattern, it is delivered via a *Promise* – an object returned by the function.
- *How* a result is delivered via a Promise is similar to events: You register listeners for results and errors via the Promise.

Let's look at an example next.

### 37.1.1 Asynchronous callback-based code

Consider the following text file `person.json` with JSON data in it:

```
{
  "first": "Jane",
  "last": "Doe"
}
```

The following code reads the contents of this file and converts it to a JavaScript object. It is based on Node.js-style callbacks:

```
import * as fs from 'fs';
fs.readFile('person.json',
  function (error, text) {
```

```

if (error) { // (A)
  // Failure
  assert.fail(error);
} else {
  // Success
  try { // (B)
    const obj = JSON.parse(text); // (C)
    assert.deepEqual(obj, {
      first: 'Jane',
      last: 'Doe',
    });
  } catch (e) {
    // Invalid JSON
    assert.fail(e);
  }
}
});

```

`fs` is a built-in Node.js module for file system operations. We use the callback-based function `fs.readFile()` to read a file whose name is `person.json`. If we succeed, the content is delivered via the parameter `text`, as a string. In line C, we convert that string from the text-based data format *JSON* into a JavaScript object. *JSON* is part of JavaScript's standard library.

Note that there are two error handling mechanisms: The `if` in line A takes care of asynchronous errors reported by `fs.readFile()`, while the `try` in line B takes care of synchronous errors reported by `JSON.parse()`.

### 37.1.2 The same functionality via Promises

The following code uses `readFileAsync()`, a Promise-based version of `fs.readFile()` (created via `util.promisify()`, which is explained later):

```

readFileAsync('person.json')
.then(text => { // (A)
  // Success
  const obj = JSON.parse(text);
  assert.deepEqual(obj, {
    first: 'Jane',
    last: 'Doe',
  });
})
.catch(err => { // (B)
  // Failure: file I/O error or JSON syntax error
  assert.fail(err);
});

```

Function `readFileAsync()` returns a Promise. In line A, we specify a success callback via method `.then()` of that Promise. The remaining code in `then`'s callback is synchronous.

`.then()` returns a Promise, which enables the invocation of the Promise method `.catch()` in line B. We use it to specify a failure callback.

Note that `.catch()` lets us handle both the asynchronous errors of `readFileAsync()` and the synchronous errors of `JSON.parse()`. We'll see later how exactly that works.

### 37.1.3 What is a Promise?

So what is a Promise? There are two ways of looking at it:

- On one hand, it is a placeholder or container for the final result that will eventually be delivered.
- On the other hand, it is an object with which you can register listeners.

A key feature of Promises is, that successes and failures are handled by chained `.then()` and `.catch()` method calls. Chaining is enabled by both methods returning Promises. Details will be explained soon.

## 37.2 The basics of using Promises

This is how you implement a Promise-based function that adds two numbers `x` and `y`:

```
function addAsync(x, y) {
  return new Promise(
    (resolve, reject) => { // (A)
      if (x === undefined || y === undefined) {
        reject(new Error('Must provide two parameters'));
      } else {
        resolve(x + y);
      }
    }
  );
}
```

With this style of returning a Promise, `addAsync()` immediately invokes the Promise constructor. The actual implementation of that function resides in the callback that is passed to that constructor (line A). That callback is provided with two functions:

- `resolve` is used for delivering a result (in case of success).
- `reject` is used for delivering an error (in case of failure).

This is how you invoke `addAsync()` and handle its results and errors:

```
addAsync(3, 4)
  .then(result => {
    assert.equal(result, 7);
  })
  .catch(error => {
    assert.fail(error);
  });
```

### 37.2.1 States of promises

Fig. 37.1 depicts the three states a Promise can be in. Promises specialize in one-off results and protect you against *race conditions* (registering too early or too late):

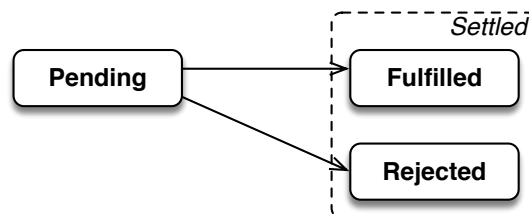


Figure 37.1: A Promise can be in either one of three states: pending, fulfilled, or rejected. If a Promise is in a final (non-pending) state, it is called *settled*.

- If you register a `.then()` callback or a `.catch()` callback too early, it is notified once a Promise is settled.
- Once a Promise is settled, the settlement value (result or error) is cached. Thus, if `.then()` or `.catch()` are called after the settlement, they receive the cached value.

Additionally, once a Promise is settled, its state and settlement value can't change, anymore. That helps make code predictable and enforces the one-off nature of Promises.

Next, we'll see more ways of creating Promises.

### 37.2.2 `Promise.resolve()`: create a Promise fulfilled with a given value

`Promise.resolve(x)` creates a Promise that is fulfilled with the value `x`:

```

Promise.resolve(123)
  .then(x => {
    assert.equal(x, 123);
  });
  
```

If the parameter is already a Promise, it is returned unchanged:

```

const abcPromise = Promise.resolve('abc');
assert.equal(
  Promise.resolve(abcPromise),
  abcPromise);
  
```

Therefore, given an arbitrary value `x`, you can use `Promise.resolve(x)` to ensure you have a Promise.

Note that the name is `resolve`, not `fulfill`, because `.resolve()` returns a rejected Promise if its Parameter is a rejected Promise.

### 37.2.3 `Promise.reject()`: create a Promise rejected with a given value

`Promise.reject(err)` creates a Promise that is fulfilled with the value `err`:

```

const myError = new Error('My error!');
Promise.reject(myError)
  .catch(err => {
    assert.equal(err, myError);
  });
  
```

### 37.2.4 Returning and throwing in `.then()` callbacks

`.then()` handles Promise fulfillments. It returns a fresh Promise. How that Promise is settled depends on what happens inside the callback. Let's look at three common cases.

First, the callback can return a non-Promise value (line A). Consequently, the Promise returned by `.then()` is fulfilled with that value (as checked in line B):

```
Promise.resolve('abc')
  .then(str => {
    return str + str; // (A)
  })
  .then(str2 => {
    assert.equal(str2, 'abcbc'); // (B)
  });
```

Second, the callback can return a Promise `p` (line A). Consequently, `p` “becomes” what `.then()` returns (the Promise that `.then()` has already returned is effectively replaced by `p`).

```
Promise.resolve('abc')
  .then(str => {
    return Promise.resolve(123); // (A)
  })
  .then(num => {
    assert.equal(num, 123);
  });
```

Third, the callback can throw an exception. Consequently, the Promise returned by `.then()` is rejected with that exception. That is, a synchronous error is converted into an asynchronous error.

```
const myError = new Error('My error!');
Promise.resolve('abc')
  .then(str => {
    throw myError;
  })
  .catch(err => {
    assert.equal(err, myError);
  });
```

### 37.2.5 `.catch()` and its callback

The only difference between `.then()` and `.catch()` is that the latter reacts to rejections, not to fulfillments. However, both methods turn the actions of their callbacks into Promises in the same manner. For example, in the following code, the value returned by the `.catch()` callback in line A becomes a fulfillment value:

```
retrieveFileName()
  .catch(() => {
    // Something went wrong, use a default value
    return 'Untitled.txt'; // (A)
  })
  .then(fileName => {
```

```
// ...
});
```

### 37.2.6 Chaining method calls

Due to `.then()` and `.catch()` always returning Promises, you can create arbitrary long chains of method calls:

```
function myAsyncFunc() {
  return asyncFunc1()
    .then(result1 => {
      // ...
      return asyncFunc2(); // a Promise
    })
    .then(result2 => {
      // ...
      return result2 || '(Empty)'; // not a Promise
    })
    .then(result3 => {
      // ...
      return asyncFunc4(); // a Promise
    });
}
```

In a way, `.then()` is the asynchronous version of the synchronous semicolon:

- `.then()` executes two asynchronous operations sequentially.
- The semicolon executes two synchronous operations sequentially.

You can also add `.catch()` into the mix and let it handle multiple error sources at the same time:

```
asyncFunc1()
  .then(result1 => {
    // ...
    return asyncFunction2();
  })
  .then(result2 => {
    // ...
  })
  .catch(error => {
    // Failure: handle errors of asyncFunc1(), asyncFunc2()
    // and any (sync) exceptions thrown in previous callbacks
  });
```

### 37.2.7 Advantages of promises

These are some of the advantages of Promises over plain callbacks when it comes to handling one-off results:

- The type signatures of Promise-based functions and methods are cleaner: If a function is callback-based, some parameters are about input, while the one or two callbacks at the end are about output. With Promises, everything output-related is handled via the returned value.
- Chaining asynchronous processing steps is more convenient.
- Error handling takes care of both synchronous and asynchronous errors.
- Composing Promise-based functions is slightly easier, because you can use some of the synchronous tools (e.g. `.map()`) that call functions and process results. We'll see an example at the end of this chapter.
- Promises are a single standard that is slowly replacing several, mutually incompatible alternatives. For example, in Node.js, many functions are now available in Promise-based versions. And new asynchronous browser APIs are usually Promise-based.

One of the biggest advantage of Promises involves not working with them directly: They are the foundation of *async functions*, a synchronous-looking syntax for performing asynchronous computations. Asynchronous functions are covered in the next chapter.

## 37.3 Examples

Seeing them in action helps with understanding Promises. Let's look at examples.

### 37.3.1 Browsers: Promisifying XMLHttpRequest

We have previously seen the event-based XMLHttpRequest API for downloading data in web browsers. The following function promisifies that API:

```
function httpGet(url) {
  return new Promise(
    function (resolve, reject) {
      const xhr = new XMLHttpRequest();
      xhr.onload = () => {
        if (xhr.status === 200) {
          resolve(xhr.responseText); // (A)
        } else {
          // Something went wrong (404 etc.)
          reject(new Error(xhr.statusText)); // (B)
        }
      }
      xhr.onerror = () => {
        reject(new Error('Network error')); // (C)
      };
      xhr.open('GET', url);
      xhr.send();
    });
}
```

Note how the results of XMLHttpRequest are handled via `resolve()` and `reject()`:



- A successful outcome leads to the returned Promise being fulfilled with it (line A).
- Errors lead to the Promise being rejected (lines B and C).

This is how you use `httpGet()`:

```
httpGet('http://example.com/textfile.txt')
  .then(content => {
    assert.equal(content, 'Content of textfile.txt\n');
  })
  .catch(error => {
    assert.fail(error);
  });
```



#### Exercise: Timing out a Promise

`exercises/promises/promise_timeout_test.js`

### 37.3.2 Node.js: `util.promisify()`

`util.promisify()` is a utility function that converts a callback-based function `f` into a Promise-based one. That is, we are going from this type signature:

`f(arg_1, ..., arg_n, (err: Error, result: T) => void) : void`

To this type signature:

`f(arg_1, ..., arg_n) : Promise<T>`

The following code promisifies the callback-based `fs.readFile()` (line A) and uses it:

```
import * as fs from 'fs';
import {promisify} from 'util';

const readFileAsync = promisify(fs.readFile); // (A)

readFileAsync('some-file.txt', {encoding: 'utf8'})
  .then(text => {
    assert.equal(text, 'The content of some-file.txt\n');
  })
  .catch(err => {
    assert.fail(err);
  });
```



#### Exercises: `util.promisify()`

- Using `util.promisify()`: `exercises/promises/read_file_async_exrc.js`
- Implementing `util.promisify()` yourself: `exercises/promises/my_promisify_test.js`

### 37.3.3 Browsers: Fetch API

All modern browsers support Fetch, a new Promise-based API for downloading data. Think of it as a Promise-based version of XMLHttpRequest. The following is an excerpt of the API<sup>1</sup>:

```
declare function fetch(str) : Promise<Response>;
interface Body {
  text() : Promise<string>;
  ...
}
interface Response extends Body {
  ...
}
```

That means, you can use `fetch()` as follows:

```
fetch('http://example.com/textfile.txt')
  .then(response => response.text())
  .then(text => {
    assert.equal(text, 'Content of textfile.txt\n');
  });
```



#### Exercise: Using the fetch API

`exercises/promises/fetch_json_test.js`

## 37.4 Promise-based functions start synchronously, settle asynchronously

Most Promise-based functions are executed as follows:

- Their execution starts right away, synchronously.
- But the Promise they return is guaranteed to be settled asynchronously (if ever).

The following code demonstrates that:

```
function asyncFunc() {
  console.log('asyncFunc');
  return new Promise(
    (resolve, _reject) => {
      console.log('Callback of new Promise()');
      resolve();
    }
  );
}
console.log('Start');
asyncFunc()
  .then(() => {
    console.log('Callback of .then()'); // (A)
```

<sup>1</sup><https://fetch.spec.whatwg.org/#fetch-api>

```
});
console.log('End');

// Output:
// 'Start'
// 'asyncFunc'
// 'Callback of new Promise()'
// 'End'
// 'Callback of .then()'
```

We can see that the callback of `new Promise()` is executed before the end of the code, while the result is delivered later (line A).

That means that your code can rely on run-to-completion semantics (as explained in [the previous chapter](#)) and that chaining Promises won't starve other tasks of processing time.

Additionally, this rule leads to Promise-based functions consistently returning results asynchronously. Not sometimes immediately, sometimes asynchronously. This kind of predictability makes code easier to work with. For more information, consult “Designing APIs for Asynchrony<sup>2</sup>” by Isaac Z. Schlueter.

## 37.5 Error handling: don't mix rejections and exceptions

The general rule for error handling in asynchronous code is:

Don't mix (asynchronous) rejections and (synchronous) exceptions

The rationale is that your code is less redundant if you can use a single error handling mechanism.

Alas, it is easy to accidentally break that rule. For example:

```
// Don't do this
function asyncFunc() {
  doSomethingSync(); // (A)
  return doSomethingAsync()
    .then(result => {
      // ...
    });
}
```

The problem is that, if an exception is thrown in line A, then `asyncFunc()` will throw an exception. Callers of that function only expect rejections and are not prepared for an exception. There are three ways in which we can fix this issue.

We can wrap the whole body of the function in a `try-catch` statement and return a rejected Promise if an exception is thrown:

```
// Solution 1
function asyncFunc() {
  try {
    doSomethingSync();
    return doSomethingAsync()
  } catch (err) {
    return Promise.reject(err);
  }
}
```

---

<sup>2</sup><http://blog.izs.me/post/59142742143/designing-apis-for-asynchrony>

```

    .then(result => {
      // ...
    });
  } catch (err) {
    return Promise.reject(err);
  }
}

```

Given that `.then()` converts exceptions to rejections, we can execute `doSomethingSync()` inside a `.then()` callback. To do so, we start a Promise chain via `Promise.resolve()`. We ignore the fulfillment value, undefined, of that initial Promise.

```

// Solution 2
function asyncFunc() {
  return Promise.resolve()
    .then(() => {
      doSomethingSync();
      return doSomethingAsync();
    })
    .then(result => {
      // ...
    });
}

```

Lastly, `new Promise()` also converts exceptions to rejections. Using this constructor is therefore similar to the previous solution:

```

// Solution 3
function asyncFunc() {
  return new Promise((resolve, reject) => {
    doSomethingSync();
    resolve(doSomethingAsync());
  })
    .then(result => {
      // ...
    });
}

```

## 37.6 Promise.all(): working with Arrays of Promises

### 37.6.1 Sequential execution vs. concurrent execution

Consider the following code:

```

const asyncFunc1 = () => Promise.resolve('one');
const asyncFunc2 = () => Promise.resolve('two');

asyncFunc1()
  .then(result1 => {
    assert.equal(result1, 'one');
  })

```

```
    return asyncFunc2();
  })
  .then(result2 => {
    assert.equal(result2, 'two');
  });
```

With `.then()`, Promise-based functions are always executed *sequentially*: Only after the result of `asyncFunc1()` is settled, will `asyncFunc2()` be executed.

In contrast, the helper function `Promise.all()` executes Promise-based functions in a manner that is more concurrent:

```
Promise.all([asyncFunc1(), asyncFunc2()])
  .then(arr => {
    assert.deepEqual(arr, ['one', 'two']);
  });
```

The parameter of `Promise.all()` is an Array of Promises. It returns a single Promise that is settled as follows:

- If and when all input Promises are fulfilled, the output Promise is fulfilled with an Array of the fulfillment values.
- If at least one input Promise is rejected, the output Promise is rejected with the rejection value of the input Promise.

In other words: You go from an Array of Promises to a Promise for an Array.

### 37.6.2 Focus on when computations are started

Tip for determining how “concurrent” asynchronous code is: Focus on when asynchronous computations are started, not on how Promises are handled. For example, the following code that uses `.then()`, is as “concurrent” as the version that uses `Promise.all()`:

```
const promise1 = asyncFunc1();
const promise2 = asyncFunc2();

promise1
  .then(result1 => {
    assert.equal(result1, 'one');
    return promise2;
  })
  .then(result2 => {
    assert.equal(result2, 'two');
  });
```

Both `asyncFunc1()` and `asyncFunc2()` are started at roughly the same time. Once both Promises are fulfilled, both `.then()` calls are executed almost immediately. If `promise1` is fulfilled first, this approach is even faster than using `Promise.all()` (which waits until all Promises are fulfilled).

### 37.6.3 `Promise.all()` is fork-join

`Promise.all()` is loosely related to the concurrency pattern “fork join”. For example:

```
Promise.all([
  // Fork async computations
  httpGet('http://example.com/file1.txt'),
  httpGet('http://example.com/file2.txt'),
])
// Join async computations
.then(([text1, text2]) => {
  assert.equal(text1, 'Content of file1.txt\n');
  assert.equal(text2, 'Content of file2.txt\n');
});
```

### 37.6.4 Asynchronous `.map()`

Array transformation methods such as `.map()`, `.filter()`, etc., are made for synchronous computations. For example:

```
function timesTwoSync(x) {
  return 2 * x;
}
const arr = [1, 2, 3];
const result = arr.map(timesTwoSync);
assert.deepEqual(result, [2, 4, 6]);
```

Is it possible for the callback of `.map()` to be a Promise-based function? Yes it is, if you use `Promise.all()` to convert an Array of Promises to an Array of (fulfillment) values:

```
function timesTwoAsync(x) {
  return new Promise(resolve => resolve(x * 2));
}
const arr = [1, 2, 3];
const promiseArr = arr.map(timesTwoAsync);
Promise.all(promiseArr)
  .then(result => {
    assert.deepEqual(result, [2, 4, 6]);
  });
```

#### 37.6.4.1 A more realistic use case

The following code demonstrates a more realistic use case: We are using `.map()` to convert the example from the section on fork-join, into a function whose parameter is an Array with URLs of text files to download.

```
function downloadTexts(fileUrls) {
  const promisedTexts = fileUrls.map(httpGet);
  return Promise.all(promisedTexts);
}
```

```
downloadTexts([
  'http://example.com/file1.txt',
  'http://example.com/file2.txt',
])
.then(texts => {
  assert.deepEqual(
    texts, [
      'Content of file1.txt\n',
      'Content of file2.txt\n',
    ]
  );
});
```



**Exercise: `Promise.all()` and listing files**

`exercises/promises/list_files_async_test.js`

## 37.7 Tips for using Promises

This section gives tips for using Promises.

### 37.7.1 Chaining mistake: losing the tail

Problem:

```
// Don't do this
function foo() {
  const promise = asyncFunc();
  promise.then(result => {
    // ...
  });

  return promise;
}
```

Computation starts with the Promise returned by `asyncFunc()`. But afterwards, computation continues and another Promise is created, via `.then()`. `foo()` returns the former Promise, but should return the latter. This is how to fix it:

```
function foo() {
  const promise = asyncFunc();
  return promise.then(result => {
    // ...
  });
}
```

### 37.7.2 Chaining mistake: nesting

Problem:

```
// Don't do this
asyncFunc1()
  .then(result1 => {
    return asyncFunc2()
      .then(result2 => { // (A)
        // ...
      });
  });
```

The `.then()` in line A is nested. A flat structure would be better:

```
asyncFunc1()
  .then(result1 => {
    return asyncFunc2();
  })
  .then(result2 => {
    // ...
  });
```

### 37.7.3 Chaining mistake: more nesting than necessary

This is another example of avoidable nesting:

```
// Don't do this
asyncFunc1()
  .then(result1 => {
    if (result1 < 0) {
      return asyncFuncA()
        .then(resultA => 'Result: ' + resultA);
    } else {
      return asyncFuncB()
        .then(resultB => 'Result: ' + resultB);
    }
  });
```

We can once again get a flat structure:

```
asyncFunc1()
  .then(result1 => {
    return result1 < 0 ? asyncFuncA() : asyncFuncB();
  })
  .then(resultAB => {
    return 'Result: ' + resultAB;
  });
```



### 37.7.4 Nesting per se is not evil

In the following code, we benefit from *not* nesting:

```
db.open()
.then(connection => { // (A)
  return connection.select({ name: 'Jane' })
  .then(result => { // (B)
    // Process result
    // Use `connection` to make more queries
  })
  // ...
  .catch(error => {
    // handle errors
  })
  .finally(() => {
    connection.close(); // (C)
  });
});
```

We are receiving an asynchronous result in line A. In line B, we are nesting, so that we have access to variable `connection` inside the callback and in line C.

### 37.7.5 Chaining mistake: creating Promises instead of chaining

Problem:

```
// Don't do this
class Model {
  insertInto(db) {
    return new Promise((resolve, reject) => { // (A)
      db.insert(this.fields)
        .then(resultCode => {
          this.notifyObservers({event: 'created', model: this});
          resolve(resultCode);
        }).catch(err => {
          reject(err);
        })
    });
  }
  // ...
}
```

In line A, we are creating a Promise to deliver the result of `db.insert()`. That is unnecessarily verbose and can be simplified:

```
class Model {
  insertInto(db) {
    return db.insert(this.fields)
      .then(resultCode => {
        this.notifyObservers({event: 'created', model: this});
      });
  }
}
```

```
        return resultCode;
    });
}
// ...
}
```

The key idea is that we don't need to create a Promise; we can return the result of the `.then()` call. An additional benefit is that we don't need to catch and re-reject the failure of `db.insert()`. We simply pass its rejection on, to the caller of `.insertInto()`.

## 37.8 Further reading

- “Exploring ES6<sup>3</sup>” goes deeper into Promises, including how they are implemented.

---

<sup>3</sup>[http://exploringjs.com/es6/ch\\_promises.html](http://exploringjs.com/es6/ch_promises.html)

## Chapter 38

# Async functions

### Contents

---

<b>38.1 Async functions: the basics</b>	<b>379</b>
38.1.1 Async constructs	380
38.1.2 Async functions always return Promises	381
38.1.3 Returned Promises are not wrapped	382
38.1.4 <code>await</code> : working with Promises	382
38.1.5 <code>await</code> and fulfilled Promises	382
38.1.6 <code>await</code> and rejected Promises	382
<b>38.2 Terminology</b>	<b>383</b>
<b>38.3 <code>await</code> is shallow (you can't use it in callbacks)</b>	<b>383</b>
<b>38.4 (Advanced)</b>	<b>384</b>
<b>38.5 Immediately invoked async arrow functions</b>	<b>384</b>
<b>38.6 Concurrency and <code>await</code></b>	<b>385</b>
38.6.1 <code>await</code> : running multiple asynchronous functions sequentially	385
38.6.2 <code>await</code> : running multiple asynchronous functions concurrently	385
<b>38.7 Tips for using async functions</b>	<b>386</b>
38.7.1 Async functions are started synchronously, settled asynchronously	386
38.7.2 You don't need <code>await</code> if you "fire and forget"	387
38.7.3 It can make sense to <code>await</code> and ignore the result	387

---

Roughly, *async functions* provide better syntax for code that uses Promises.

### 38.1 Async functions: the basics

Consider the following async function:

```
async function fetchJsonAsync(url) {  
  try {  
    const request = await fetch(url); // async
```

```

    const text = await request.text(); // async
    return JSON.parse(text); // sync
  }
  catch (error) {
    assert.fail(error);
  }
}

```

The previous rather synchronous-looking code is equivalent to the following Promise-based code:

```

function fetchJsonViaPromises(url) {
  return fetch(url) // async
    .then(request => request.text()) // async
    .then(text => {
      return JSON.parse(text); // sync
    })
    .catch(error => {
      assert.fail(error);
    });
}

```

A few observations about the async function `fetchJsonAsync()`:

- Async functions are marked with the keyword `async`.
- Inside the body of an async function, you write Promise-based code as if it were synchronous. You only need to apply the `await` operator whenever a value is a Promise. That operator pauses the async function and resumes it once the Promise is settled:
  - If the Promise is fulfilled, `await` returns the fulfillment value.
  - If the Promise is rejected, `await` throws the rejection value.
- The result of an async function is always a Promise:
  - Any value that is returned (explicitly or implicitly) is used to fulfill the Promise.
  - Any exception that is thrown is used to reject the Promise.

Both `fetchJsonAsync()` and `fetchJsonViaPromises()` are called in exactly the same way, like this:

```

fetchJsonAsync('http://example.com/person.json')
  .then(obj => {
    assert.deepEqual(obj, {
      first: 'Jane',
      last: 'Doe',
    });
  });

```

### 38.1.1 Async constructs

JavaScript has the following async versions of synchronous callable entities. Their roles are always either real function or method.

```

// Async function declaration
async function func1() {}

```

```
// Async function expression
const func2 = async function () {};

// Async arrow function
const func3 = async () => {};

// Async method definition (in classes, too)
const obj = { async m() {} };
```

### 38.1.2 Async functions always return Promises

Each async function always returns a Promise.

Inside the async function, you fulfill the result Promise via `return` (line A):

```
async function asyncFunc() {
  return 123; // (A)
}

asyncFunc()
  .then(result => {
    assert.equal(result, 123);
  });
```

As usual, if you don't explicitly return anything, `undefined` is returned for you:

```
async function asyncFunc() {
}

asyncFunc()
  .then(result => {
    assert.equal(result, undefined);
  });
```

You reject the result Promise via `throw` (line A):

```
let thrownError;
async function asyncFunc() {
  thrownError = new Error('Problem!');
  throw thrownError; // (A)
}

asyncFunc()
  .catch(err => {
    assert.equal(err, thrownError);
  });
```

### 38.1.3 Returned Promises are not wrapped

If you return a Promise *p* from an async function, then *p* becomes the result of the function (or rather, the result “locks in” on *p* and behaves exactly like it). That is, the Promise is not wrapped in yet another Promise.

```
async function asyncFunc() {
  return Promise.resolve('abc');
}

asyncFunc()
  .then(result => assert.equal(result, 'abc'));
```

Recall that any Promise *q* is treated similarly in the following situations:

- `resolve(q)` inside `new Promise((resolve, reject) => { ... })`
- `return q` inside `.then(result => { ... })`
- `return q` inside `.catch(err => { ... })`

### 38.1.4 `await`: working with Promises

The `await` operator can only be used inside async functions. Its operand is usually a Promise and leads to the following steps being performed:

- The current async function is paused.
- Processing of the task queue continues.
- Once the Promise is settled, execution of the async function resumes.
- If the Promise is fulfilled, `await` returns the fulfillment value. If the Promise is rejected, `await` throws the rejection value.

The following two sections provide more details.

### 38.1.5 `await` and fulfilled Promises

If its operand ends up being a fulfilled Promise, `await` returns its fulfillment value:

```
assert.equal(await Promise.resolve('yes!'), 'yes!');
```

Non-Promise values are allowed, too, and simply passed on (synchronously, without pausing the async function):

```
assert.equal(await 'yes!', 'yes!');
```

### 38.1.6 `await` and rejected Promises

If its operand is a rejected Promise, then `await` throws the rejection value:

```
try {
  await Promise.reject(new Error());
  assert.fail(); // we never get here
} catch (e) {
```

```
    assert.ok(e instanceof Error);
}
```

Instances of `Error` (which includes instances of its subclasses) are treated specially and also thrown:

```
try {
  await new Error();
  assert.fail(); // we never get here
} catch (e) {
  assert.ok(e instanceof Error);
}
```



#### Exercise: Fetch API, via async functions

`exercises/async-functions/fetch_json2_test.js`

## 38.2 Terminology

Let's clarify a few terms:

- *Async functions, async methods*: are defined using the keyword `async`. Async functions are also called *async/await*, based on the two keywords that are their syntactic foundation.
- *Directly using Promises*: means that code is handling Promises without `await`.
- *Promise-based*: a function or method that delivers its results and errors via Promises. That is, both async functions and functions that return Promises, qualify.
- *Asynchronous*: a function or method that delivers its results and errors asynchronously. Here, any operation that uses an asynchronous pattern (callbacks, events, Promises, etc.) qualifies. Alas, things are a bit confusing, because the “async” in “async function” is an abbreviation for “asynchronous”.

## 38.3 await is shallow (you can't use it in callbacks)

If you are inside an async function and want to pause it via `await`, you must do so within that function, you can't use it inside a nested function, such as a callback. That is, pausing is *shallow*.

For example, the following code can't be executed:

```
async function downloadContent(urls) {
  return urls.map(url => {
    return await httpGet(url); // SyntaxError!
  });
}
```

The reason is that normal arrow functions don't allow `await` inside their bodies.

OK, let's try an async arrow function, then:

```

async function downloadContent(urls) {
  return urls.map(async (url) => {
    return await httpGet(url);
  });
}

```

Alas, this doesn't work, either: Now `.map()` (and therefore `downloadContent()`) returns an Array with Promises, not an Array with (unwrapped) values.

One possible solution is to use `Promise.all()` to unwrap all Promises:

```

async function downloadContent(urls) {
  const promiseArray = urls.map(async (url) => {
    return await httpGet(url); // (A)
  });
  return await Promise.all(promiseArray);
}

```

Can this code be improved? Yes it can, because in line A, we are unwrapping a Promise via `await`, only to re-wrap it immediately via `return`. We can omit `await` and then don't even need an async arrow function:

```

async function downloadContent(urls) {
  const promiseArray = urls.map(
    url => httpGet(url));
  return await Promise.all(promiseArray); // (B)
}

```

For the same reason, we can also omit `await` in line B.



#### Exercise: Mapping and filtering asynchronously

`exercises/async-functions/map_async_test.js`

## 38.4 (Advanced)

All remaining sections are advanced.

## 38.5 Immediately invoked async arrow functions

If you need an `await` outside an async function (e.g., at the top level of a module), then you can use a technique that's similar to **immediately invoked function expressions**: you can immediately invoke an async arrow function:

```

(async () => { // start
  const promise = Promise.resolve('abc');
  const value = await promise;
  assert.equal(value, 'abc');
})(); // end

```



The result of an immediately invoked async arrow function is a Promise:

```
const promise = (async () => 123)();  
promise.then(x => assert.equal(x, 123));
```

Note that async arrow functions must always be wrapped in parentheses. For ordinary function expressions, you only need to do that in expression context.

## 38.6 Concurrency and await

### 38.6.1 await: running multiple asynchronous functions sequentially

If you prefix the invocations of multiple asynchronous functions with `await`, then those functions are executed sequentially:

```
const otherAsyncFunc1 = () => Promise.resolve('one');  
const otherAsyncFunc2 = () => Promise.resolve('two');  
  
async function asyncFunc() {  
  const result1 = await otherAsyncFunc1();  
  assert.equal(result1, 'one');  
  
  const result2 = await otherAsyncFunc2();  
  assert.equal(result2, 'two');  
}
```

That is, `otherAsyncFunc2()` is only started after `otherAsyncFunc1()` is completely finished.

### 38.6.2 await: running multiple asynchronous functions concurrently

If we want to run multiple functions concurrently, we need to resort to the tool method `Promise.all()`:

```
async function asyncFunc() {  
  const [result1, result2] = await Promise.all([  
    otherAsyncFunc1(),  
    otherAsyncFunc2(),  
  ]);  
  assert.equal(result1, 'one');  
  assert.equal(result2, 'two');  
}
```

Here, both asynchronous functions are started at the same time. Once both are settled, `await` gives us either an Array of fulfillment values or – if at least one Promise is rejected – an exception.

Recall from the previous chapter that what counts is when you start a Promise-based computation – not how you process its result. Therefore, the following code is as “concurrent” as the previous one:

```
async function asyncFunc() {  
  const promise1 = otherAsyncFunc1();  
  const promise2 = otherAsyncFunc2();
```

```
const result1 = await promise1;
const result2 = await promise2;

assert.equal(result1, 'one');
assert.equal(result2, 'two');
}
```

## 38.7 Tips for using async functions

### 38.7.1 Async functions are started synchronously, settled asynchronously

Async functions are executed as follows:

- The Promise *p* for the result is created when the async function is started.
- Then the body is executed. There are two ways in which execution can leave the body:
  - Execution can leave permanently, while settling *p*:
    - \* A `return` fulfills *p*.
    - \* A `throw` rejects *p*.
  - Execution can also leave temporarily, when awaiting the settlement of another Promise *q* via `await`. The async function is paused and execution leaves it. It is resumed once *q* is settled.
- Promise *p* is returned after execution has left the body for the first time (permanently or temporarily).

Note that the notification of the settlement of the result *p* happens asynchronously, as is always the case with Promises.

The following code demonstrates that an async function is started synchronously (line A), then the current task finishes (line C), then the result Promise is settled – asynchronously (line B).

```
async function asyncFunc() {
  console.log('asyncFunc() starts'); // (A)
  return 'abc';
}
asyncFunc().
then(x => { // (B)
  console.log(`Resolved: ${x}`);
});
console.log('Task ends'); // (C)

// Output:
// 'asyncFunc() starts'
// 'Task ends'
// 'Resolved: abc'
```

### 38.7.2 You don't need `await` if you "fire and forget"

`await` is not required when working with a Promise-based function, you only need it if you want to pause and wait until the returned Promise is settled. If all you want to do, is start an asynchronous operation, then you don't need it:

```
async function asyncFunc() {  
  const writer = openFile('someFile.txt');  
  writer.write('hello'); // don't wait  
  writer.write('world'); // don't wait  
  await writer.close(); // wait for file to close  
}
```

In this code, we don't `await` `.write()`, because we don't care when it is finished. We do, however, want to wait until `.close()` is done.

### 38.7.3 It can make sense to `await` and ignore the result

It can occasionally make sense to use `await`, even if you ignore its result. For example:

```
await longRunningAsyncOperation();  
console.log('Done!');
```

Here, we are using `await` to join a long-running asynchronous operation. That ensures that the logging really happens *after* that operation is done.



## Chapter 39

# Asynchronous iteration

### Contents

<b>39.1 Basic asynchronous iteration</b>	<b>389</b>
39.1.1 Protocol: async iteration	389
39.1.2 Example: async iteration manually	390
39.1.3 Example: async iteration via <code>for-await-of</code>	391
<b>39.2 Asynchronous generators</b>	<b>392</b>
39.2.1 Example: creating an async iterable manually via an async generator	393
39.2.2 Example: converting a sync iterable to an async iterable	393
39.2.3 Example: transforming an async iterable	394
39.2.4 Example: mapping over asynchronous iterables	395
<b>39.3 Async iteration over Node.js streams</b>	<b>396</b>
39.3.1 Node.js streams: async via callbacks (push)	396
39.3.2 Node.js streams: async via async iteration (pull)	396
39.3.3 Example: from chunks to lines	396



### Useful knowledge

For this chapter, it helps to be familiar with [Promises](#) and [async functions](#).

## 39.1 Basic asynchronous iteration

### 39.1.1 Protocol: async iteration

To understand how asynchronous iteration works, let's first revisit [synchronous iteration](#). It comprises the following interfaces:

```
interface Iterable<T> {  
  [Symbol.iterator]() : Iterator<T>;  
}
```

```

interface Iterator<T> {
  next() : IteratorResult<T>;
}
interface IteratorResult<T> {
  value: T;
  done: boolean;
}

```

- An `Iterable` is a data structure whose contents can be accessed via iteration. It is a factory for iterators.
- An `Iterator` is a factory for iteration results, that you retrieve by calling the method `.next()`.
- After the iterator has delivered all iterated values, property `.done` of the iteration results switches from `false` to `true`.

For the protocol for asynchronous iteration, we only want to change one thing: The values produced by `.next()` should be delivered asynchronously. There are two conceivable options:

- The `.value` could contain a `Promise<T>`.
- `.next()` could return `Promise<IteratorResult<T>>`.

In other words: The question is whether to wrap just values or whole iterator results in Promises.

It has to be the latter, because when `.next()` returns a result, it starts an asynchronous computation. Whether or not that computation produces a value or signals the end of the iteration, can only be determined after it is finished. Therefore, both `.done` and `.value` need to be wrapped in a Promise. The interfaces for async iteration look as follows.

```

interface AsyncIterable<T> {
  [Symbol.asyncIterator]() : AsyncIterator<T>;
}
interface AsyncIterator<T> {
  next() : Promise<IteratorResult<T>>; // (A)
}
interface IteratorResult<T> {
  value: T;
  done: boolean;
}

```

The only difference to the synchronous interfaces is the return type of `.next()` (line A).

### 39.1.2 Example: async iteration manually

The following code uses the asynchronous iteration protocol manually:

```

const asyncIterable = createAsyncIterable(['a', 'b']); // (A)
const asyncIterator = asyncIterable[Symbol.asyncIterator]();

// Call .next() until .done is true:
asyncIterator.next() // (B)
.then(iteratorResult => {
  assert.deepStrictEqual(
    iteratorResult,

```

```

    { value: 'a', done: false });
    return asyncIterator.next(); // (C)
  })
  .then(iteratorResult => {
    assert.deepEqual(
      iteratorResult,
      { value: 'b', done: false });
    return asyncIterator.next(); // (D)
  })
  .then(iteratorResult => {
    assert.deepEqual(
      iteratorResult,
      { value: undefined, done: true });
  })
;

```

In line A, we create an asynchronous iterable over the value 'a' and 'b'. We'll see an implementation of `createAsyncIterable()` later.

We call `.next()` in line B, line C and line D. Each time, we use `.next()` to unwrap the Promise and `assert.deepEqual()` to check the unwrapped value.

We can simplify this code if we use an async function. Now we unwrap Promises via `await` and the code looks synchronous:

```

async function f() {
  const asyncIterable = createAsyncIterable(['a', 'b']);
  const asyncIterator = asyncIterable[Symbol.asyncIterator]();

  // Call .next() until .done is true:
  assert.deepEqual(
    await asyncIterator.next(),
    { value: 'a', done: false });
  assert.deepEqual(
    await asyncIterator.next(),
    { value: 'b', done: false });
  assert.deepEqual(
    await asyncIterator.next(),
    { value: undefined, done: true });
}

```

### 39.1.3 Example: async iteration via `for-await-of`

The asynchronous iteration protocol is not meant to be used manually. One of the language constructs that supports it, is the `for-await-of` loop, which is an asynchronous version of the `for-of` loop. It can be used in async functions and *async generators* (which are introduced later in this chapter). This is an example of `for-await-of` in use:

```

for await (const x of createAsyncIterable(['a', 'b'])) {
  console.log(x);
}

```

```
// Output:
// 'a'
// 'b'
```

`for-await-of` is relatively flexible. In addition to asynchronous iterables, also supports synchronous iterables:

```
for await (const x of ['a', 'b']) {
  console.log(x);
}
// Output:
// 'a'
// 'b'
```

And it supports synchronous iterables over values that are wrapped in Promises:

```
const arr = [Promise.resolve('a'), Promise.resolve('b')];
for await (const x of arr) {
  console.log(x);
}
// Output:
// 'a'
// 'b'
```



#### Exercise: Convert an async iterable to an Array

Warning: You'll soon see the solution for this exercise in this chapter. \* `exercises/async-iteration/async_iterable_to_array_test.js`

## 39.2 Asynchronous generators

An asynchronous generator is two things at the same time:

- An async function (input): You can use `await` and `for-await-of` to retrieve data.
- A generator that returns an asynchronous iterable (output): You can use `yield` and `yield*` to produce data.



#### Asynchronous generators are very similar to synchronous generators

Due to async generators and sync generators being so similar, I don't explain how exactly `yield` and `yield*` work. Please consult [the chapter on synchronous generators](#) if you have doubts.

Therefore, an asynchronous generator has:

- Input that can be:
  - synchronous (values, sync iterables) or
  - asynchronous (Promises, async iterables).
- Output that is an asynchronous iterable.

This looks as follows:



```

async function* asyncGen() {
  // Input
  const x = await somePromise;
  for await (const y of someAsyncIterable) {
    // ...
  }

  // Output
  yield someValue;
  yield* otherAsyncGen();
}

```

### 39.2.1 Example: creating an async iterable manually via an async generator

Let's look at an example. The following code creates an async iterable with three numbers:

```

async function* yield123() {
  for (let i=1; i<=3; i++) {
    yield i;
  }
}

```

Does the result of `yield123()` conform to the async iteration protocol?

```

(async () => {
  const asyncIterable = yield123();
  const asyncIterator = asyncIterable[Symbol.asyncIterator]();
  assert.deepEqual(
    await asyncIterator.next(),
    { value: 1, done: false });
  assert.deepEqual(
    await asyncIterator.next(),
    { value: 2, done: false });
  assert.deepEqual(
    await asyncIterator.next(),
    { value: 3, done: false });
  assert.deepEqual(
    await asyncIterator.next(),
    { value: undefined, done: true });
})();

```

We wrapped the code in an *immediately invoked async arrow function*.

### 39.2.2 Example: converting a sync iterable to an async iterable

The following asynchronous generator converts a synchronous iterable to an asynchronous iterable. It implements the function `createAsyncIterable()` that we have used previously.

```

async function* createAsyncIterable(syncIterable) {
  for (const elem of syncIterable) {

```

```

    yield elem;
  }
}

```

Note: The input is synchronous in this case (no `await` is needed).

### 39.2.3 Example: transforming an async iterable

Let's implement an async generator that produces a new async iterable by transforming an existing async iterable.

```

async function* timesTwo(asyncNumbers) {
  for await (const x of asyncNumbers) {
    yield x * 2;
  }
}

```

In order to test `timesTwo()`, we use the solution of an exercise – the implementation of the function `asyncIterableToArray()`. Note that for this implementation, we can't use an async generator: We get our input via `for-await-of` and return an `Array` wrapped in a `Promise`. The latter requirement rules out async generators.

```

async function asyncIterableToArray(asyncIterable) {
  let result = [];
  for await (const value of asyncIterable) {
    result.push(value);
  }
  return result;
}

```

And finally, the actual test. We create test data via the async generator `yield123()`.

```

async function* yield123() {
  for (let i=1; i<=3; i++) {
    yield i;
  }
}

const asyncIterable = yield123();
assert.deepEqual(
  await asyncIterableToArray(timesTwo(asyncIterable)), // (A)
  [2, 4, 6]
);

```

Note the `await` in line A, which is needed to unwrap the `Promise` returned by `asyncIterableToArray()`. In order for `await` to work, this code fragment must be run inside an async function.



#### Exercise: Async generators

Warning: You'll soon see the solution for this exercise in this chapter. \* `exercises/async-iteration/number_lines_test.js`

### 39.2.4 Example: mapping over asynchronous iterables

As a reminder – this is how to map over synchronous iterables:

```
function* mapSync(iterable, func) {
  let index = 0;
  for (const x of iterable) {
    yield func(x, index);
    index++;
  }
}

const syncIterable = mapSync(['a', 'b', 'c'], s => s.repeat(3));
assert.deepEqual(
  [...syncIterable],
  ['aaa', 'bbb', 'ccc']);
```

The asynchronous version looks as follows:

```
async function* mapAsync(asyncIterable, func) { // (A)
  let index = 0;
  for await (const x of asyncIterable) { // (B)
    yield func(x, index);
    index++;
  }
}
```

Note how similar the sync implementation and the async implementation are. The only two differences are the `async` in line A and the `await` in line B. That is comparable to going from a synchronous function to an asynchronous function – you only need to add the keyword `async` and the occasional `await`.

To test `mapAsync()`, we use the helper functions `asyncIterableToArray()` (shown previously in this chapter) and `createAsyncIterable()` (shown below):

```
async function* createAsyncIterable() {
  yield 'a';
  yield 'b';
}

const asyncIterable = mapAsync(['a', 'b', 'c'], s => s.repeat(3));
assert.deepEqual(
  await asyncIterableToArray(asyncIterable), // (A)
  ['aaa', 'bbb', 'ccc']);
```

Once again, we `await` to unwrap a Promise (line A) and this code fragment must run inside an `async` function.



#### Exercise: `filterAsyncIter()`

`exercises/async-iteration/filter_async_iter_test.js`

## 39.3 Async iteration over Node.js streams

### 39.3.1 Node.js streams: async via callbacks (push)

Traditionally, reading asynchronously from Node.js streams is done via callbacks:

```
function main(inputFilePath) {
  const readStream = fs.createReadStream(inputFilePath,
    { encoding: 'utf8', highWaterMark: 1024 });
  readStream.on('data', (chunk) => {
    console.log('>>> ' + chunk);
  });
  readStream.on('end', () => {
    console.log('### DONE ###');
  });
}
```

That is, the stream is in control and pushes data to the reader.

### 39.3.2 Node.js streams: async via async iteration (pull)

Starting with Node.js 10, you can also use asynchronous iteration to read from streams:

```
async function main(inputFilePath) {
  const readStream = fs.createReadStream(inputFilePath,
    { encoding: 'utf8', highWaterMark: 1024 });

  for await (const chunk of readStream) {
    console.log('>>> ' + chunk);
  }
  console.log('### DONE ###');
}
```

That is, the reader is in control and pulls data from the stream.

### 39.3.3 Example: from chunks to lines

Node.js streams iterate over *chunks* (arbitrarily long pieces) of data. The following asynchronous generator converts an async iterable over chunks to an async iterable over lines

```
/**
 * Parameter: async iterable of chunks (strings)
 * Result: async iterable of lines (incl. newlines)
 */
async function* chunksToLines(chunksAsync) {
  let previous = '';
  for await (const chunk of chunksAsync) { // input
    previous += chunk;
    let eolIndex;
```

```

    while ((eolIndex = previous.indexOf('\n')) >= 0) {
      // line includes the EOL (Windows '\r\n' or Unix '\n')
      const line = previous.slice(0, eolIndex+1);
      yield line; // output
      previous = previous.slice(eolIndex+1);
    }
  }
  if (previous.length > 0) {
    yield previous;
  }
}

```

Let's apply `chunksToLines()` to an async iterable over chunks (as produced by `genChunks()`):

```

async function* genChunks() {
  yield 'First\nSec';
  yield 'ond\nThird\nF';
  yield 'ourth';
}
const linesIterable = chunksToLines(genChunks());
assert.deepEqual(
  await asyncIterableToArray(linesIterable),
  [
    'First\n',
    'Second\n',
    'Third\n',
    'Fourth',
  ]
);

```

Now that we have an asynchronous iterable over lines, we can use the solution of a previous exercise, `numberLines()`, to number those lines:

```

async function* numberLines(linesAsync) {
  let lineNumber = 1;
  for await (const line of linesAsync) {
    yield lineNumber + ': ' + line;
    lineNumber++;
  }
}
const numberedLines = numberLines(chunksToLines(genChunks()));
assert.deepEqual(
  await asyncIterableToArray(numberedLines),
  [
    '1: First\n',
    '2: Second\n',
    '3: Third\n',
    '4: Fourth',
  ]
);

```



## **Part IX**

# **More standard library**





## Chapter 40

# Regular expressions (RegExp)

### Contents

---

<b>40.1 Creating regular expressions</b>	<b>402</b>
40.1.1 Literal vs. constructor	402
40.1.2 Cloning and non-destructively modifying regular expressions	402
<b>40.2 Syntax</b>	<b>403</b>
40.2.1 Syntax characters	403
40.2.2 Basic atoms	403
40.2.3 Character classes	404
40.2.4 Groups	405
40.2.5 Quantifiers	405
40.2.6 Assertions	405
40.2.7 Disjunction ( <code>()</code> )	406
<b>40.3 Flags</b>	<b>406</b>
40.3.1 Flag: Unicode mode via <code>/u</code>	407
<b>40.4 Properties of regular expression objects</b>	<b>409</b>
40.4.1 Flags as properties	409
40.4.2 Other properties	409
<b>40.5 Methods for working with regular expressions</b>	<b>410</b>
40.5.1 <code>RegExp.test(str)</code> : is there a match?	410
40.5.2 <code>str.search(RegExp)</code> : at what index is the match?	410
40.5.3 <code>RegExp.exec(str)</code> : capturing groups	410
40.5.4 <code>str.match(RegExp)</code> : return all matching substrings	412
40.5.5 <code>str.replace(searchValue, replacementValue)</code>	412
40.5.6 Other methods for working with regular expressions	414
<b>40.6 Flag <code>/g</code> and its pitfalls</b>	<b>414</b>
40.6.1 Problem: You can't inline a regular expression with flag <code>/g</code>	414
40.6.2 Problem: Removing <code>/g</code> can break code	415
40.6.3 Problem: Adding <code>/g</code> can break code	415
40.6.4 Problem: Code can break if <code>.lastIndex</code> isn't zero	415
40.6.5 Dealing with <code>/g</code> and <code>.lastIndex</code>	416

40.7 Techniques for working with regular expressions . . . . .	417
40.7.1 Escaping arbitrary text for regular expressions . . . . .	417
40.7.2 Matching everything or nothing . . . . .	417



### Availability of features

Unless stated otherwise, all regular expression features are supported by ES5 and later.

## 40.1 Creating regular expressions

### 40.1.1 Literal vs. constructor

The two main ways of creating regular expressions, are:

- Literal: `/abc/ui`, compiled statically (at load time).
- Constructor: `new RegExp('abc', 'ui')`, compiled dynamically (at runtime)
  - The second parameter is optional.

Both regular expressions have the same two parts:

- The *body* `abc` – the actual regular expression.
- The *flags* `u` and `i`. Flags configure how the pattern is interpreted: `u` switches on the *Unicode mode*. `i` enables case-insensitive matching.

### 40.1.2 Cloning and non-destructively modifying regular expressions

There are two variants of the constructor `RegExp()`:

- `new RegExp(pattern : string, flags = '')`  
A new regular expression is created as specified via `pattern`. If `flags` is missing, the empty string `''` is used.
- `new RegExp(regExp : RegExp, flags = regExp.flags)` <sup>[ES6]</sup>  
`regExp` is cloned. If `flags` is provided then it determines the flags of the copy.

The second variant is useful for cloning regular expressions, optionally while modifying them. Flags are immutable and this is the only way of changing them. For example:

```
function copyAndAddFlags(regExp, flags='') {
  // The constructor doesn't allow duplicate flags,
  // make sure there aren't any:
  const newFlags = [...new Set(regExp.flags + flags)].join('');
  return new RegExp(regExp, newFlags);
}
assert.equal(/abc/i.flags, 'i');
assert.equal(copyAndAddFlags(/abc/i, 'g').flags, 'gi');
```

## 40.2 Syntax

### 40.2.1 Syntax characters

At the top level of a regular expression, the following *syntax characters* are special. They are escaped by prefixing a backslash (\).

```
\ ^ $ . * + ? ( ) [ ] { } |
```

In regular expression literals, you must also escape the slash (not necessary with new `RegExp()`):

```
> /\./.test('/')
true
> new RegExp('/').test('/')
true
```

### 40.2.2 Basic atoms

*Atoms* are the basic building blocks of regular expressions.

- Pattern characters: are all characters *except* syntax characters (^, \$, etc.). Pattern characters match themselves. Examples: A b %
- . matches any character. You can use the flag /s (dotall) to control if the dot matches line terminators or not ([more below](#)).
- Character escapes (each escape matches a single fixed character):
  - Control escapes (for a few control characters):
    - \* \f: form feed (FF)
    - \* \n: line feed (LF)
    - \* \r: carriage return (CR)
    - \* \t: character tabulation
    - \* \v: line tabulation
  - Arbitrary control characters: \cA (Ctrl-A), \cB (Ctrl-B), etc.
  - Unicode code units: \u00E4
  - Unicode code points (require flag /u): \u{1F44D}
- Character class escapes (each escape matches one out of a set of characters):
  - \d: digits (same as [0-9])
    - \* \D: non-digits
  - \w: “word” characters (same as [A-Za-z0-9\_])
    - \* \W: non-word characters
  - \s: whitespace (space, tab, line terminators, etc.)
    - \* \S: non-whitespace
  - Unicode property escapes (ES2018): \p{White\_Space}, \P{White\_Space}, etc.
    - \* Require flag /u.
    - \* Described in the next subsection.

#### 40.2.2.1 Unicode property escapes

Unicode property escapes look like this:

1. `\p{prop=value}`: matches all characters whose property `prop` has the value `value`.
2. `\P{prop=value}`: matches all characters that do not have a property `prop` whose value is `value`.
3. `\p{bin_prop}`: matches all characters whose binary property `bin_prop` is `True`.
4. `\P{bin_prop}`: matches all characters whose binary property `bin_prop` is `False`.

Comments:

- You can only use Unicode property escapes if the flag `/u` is set. Without `/u`, `\p` is the same as `p`.
- Forms (3) and (4) can be used as abbreviations if the property is `General_Category`. For example, `\p{Lowercase_Letter}` is an abbreviation for `\p{General_Category=Lowercase_Letter}`

Examples:

- Checking for whitespace:
 

```
> /^[\p{White_Space}]+$/u.test('\t \n\r')
true
```
- Checking for Greek letters:
 

```
> /^[\p{Script=Greek}]+$/u.test('μετά')
true
```
- Deleting any letters:
 

```
> '1n2ü3é4'.replace(/[\p{Letter}]/ug, '')
'1234'
```
- Deleting lowercase letters:
 

```
> 'AbCdEf'.replace(/[\p{Lowercase_Letter}]/ug, '')
'ACE'
```

Further reading:

- Lists of Unicode properties and their values: “Unicode Standard Annex #44: Unicode Character Database<sup>1</sup>” (Editors: Mark Davis, Laurențiu Iancu, Ken Whistler)
- Unicode property escapes in more depth: Chapter “RegExp Unicode property escapes<sup>2</sup>” in “Exploring ES2018 and ES2019”

### 40.2.3 Character classes

- Match one of a set of characters: `[abc]`
  - Match any character not in a set: `[^abc]`
- Inside the square brackets, only the following characters are special and must be escaped:
 

```
^ \ - ]
```

`^` only has to be escaped if it comes first. `-` need not be escaped if it comes first or last
- Character escapes (`\n`, `\u{1F44D}`) and character class escapes (`\d`, `\p{White_Space}`) work as usual.

<sup>1</sup><https://unicode.org/reports/tr44/#Properties>

<sup>2</sup>[http://exploringjs.com/es2018-es2019/ch\\_regexp-unicode-property-escapes.html](http://exploringjs.com/es2018-es2019/ch_regexp-unicode-property-escapes.html)

- Exception: Inside square brackets, `\b` matches backspace. Elsewhere, it matches word boundaries.
- Character ranges are specified via dashes: `[a-z]`, `[^a-z]`

#### 40.2.4 Groups

- Positional capturing group: `(#+)`
  - Backreference: `\1`, `\2`, etc.
- Named capturing group (ES2018): `(?<hashes>#+)`
  - Backreference: `\k<hashes>`
- Noncapturing group: `(?:#+)`

#### 40.2.5 Quantifiers

By default, all of the following quantifiers are greedy:

- `?`: match never or once
- `*`: match zero or more times
- `+`: match one or more times
- `{n}`: match `n` times
- `{n,}`: match `n` or more times
- `{n,m}`: match at least `n` times, at most `m` times.

To make them reluctant, put question marks (`?`) after them:

```
> /".*"/.exec('"abc"def"')[0] // greedy
'"abc"def"'
> /".*?"/.exec('"abc"def"')[0] // reluctant
'"abc"'
```

#### 40.2.6 Assertions

- `^` matches only at the beginning of the input
- `$` matches only at the end of the input
- `\b` matches only at a word boundary
  - `\B` matches only when not at a word boundary
- Lookahead:
  - `(?=«pattern»)` matches if `pattern` matches what comes next (positive lookahead). Example (“sequences of lowercase letters that are followed by an `X`” – note that the `X` itself is not part of the matched substring):
 

```
> 'abcX def'.match(/[a-z]+(?=X)/g)
[ 'abc' ]
```
  - `(?!«pattern»)` matches if `pattern` does not match what comes next (negative lookahead). Example (“sequences of lowercase letters that are not followed by an `X`”):
 

```
> 'abcX def'.match(/[a-z]+(?!X)/g)
[ 'ab', 'def' ]
```

- Further reading: “RegExp lookbehind assertions<sup>3</sup>” in “Exploring ES2018 and ES2019” (covers lookahead assertions, too)
- Lookbehind (ES2018):
  - `(?<=«pattern»)` matches if `pattern` matches what came before (positive lookbehind)
 

```
> 'Xabc def'.match(/(?<=X)[a-z]+/g)
[ 'abc' ]
```
  - `(?<!=«pattern»)` matches if `pattern` does not match what came before (negative lookbehind)
 

```
> 'Xabc def'.match(/(?<!=X)[a-z]+/g)
[ 'bc', 'def' ]
```
  - Further reading: “RegExp lookbehind assertions<sup>4</sup>” in “Exploring ES2018 and ES2019”

### 40.2.7 Disjunction (|)

Caveat: this operator has low precedence. Use groups if necessary:

- `^aa|zz$` matches all strings that start with `aa` and/or end with `zz`. Note that `|` has a lower precedence than `^` and `$`.
- `^(aa|zz)$` matches the two strings `'aa'` and `'zz'`.
- `^a(a|z)z$` matches the two strings `'aaz'` and `'azz'`.

## 40.3 Flags

Table 40.1: These are the regular expression flags supported by JavaScript.

Literal flag	Property name	ES	Description
<code>g</code>	<code>global</code>	ES3	Match multiple times
<code>i</code>	<code>ignoreCase</code>	ES3	Match case-insensitively
<code>m</code>	<code>multiline</code>	ES3	<code>^</code> and <code>\$</code> match per line
<code>s</code>	<code>dotall</code>	ES2018	Dot matches line terminators
<code>u</code>	<code>unicode</code>	ES6	Unicode mode (recommended)
<code>y</code>	<code>sticky</code>	ES6	No characters between matches

The following regular expression flags are available in JavaScript (tbl. 40.1 provides a compact overview):

- `/g` (`.global`): fundamentally changes how the methods `RegExp.prototype.test()`, `RegExp.prototype.exec()` and `String.prototype.match()` work. It is explained in detail along with these methods. In a nutshell: Without `/g`, the methods only consider the first match for a regular expression in an input string. With `/g`, they consider all matches.
- `/i` (`.ignoreCase`): switches on case-insensitive matching:
 

```
> /a/.test('A')
false
```

<sup>3</sup>[http://exploringjs.com/es2018-es2019/ch\\_regexp-lookbehind-assertions.html](http://exploringjs.com/es2018-es2019/ch_regexp-lookbehind-assertions.html)

<sup>4</sup>[http://exploringjs.com/es2018-es2019/ch\\_regexp-lookbehind-assertions.html](http://exploringjs.com/es2018-es2019/ch_regexp-lookbehind-assertions.html)

```
> /a/i.test('A')
true
```

- */m* (*.multiline*): If this flag is on, *^* matches the beginning of each line and *\$* matches the end of each line. If it is off, *^* matches the beginning of the whole input string and *\$* matches the end of the whole input string.

```
> 'a1\na2\na3'.match(/^a./gm)
[ 'a1', 'a2', 'a3' ]
> 'a1\na2\na3'.match(/^a./g)
[ 'a1' ]
```

- */u* (*.unicode*): This flag switches on the Unicode mode for a regular expression. That mode is explained in the next subsection.
- */y* (*.sticky*): This flag only makes sense in conjunction with */g*. When both are switched on, any match after the first one must directly follow the previous match (without any characters between them).

```
> 'a1a2 a3'.match(/a./gy)
[ 'a1', 'a2' ]
> 'a1a2 a3'.match(/a./g)
[ 'a1', 'a2', 'a3' ]
```

- */s* (*.dotall*): By default, the dot does not match line terminators. With this flag, it does:

```
> /. /.test('\n')
false
> /. /s.test('\n')
true
```

Alternative for older ECMAScript versions:

```
> /[^]/.test('\n')
true
```

### 40.3.1 Flag: Unicode mode via */u*

The flag */u* switches on a special Unicode mode for a regular expression. That mode enables several features:

- In patterns, you can use Unicode code point escapes such as `\u{1F42A}` to specify characters. Code unit escapes such as `\u03B1` only have a range of four hexadecimal digits (which equals the basic multilingual plane).
- In patterns, you can use Unicode property escapes (ES2018) such as `\p{White_Space}`.
- Many escapes are now forbidden (which enables the previous feature):

```
> /\a/
/\a/
> /\a/u
SyntaxError: Invalid regular expression: /\a/: Invalid escape

> /\-/
```

```

/\-/
> /\-/u
SyntaxError: Invalid regular expression: /\-/: Invalid escape

> /\:/
/\:/
> /\:/u
SyntaxError: Invalid regular expression: /\:/: Invalid escape

```

- The atomic units for matching (“characters”) are code points, not code units.

The following subsections explain the last item in more detail. They use the following Unicode character to explain when the atomic units are code points and when they are code units:

```

const codePoint = 'Ⓢ';
const codeUnits = '\uD83D\uDE42'; // UTF-16

assert.equal(codePoint, codeUnits); // same string!

```

I’m only switching between Ⓢ and \uD83D\uDE42, to illustrate how JavaScript sees things. Both are equivalent and can be used interchangeably in strings and regular expressions.

#### 40.3.1.1 Consequence: you can put code points in character classes

With /u, the two code units of Ⓢ are interpreted as a single character:

```

> /^[Ⓢ]$/.test('Ⓢ')
true

```

Without /u, Ⓢ is interpreted as two characters:

```

> /^[\uD83D\uDE42]$/.test('\uD83D\uDE42')
false
> /^[\uD83D\uDE42]$/.test('\uDE42')
true

```

Note that ^ and \$ demand that the input string have a single character. That’s why the first result is false.

#### 40.3.1.2 Consequence: the dot operator (.) matches code points, not code units

With /u, the dot operator matches code points (.match() plus /g returns an Array with all the matches of a regular expression):

```

> 'Ⓢ'.match(/./gu).length
1

```

Without /u, the dot operator matches single code units:

```

> '\uD83D\uDE42'.match(/./g).length
2

```



### 40.3.1.3 Consequence: quantifiers apply to code points, not code units

With `/u`, a quantifier applies to the whole preceding code point:

```
> /^@{3}$/u.test('@@@')
true
```

Without `/u`, a quantifier only applies to the preceding code unit:

```
> /^\\uD83D\\uDE80{3}$/u.test('\\uD83D\\uDE80\\uDE80\\uDE80')
true
```

## 40.4 Properties of regular expression objects

Noteworthy:

- Strictly speaking, only `.lastIndex` is a real instance property. All other properties are implemented via getters.
- Accordingly, `.lastIndex` is the only mutable property. All other properties are read-only. If you want to change them, you need to copy the regular expression (consult [the section on cloning for details](#)).

### 40.4.1 Flags as properties

Each regular expression flag exists as a property, with a longer, more descriptive name:

```
> /a/i.ignoreCase
true
> /a/.ignoreCase
false
```

This is the complete list of flag properties:

- `.dotall (/s)`
- `.global (/g)`
- `.ignoreCase (/i)`
- `.multiline (/m)`
- `.sticky (/y)`
- `.unicode (/u)`

### 40.4.2 Other properties

Each regular expression also has the following properties:

- `.source`: The regular expression pattern.  

```
> /abc/ig.source
'abc'
```
- `.flags`: The flags of the regular expression.

```
> /abc/ig.flags  
'gi'
```

- `.lastIndex`: Used when flag `/g` is switched on. Consult [the section on that flag](#) for details.

## 40.5 Methods for working with regular expressions

### 40.5.1 `RegExp.test(str)`: is there a match?

The regular expression method `.test()` returns `true` if `RegExp` matches `str`:

```
> /abc/.test('ABC')  
false  
> /abc/i.test('ABC')  
true  
> /\.js$/.test('main.js')  
true
```

With `.test()` you should normally avoid the `/g` flag. If you use it, you generally don't get the same result every time you call the method:

```
> const r = /a/g;  
> r.test('aab')  
true  
> r.test('aab')  
true  
> r.test('aab')  
false
```

The results are due to `/a/` having two matches in the string. After all of those were found, `.test()` returns `false`.

### 40.5.2 `str.search(RegExp)`: at what index is the match?

The string method `.search()` returns the first index of `str` at which there is a match for `RegExp`:

```
> '_abc_'.search(/abc/)  
1  
> 'main.js'.search(/\.js$/)  
4
```

### 40.5.3 `RegExp.exec(str)`: capturing groups

#### 40.5.3.1 Getting a match object for the first match

Without the flag `/g`, `.exec()` returns all captures of the first match for `RegExp` in `str`:

```
assert.deepEqual(
  /(a+)b/.exec('ab aab'),
  {
    0: 'ab',
    1: 'a',
    index: 0,
    input: 'ab aab',
    groups: undefined,
  }
);
```

The result is a *match object* with the following properties:

- [0]: the complete substring matched by the regular expression
- [1]: capture of positional group 1 (etc.)
- .index: where did the match occur?
- .input: the string that was matched against
- .groups: captures of named groups

#### 40.5.3.2 Named groups (ES2018)

The previous example contained a single positional group. The following example demonstrates named groups:

```
const regexp = /^(?<key>[A-Za-z]+): (?<value>.*$)/u;
assert.deepEqual(
  regexp.exec('first: Jane'),
  {
    0: 'first: Jane',
    1: 'first',
    2: 'Jane',
    index: 0,
    input: 'first: Jane',
    groups: { key: 'first', value: 'Jane' },
  }
);
```

As you can see, the named groups `key` and `value` also exist as positional groups.

#### 40.5.3.3 Looping over multiple matches

If you want to retrieve all matches of a regular expression (not just the first one), you need to switch on the flag `/g`. Then you can call `.exec()` multiple times and get another match each time. After the last match, `.exec()` returns `null`.

```
> const regexp = /(a+)b/g;
> regexp.exec('ab aab')
{ 0: 'ab', 1: 'a', index: 0, input: 'ab aab', groups: undefined }
> regexp.exec('ab aab')
{ 0: 'aab', 1: 'aa', index: 3, input: 'ab aab', groups: undefined }
```

```
> regexp.exec('ab aab')
null
```

Therefore, you can loop over all matches as follows:

```
const regexp = /(a+)b/g;
const str = 'ab aab';

let match;
// Check for null via truthiness
// Alternative: while ((match = regexp.exec(str)) !== null)
while (match = regexp.exec(str)) {
  console.log(match[1]);
}
// Output:
// 'a'
// 'aa'
```

Sharing regular expressions with `/g` has a few pitfalls, which are explained [later](#).



**Exercise: Extract quoted text via `.exec()`**

`exercises/reg-exp/extract_quoted_test.js`

#### 40.5.4 `str.match(regExp)`: return all matching substrings

Without `/g`, `.match()` works like `.exec()` – it returns a single match object.

With `/g`, `.match()` returns all substrings of `str` that match `regExp`:

```
> 'ab aab'.match(/(a+)b/g) // important: /g
[ 'ab', 'aab' ]
```

If there is no match, `.match()` returns `null`:

```
> 'xyz'.match(/(a+)b/g)
null
```

You can use the Or operator to protect yourself against `null`:

```
const numberOfMatches = (str.match(regExp) || []).length;
```

#### 40.5.5 `str.replace(searchValue, replacementValue)`

`.replace()` has several different modes, depending on what values you provide for its parameters:

- `searchValue` is ...
  - a regular expression without `/g`: replace first occurrence.
  - a regular expression with `/g`: replace all occurrences.
  - a string: replace first occurrence (the string is interpreted verbatim, not as a regular expression). Alas, that means that strings are of limited use as search values. Later in this chapter, you'll find [a tool function for turning an arbitrary text into a regular expression](#).

- `replacementValue` is ...
  - a string: describe replacement
  - a function: compute replacement

The next subsections assume that a regular expression with `/g` is being used.

#### 40.5.5.1 `replacementValue` is a string

If the replacement value is a string, the dollar sign has special meaning – it inserts things matched by the regular expression:

Text	Result
<code>\$\$</code>	single <code>\$</code>
<code>\$&amp;</code>	complete match
<code>\$`</code>	text before match
<code>\$'</code>	text after match
<code>\$n</code>	capture of positional group <code>n</code> ( <code>n &gt; 0</code> )
<code>\$&lt;name&gt;</code>	capture of named group <code>name</code>

Example: Inserting the text before, inside, and after the matched substring.

```
> 'a1 a2'.replace(/a/g, "($`|$&|$')")
'(|a|1 a2)1 (a1 |a|2)2'
```

Example: Inserting the captures of positional groups.

```
> const regExp = /^( [A-Za-z]+ ): (.* )$/ug;
> 'first: Jane'.replace(regExp, 'KEY: $1, VALUE: $2')
'KEY: first, VALUE: Jane'
```

Example: Inserting the captures of named groups.

```
> const regExp = /^(?<key>[A-Za-z]+): (?<value>.* )$/ug;
> 'first: Jane'.replace(regExp, 'KEY: $<key>, VALUE: $<value>')
'KEY: first, VALUE: Jane'
```

#### 40.5.5.2 `replacementValue` is a function

If the replacement value is a function, you can compute each replacement. In the following example, we multiply each non-negative integer, that we find, by two.

```
assert.equal(
  '3 cats and 4 dogs'.replace(/[0-9]+/g, (all) => 2 * Number(all)),
  '6 cats and 8 dogs'
);
```

The replacement function gets the following parameters. Note how similar they are to match objects. The parameters are all positional, but I've included how one usually names them:

- `all`: complete match
- `g1`: capture of positional group 1

- Etc.
- `index`: where did the match occur?
- `input`: the string that was matched against
- `groups`: captures of named groups (an object)



**Exercise: Change quotes via `.replace()` and a named group**

`exercises/reg-exp/change_quotes_test.js`

### 40.5.6 Other methods for working with regular expressions

The first parameter of `String.prototype.split()` is either a string or a regular expression. If it is the latter then substrings captured by groups are added to the result of the method:

```
> 'a : b : c'.split(/( *):( *)/)
[ 'a', ' ', ' ', ' ', 'b', ' ', ' ', ' ', 'c' ]
```

Consult [the chapter on strings](#) for more information.

## 40.6 Flag `/g` and its pitfalls

The following two regular expression methods do something unusual if `/g` is switched on:

- `RegExp.prototype.exec()`
- `RegExp.prototype.test()`

Then they can be called repeatedly and deliver all matches inside a string. Property `.lastIndex` of the regular expression is used to track the current position inside the string. For example:

```
const r = /a/g;
assert.equal(r.lastIndex, 0);

assert.equal(r.test('aa'), true); // 1st match?
assert.equal(r.lastIndex, 1); // after 1st match

assert.equal(r.test('aa'), true); // 2nd match?
assert.equal(r.lastIndex, 2); // after 2nd match

assert.equal(r.test('aa'), false); // 3rd match?
assert.equal(r.lastIndex, 0); // start over
```

So how is flag `/g` problematic? We'll first explore the problems and then solutions.

### 40.6.1 Problem: You can't inline a regular expression with flag `/g`

A regular expression with `/g` can't be inlined: For example, in the following while loop, the regular expression is created fresh, every time the condition is checked. Therefore, its `.lastIndex` is always zero and the loop never terminates.

```
let count = 0;
// Infinite loop
while (/a/g.test('babaa')) {
  count++;
}
```

### 40.6.2 Problem: Removing /g can break code

If code expects a regular expression with /g and has a loop over the results of `.exec()` or `.test()` then a regular expression without /g can cause an infinite loop:

```
const regexp = /a/; // Missing: flag /g

let count = 0;
// Infinite loop
while (regexp.test('babaa')) {
  count++;
}
```

Why? Because `.test()` always returns the first result, true, and never false.

### 40.6.3 Problem: Adding /g can break code

With `.test()`, there is another caveat: If you want to check exactly once if a regular expression matches a string then the regular expression must not have /g. Otherwise, you generally get a different result, every time you call `.test()`:

```
> const r = /^X/g;
> r.test('Xa')
true
> r.test('Xa')
false
```

Normally, you won't add /g if you intend to use `.test()` in this manner. But it can happen if, e.g., you use the same regular expression for testing and for replacing. Or if you get the regular expression via a parameter.

### 40.6.4 Problem: Code can break if `.lastIndex` isn't zero

When a regular expression is created, `.lastIndex` is initialized to zero. If code ever receives a regular expression whose `.lastIndex` is not zero, it can break. For example:

```
const regexp = /a/g;
regexp.lastIndex = 4;

let count = 0;
while (regexp.test('babaa')) {
  count++;
}
```

```

}
assert.equal(count, 1); // should be 3

```

.lastIndex not being zero can happen relatively easily if a regular expression is shared and not handled properly.

### 40.6.5 Dealing with /g and .lastIndex

Consider the following scenario: You want to implement a function `countOccurrences(regExp, str)` that counts how often `regExp` has a match inside `str`. How do you prevent a wrong `regExp` from breaking your code? Let's look at three approaches.

First, you can throw exceptions if `/g` isn't set or `.lastIndex` isn't zero:

```

function countOccurrences(regExp, str) {
  if (!regExp.global) {
    throw new Error('Flag /g of regExp must be set');
  }
  if (regExp.lastIndex !== 0) {
    throw new Error('regExp.lastIndex must be zero');
  }

  let count = 0;
  while (regExp.test(str)) {
    count++;
  }
  return count;
}

```

Second, you can clone the parameter. That has the added benefit that `regExp` won't be changed.

```

function countOccurrences(regExp, str) {
  const cloneFlags = regExp.flags + (regExp.global ? 'g' : '');
  const clone = new RegExp(regExp, cloneFlags);

  let count = 0;
  while (clone.test(str)) {
    count++;
  }
  return count;
}

```

Third, you can use `.match()` to count occurrences – which doesn't change or depend on `.lastIndex`.

```

function countOccurrences(regExp, str) {
  if (!regExp.global) {
    throw new Error('Flag /g of regExp must be set');
  }
  return (str.match(regExp) || []).length;
}

```



## 40.7 Techniques for working with regular expressions

### 40.7.1 Escaping arbitrary text for regular expressions

The following function escapes an arbitrary text so that it is matched verbatim if you put it inside a regular expression:

```
function escapeForRegExp(str) {
  return str.replace(/[\^\$. *+?() \[\]{}|]/g, '\\$&'); // (A)
}
assert.equal(escapeForRegExp('[yes?]', String.raw`\[yes\?\]`);
assert.equal(escapeForRegExp('_g_'), String.raw`\_g_\`);
```

In line A, we escape all syntax characters. Note that `/u` forbids many escapes: among others, `\:` and `\-`.

This is how you can use `escapeForRegExp()` to replace an arbitrary text multiple times:

```
> const re = new RegExp(escapeForRegExp('/:)'), 'ug');
> ':-) :-) :-)'.replace(re, '☺')
'☺ ☺ ☺'
```

### 40.7.2 Matching everything or nothing

Sometimes, you may need a regular expression that matches everything or nothing. For example, as a sentinel value.

- Match everything: `/(?:)/` (the empty group matches everything; making it noncapturing avoids unnecessary work)

```
> /(?:)/.test('')
true
> /(?:)/.test('abc')
true
```

- Match nothing: `/.^/` (once matching has progressed beyond the first character, `^` doesn't match, anymore)

```
> /.^/.test('')
false
> /.^/.test('abc')
false
```



# Chapter 41

## Dates (Date)

### Contents

---

<b>41.1 Best practice: don't use the current built-in API</b>	<b>419</b>
<b>41.2 Background: UTC vs. GMT</b>	<b>420</b>
<b>41.3 Background: date time formats (ISO)</b>	<b>420</b>
<b>41.4 Time values</b>	<b>421</b>
41.4.1 Creating time values	421
41.4.2 Getting and setting time values	421
<b>41.5 Creating Dates</b>	<b>422</b>
<b>41.6 Getters and setters</b>	<b>422</b>
41.6.1 Time unit getters and setters	422
<b>41.7 Converting Dates to strings</b>	<b>423</b>
41.7.1 Strings with times	423
41.7.2 Strings with dates	423
41.7.3 Strings with dates and times	423
<b>41.8 Pitfalls of the Date API</b>	<b>424</b>

---

This chapter describes JavaScript's API for working with dates – Date.

### 41.1 Best practice: don't use the current built-in API

The JavaScript Date API is cumbersome to use. Hence, it's best to rely on a library for anything related to dates. Two examples are:

- `js-joda`<sup>1</sup>
- `date-fns`<sup>2</sup>

Additionally, TC39 is working on a new date API for JavaScript: `temporal`<sup>3</sup>.

---

<sup>1</sup><https://github.com/js-joda/js-joda>

<sup>2</sup><https://github.com/date-fns/date-fns>

<sup>3</sup><https://github.com/maggiepint/proposal-temporal>

## 41.2 Background: UTC vs. GMT

UTC (Coordinated Universal Time) and GMT (Greenwich Mean Time) have the same current time, but they are different things:

- UTC: is the time standard that all times zones are based on. They are specified relative to it. That is, no country or territory has UTC as its local time zone.
- GMT: is a time zone used in some European and African countries.

Source: “The Difference Between GMT and UTC<sup>4</sup>” at TimeAndDate.com

## 41.3 Background: date time formats (ISO)

Date time formats describe:

- The strings accepted by:
  - `Date.parse()`
  - `new Date()`
- The strings returned by (always longest format):
  - `Date.prototype.toISOString()`

The following is an example of a date time string returned by `.toISOString()`:

`'2033-05-28T15:59:59.123Z'`

Date time formats have the following structures:

- Date formats: Y=year; M=month; D=day
  - YYYY-MM-DD
  - YYYY-MM
  - YYYY
- Time formats: T=separator (the string 'T'); H=hour; m=minute; s=second and millisecond; Z=time zone is UTC (the string 'Z')
  - THH:mm:ss.sss
  - THH:mm:ss.sssZ
  - THH:mm:ss
  - THH:mm:ssZ
  - THH:mm
  - THH:mmZ
- Date time formats: are date formats followed by time formats.
  - For example (longest): YYYY-MM-DDTHH:mm:ss.sssZ

Alternative to Z – time zones relative to UTC:

- +hh:mm

---

<sup>4</sup><https://www.timeanddate.com/time/gmt-utc-time.html>

- -hh:mm

## 41.4 Time values

A *time value* represents a date via the number of milliseconds since 1 January 1970 00:00:00 UTC.

Time values can be used to create Dates:

```
const timeValue = 0;
assert.equal(
  new Date(timeValue).toISOString(),
  '1970-01-01T00:00:00.000Z');
```

Coercing a Date to a number returns its time value:

```
> Number(new Date(123))
123
```

Ordering operators coerce their operands to numbers. Therefore, you can use these operators to compare Dates:

```
assert.equal(new Date('1972-05-03') < new Date('2001-12-23'), true);
// Internally:
assert.equal(736992000000 < 1009065600000, true);
```

### 41.4.1 Creating time values

The following methods create time values:

- `Date.now()`: number  
Returns the current time as a time value.
- `Date.parse(dateTimeString)`: number (local time zone)  
Parses `dateTimeString` and returns the corresponding time value.
- `Date.UTC(year, month, date?, hours?, minutes?, seconds?, milliseconds?)`: number  
Returns the time value for the specified UTC date time.

### 41.4.2 Getting and setting time values

- `Date.prototype.getTime()`: number  
Returns the time value corresponding to the Date.
- `Date.prototype.setTime(timeValue)`  
Sets this to the date encoded by `timeValue`.

## 41.5 Creating Dates

- `new Date(year: number, month: number, date?: number, hours?: number, minutes?: number, seconds?: number, milliseconds?: number)` (local time zone)
 

```
> new Date(2077,0,27, 21,49,58, 888).toISOString() // CET (UTC+1)
'2077-01-27T20:49:58.888Z'
```
- `new Date(dateTimeStr: string)` (UTC)
 

```
> new Date('2077-01-27').toISOString()
'2077-01-27T00:00:00.000Z'
```
- `new Date(timeValue: number)`

```
> new Date(0).toISOString()
'1970-01-01T00:00:00.000Z'
```
- `new Date()` (same as `new Date(Date.now())`)

## 41.6 Getters and setters

### 41.6.1 Time unit getters and setters

Dates have getters and setters for time units. For example:

- `Date.prototype.getFullYear()`
- `Date.prototype.setFullYear(num)`

These getters and setters conform to the following patterns:

- Local time:
  - `Date.prototype.get«Unit»()`
  - `Date.prototype.set«Unit»(num)`
- Universal time:
  - `Date.prototype.getUTC«Unit»()`
  - `Date.prototype.setUTC«Unit»(num)`

These are the time units that are supported:

- Date
  - FullYear
  - Month: month (0–11). **Pitfall:** 0 is January, etc.
  - Date: day of the month (1–31)
  - Day (getter only): day of the week (0–6); 0 is Sunday
- Time
  - Hours: hour (0–23)
  - Minutes: minutes (0–59)
  - Seconds: seconds (0–59)
  - Milliseconds: milliseconds (0–999)

There is one more getter that doesn't conform to the previously mentioned patterns:

- `Date.prototype.getTimezoneOffset()`

Returns the time difference between local time and UTC in minutes. For example, for CET, it returns -60.

## 41.7 Converting Dates to strings

Example Date:

```
const d = new Date(0);
```

### 41.7.1 Strings with times

- `Date.prototype.toString()` (local time zone)  

```
> d.toString()  
'01:00:00 GMT+0100 (Central European Standard Time)'
```
- `Date.prototype.toLocaleTimeString()` (see ECMAScript internationalization API<sup>5</sup>)

### 41.7.2 Strings with dates

- `Date.prototype.toString()` (local time zone)  

```
> d.toString()  
'Thu Jan 01 1970'
```
- `Date.prototype.toLocaleDateString()` (ECMAScript internationalization API<sup>6</sup>)

### 41.7.3 Strings with dates and times

- `Date.prototype.toString()` (local time zone)  

```
> d.toString()  
'Thu Jan 01 1970 01:00:00 GMT+0100 (Central European Standard Time)'
```
- `Date.prototype.toLocaleString()` (see ECMAScript internationalization API<sup>7</sup>)
- `Date.prototype.toUTCString()` (UTC)  

```
> d.toUTCString()  
'Thu, 01 Jan 1970 00:00:00 GMT'
```
- `Date.prototype.toISOString()` (UTC)  

```
> d.toISOString()  
'1970-01-01T00:00:00.000Z'
```

---

<sup>5</sup>[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Intl](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl)

<sup>6</sup>[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Intl](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl)

<sup>7</sup>[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Intl](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl)

**Exercise: Creating a date string**

`exercises/dates/create_date_string_test.js`

## 41.8 Pitfalls of the Date API

- You can't specify a local time zone. That can lead to location-specific bugs if you are not careful. For example, the following interaction leads to different results, depending on where it is executed. We execute it in CET, which is why the ISO string (in UTC) has a different day of the month (26 vs. 27).

```
> new Date(2077, 0, 27).toISOString()  
'2077-01-26T23:00:00.000Z'
```

- January is 0, February is 1, etc.:

```
> new Date(1979, 0, 27, 10, 30).toISOString()  
'1979-01-27T09:30:00.000Z'
```

- For years  $y$  with  $0 \leq y \leq 99$ , 1900 is added:

```
> new Date(12, 1, 22, 19, 11).toISOString()  
'1912-02-22T18:11:00.000Z'
```



## Chapter 42

# Creating and parsing JSON (JSON)

### Contents

---

<b>42.1 The discovery and standardization of JSON</b>	<b>426</b>
42.1.1 JSON's grammar is frozen	426
<b>42.2 JSON syntax</b>	<b>426</b>
<b>42.3 Using the JSON API</b>	<b>427</b>
42.3.1 JSON.stringify(value, replacer?, space?)	427
42.3.2 JSON.parse(text, reviver?)	428
42.3.3 Example: converting to and from JSON	428
<b>42.4 Configuring what is stringified or parsed (advanced)</b>	<b>429</b>
42.4.1 .stringify(): specifying the only properties that objects should have	429
42.4.2 .stringify() and .parse(): value visitors	430
42.4.3 Example: visiting values	430
42.4.4 Example: stringifying unsupported values	431
42.4.5 Example: parsing unsupported values	431
<b>42.5 FAQ</b>	<b>432</b>
42.5.1 Why doesn't JSON support comments?	432
<b>42.6 Quick reference: JSON</b>	<b>432</b>
42.6.1 Sources	433

---

JSON ("JavaScript Object Notation") is a storage format that uses text to encode data. Its syntax is a subset of JavaScript expressions. As an example, consider the following data, stored as text in a file `jane.json`:

```
{
  "first": "Jane",
  "last": "Porter",
  "married": true,
  "born": 1890,
  "friends": [ "Tarzan", "Cheeta" ]
}
```

JavaScript has the global namespace object `JSON` provides methods for creating and parsing JSON.

## 42.1 The discovery and standardization of JSON

A specification for JSON was published by Douglas Crockford in 2001, at [json.org](http://json.org)<sup>1</sup>. He explains:

I discovered JSON. I do not claim to have invented JSON, because it already existed in nature. What I did was I found it, I named it, I described how it was useful. I don't claim to be the first person to have discovered it; I know that there are other people who discovered it at least a year before I did. The earliest occurrence I've found was, there was someone at Netscape who was using JavaScript array literals for doing data communication as early as 1996, which was at least five years before I stumbled onto the idea.

Later, JSON was standardized as ECMA-404<sup>2</sup>:

- 1st edition: October 2013
- 2nd edition: December 2017

### 42.1.1 JSON's grammar is frozen

Quoting the ECMA-404 standard:

Because it is so simple, it is not expected that the JSON grammar will ever change. This gives JSON, as a foundational notation, tremendous stability.

Therefore, JSON will never get improvements such as optional trailing commas, comments or unquoted keys – independently of whether or not they are considered desirable. However, that still leaves room for creating supersets of JSON that compile to plain JSON.

## 42.2 JSON syntax

JSON consists of the following parts of JavaScript:

- Compound:
  - Object literals:
    - \* Keys are double-quoted strings.
    - \* Values are JSON values.
    - \* No trailing commas are allowed.
  - Array literals:
    - \* Elements are JSON values.
    - \* No holes or trailing commas are allowed.
- Atomic:
  - null (but not undefined)
  - Booleans
  - Numbers (excluding NaN, +Infinity, -Infinity)
  - Strings (must be double-quoted)

As a consequence, you can't (directly) represent cyclic structures in JSON.

---

<sup>1</sup><http://json.org/>

<sup>2</sup><https://www.ecma-international.org/publications/standards/Ecma-404.htm>

## 42.3 Using the JSON API

The global namespace object `JSON` contains methods for working with JSON data.

### 42.3.1 `JSON.stringify(value, replacer?, space?)`

`.stringify()` converts a JavaScript value to a JSON string.

#### 42.3.1.1 Result: a single line of text

If you only provide the first argument, `.stringify()` returns a single line of text:

```
assert.equal(
  JSON.stringify({foo: ['a', 'b']}),
  `{"foo":["a","b"]}`);
```

#### 42.3.1.2 Result: a tree of indented lines

If you provide a non-negative integer for `space` (we are ignoring `replacer` here, which is explained [later](#)), then `.stringify()` returns one or more lines and indents by `space` spaces per level of nesting:

```
assert.equal(
  JSON.stringify({foo: ['a', 'b']}, null, 2),
  `{
    "foo": [
      "a",
      "b"
    ]
  }`);
```

#### 42.3.1.3 Details on how JavaScript values are stringified

Supported primitive values are stringified as expected:

```
> JSON.stringify('abc')
'"abc"'
> JSON.stringify(123)
'123'
> JSON.stringify(null)
'null'
```

Non-finite numbers (incl. `NaN`) are stringified as `'null'`:

```
> JSON.stringify(NaN)
'null'
> JSON.stringify(Infinity)
'null'
```

Unsupported primitive values are stringified as `undefined`:

```
> JSON.stringify(undefined)
undefined
> JSON.stringify(Symbol())
undefined
```

Functions are stringified as undefined:

```
> JSON.stringify(function () {})
undefined
```

In an Array, elements that would be stringified as undefined, are stringified as 'null':

```
> JSON.stringify([undefined, 123, () => {}])
'[null,123,null]'
```

In an object (that is neither an Array nor a function), properties, whose values would be stringified as undefined, are skipped:

```
> JSON.stringify({a: Symbol(), b: true})
'{"b":true}'
```

If an object (which may be an Array or a function) has a method `.toJSON()`, then the result of that method is stringified, instead of the object. For example, Dates have a method `.toJSON()` that returns strings.

```
> JSON.stringify({toJSON() {return true}})
'true'
> JSON.stringify(new Date(2999, 11, 31))
'"2999-12-30T23:00:00.000Z"'
```

For more details on stringification, consult the ECMAScript specification<sup>3</sup>.

### 42.3.2 JSON.parse(text, reviver?)

`.parse()` converts a JSON text to a JavaScript value:

```
> JSON.parse('{"foo":["a","b"]}')
```

```
{ foo: [ 'a', 'b' ] }
```

The parameter `reviver` is explained [later](#).

### 42.3.3 Example: converting to and from JSON

The following class demonstrates one technique for implementing the conversion from and to JSON:

```
class Point {
  static fromJson(jsonObj) {
    return new Point(jsonObj.x, jsonObj.y);
  }

  constructor(x, y) {
    this.coord = [x, y];
  }
}
```

---

<sup>3</sup><https://tc39.github.io/ecma262/#sec-serializejsonproperty>

```

    toJSON() {
      const [x, y] = this.coord;
      return {x, y};
    }
  }
}
assert.equal(
  JSON.stringify(new Point(3, 5)),
  '{"x":3,"y":5}');
assert.deepEqual(
  Point.fromJson(JSON.parse('{"x":3,"y":5}')),
  new Point(3, 5));

```

The **previously mentioned** method `.toJSON()` is used when stringifying instances of `Point`.



#### Exercise: Convert an object to and from JSON

`exercises/json/to_from_json_test.js`

## 42.4 Configuring what is stringified or parsed (advanced)

What is stringified or parsed, can be configured as follows:

- `.stringify()` has the optional parameter `replacer` that contains either:
  - An Array with names of properties. When stringifying an object (that may be nested) only those properties will be considered, all others will be ignored.
  - A *value visitor* – a function that can transform JavaScript values before they are stringified.
- `.parse()` has the optional parameter `reviver` – a value visitor that can transform the parsed JSON data before it is returned.

### 42.4.1 `.stringify()`: specifying the only properties that objects should have

If the second parameter of `.stringify()` is an Array, then only object properties, whose names are mentioned in the Array, are included in the result:

```

const obj = {
  a: 1,
  b: {
    c: 2,
    d: 3,
  }
};
assert.equal(
  JSON.stringify(obj, ['b', 'c']),
  '{"b":{"c":2}}');

```

### 42.4.2 .stringify() and .parse(): value visitors

What I call a *value visitor* is a function that transforms JavaScript values (compound or atomic):

- `JSON.stringify()` calls the value visitor in its parameter `replacer` before it stringifies the JavaScript value it received.
- `JSON.parse()` calls the value visitor in its parameter `reviver` after it parsed the JSON data.

A JavaScript value is transformed as follows: The value is either atomic or compound and contains more values (nested in Arrays and objects). The atomic value or the nested values are fed to the value visitor, one at a time. Depending on what the visitor returns, the current value is removed, changed or preserved.

A value visitor has the following type signature:

```
type ValueVisitor = (this: object, key: string, value: any) => any;
```

The parameters are:

- `value`: The current value.
- `this`: Parent of current value. The parent of the root value `r` is `{ '': r }`.
- `key`: Key or index of the current value inside its parent. The empty string is used for the root value.

The value visitor can return:

- `value`: means there won't be any change.
- A different value `x`: leads to `value` being replaced with `x`.
- `undefined`: leads to `value` being removed.

### 42.4.3 Example: visiting values

The following code demonstrates in which order a value visitor sees values.

```
const log = [];
function valueVisitor(key, value) {
  log.push({key, value, this: this});
  return value; // no change
}

const root = {
  a: 1,
  b: {
    c: 2,
    d: 3,
  }
};
JSON.stringify(root, valueVisitor);
assert.deepEqual(log, [
  { key: '', value: root, this: { '': root } },
  { key: 'a', value: 1, this: root },
  { key: 'b', value: root.b, this: root },
  { key: 'c', value: 2, this: root.b },
  { key: 'd', value: 3, this: root.b },
]);
```

As you can see, `.stringify()` visits values top-down (root first, leaves last). In contrast, `.parse()` visits values bottom-up (leaves first, root last).

#### 42.4.4 Example: stringifying unsupported values

`.stringify()` has no special support for regular expression objects – it stringifies them as if they were plain objects:

```
const obj = {
  name: 'abc',
  regex: /abc/ui,
};
assert.equal(
  JSON.stringify(obj),
  '{"name":"abc","regex":{}}');
```

We can fix that via a replacer:

```
function replacer(key, value) {
  if (value instanceof RegExp) {
    return {
      __type__: 'RegExp',
      source: value.source,
      flags: value.flags,
    };
  } else {
    return value; // no change
  }
}
assert.equal(
  JSON.stringify(obj, replacer, 2),
  `{
  "name": "abc",
  "regex": {
    "__type__": "RegExp",
    "source": "abc",
    "flags": "iu"
  }
}`);
```

#### 42.4.5 Example: parsing unsupported values

To `.parse()` the result from the previous section, we need a reviver:

```
function reviver(key, value) {
  // Very simple check
  if (value && value.__type__ === 'RegExp') {
    return new RegExp(value.source, value.flags);
  } else {
    return value;
  }
}
```

```

    return value;
  }
}
const str = `{
  "name": "abc",
  "regex": {
    "__type__": "RegExp",
    "source": "abc",
    "flags": "iu"
  }
}`;
assert.deepEqual(
  JSON.parse(str, reviver),
  {
    name: 'abc',
    regex: /abc/ui,
  });

```

## 42.5 FAQ

### 42.5.1 Why doesn't JSON support comments?

Douglas Crockford explains why in a Google+ post from 1 May 2012<sup>4</sup>:

I removed comments from JSON because I saw people were using them to hold parsing directives, a practice which would have destroyed interoperability. I know that the lack of comments makes some people sad, but it shouldn't.

Suppose you are using JSON to keep configuration files, which you would like to annotate. Go ahead and insert all the comments you like. Then pipe it through JSMIn [a minifier for JavaScript] before handing it to your JSON parser.

## 42.6 Quick reference: JSON

Signature of *value visitors*:

```
type ValueVisitor = (this: object, key: string, value: any) => any;
```

JSON:

- `.stringify(value: any, replacer?: ValueVisitor, space?: string | number): string` <sup>[ES5]</sup>

Convert value to a JSON string. The parameter `replacer` is explained [earlier in this chapter](#). The parameter `space` works as follows:

- If `space` is omitted, `.stringify()` returns a single line of text.

---

<sup>4</sup><https://plus.google.com/+DouglasCrockfordEsq/posts/RK8qyGVaGSr>



```
assert.equal(
  JSON.stringify({foo: 1, bar: [2, 3]}),
  '{"foo":1,"bar":[2,3]}');
```

- If space is a number, `.stringify()` returns one or more lines and indents them by space spaces per level of nesting.

```
assert.equal(
  JSON.stringify({foo: 1, bar: [2, 3]}, null, 2),
  `{
    "foo": 1,
    "bar": [
      2,
      3
    ]
  }`);
```

- If space is a string, it is used to indent.

```
assert.equal(
  JSON.stringify({foo: 1, bar: [2, 3]}, null, '>>'),
  `{
  >>"foo": 1,
  >>"bar": [
  >>>2,
  >>>3
  >>]
  }`);
```

- `.stringify(value: any, replacer?: (number | string)[] | null, space?: string | number): string` <sup>[ES5]</sup>

If replacer is an Array, then the result only includes object properties whose names are mentioned in the Array.

```
> JSON.stringify({foo: 1, bar: 2}, ['foo'])
'{"foo":1}'
```

- `.parse(text: string, reviver?: ValueVisitor): any` <sup>[ES5]</sup>

Parse the JSON inside text and return a JavaScript value. The parameter `reviver` is explained [earlier in this chapter](#).

```
assert.deepEqual(
  JSON.parse('{"a":true,"b":[1,2]}'),
  { a: true, b: [1,2] }
);
```

### 42.6.1 Sources

- TypeScript's built-in typings<sup>5</sup>

---

<sup>5</sup><https://github.com/Microsoft/TypeScript/blob/master/lib/>



## **Part X**

# **Miscellaneous topics**



## Chapter 43

# Next steps: an overview of the web development landscape (bonus)

### Contents

---

<b>43.1 Tips against feeling overwhelmed</b>	<b>437</b>
<b>43.2 Things worth learning for web development</b>	<b>438</b>
<b>43.3 Example: a tool-based JavaScript workflow</b>	<b>438</b>
<b>43.4 An overview of JavaScript tools</b>	<b>441</b>
43.4.1 Building: getting from JavaScript you write to JavaScript you deploy	441
43.4.2 Static checking	442
43.4.3 Testing	442
43.4.4 Package managers	442
43.4.5 Libraries	442
<b>43.5 Tools not related to JavaScript</b>	<b>443</b>
<b>43.6 Further reading on JavaScript</b>	<b>443</b>

---

You now know most of the JavaScript language. This chapter gives an overview of the web development landscape and describes next steps. It answers questions such as:

- What should I learn next for web development?
- What JavaScript-related tools should I know about?
- Where can I read up on advanced JavaScript features?

### 43.1 Tips against feeling overwhelmed

Web development has become a vast field: Between JavaScript, web browsers, server-side JavaScript, JavaScript libraries and JavaScript tools, there is a lot to know. Additionally, everything is always changing: some things go out of style, new things are invented, etc.

How can you avoid feeling overwhelmed when faced with this constantly changing vastness of knowledge?

- Learn the core web technology, you are working with most often, well. If you do frontend development, that may be JavaScript, CSS, SVG, or something else.
- For JavaScript, know the language, but also use one tool in each of the following categories (at least for a while). All of these categories are covered in more detail later in this chapter:
  - Compilers: compile future JavaScript or supersets of JavaScript to normal JavaScript.
  - Bundlers: combine all modules used by a web app into a single file (a script or a module). That makes loading faster and enables dead code elimination.
  - Static checkers: such as linters (that check for anti-patterns, style violations, and more) or type checkers (that type JavaScript statically and report errors).
  - Test libraries and tools
  - Version control (usually git)

Obviously, you can and should always learn something because you are curious. But I'm weary of becoming a jack of all trades and spreading myself too thin – let alone the anxiety of not knowing enough, that comes with it. Trust in your ability to learn things on demand!

## 43.2 Things worth learning for web development

These are a few things worth learning for web development:

- Browser APIs such as the *Document Object Model* (DOM), the browsers' representation of HTML in memory. These are the foundation of any kind of frontend development.
- JavaScript-adjacent technologies such as HTML and CSS.
- Frontend frameworks: When you get started with web development, it can be instructive to write user interfaces without any libraries. Once you feel more confident, frontend frameworks make many things easier, especially for larger apps. Popular frameworks include (roughly ranked by popularity):
  - React
  - Angular
  - Vue
  - Ember
- Node.js: is the most popular platform for server-side JavaScript. But it also lets you run JavaScript in the command line. Most JavaScript-related tools (even compilers!) are implemented in Node.js-based JavaScript and installed via npm. A good way to get started with Node.js is to use it for shell scripting.
- JavaScript tooling. Modern web development involves quite a lot of tools. The remainder of this chapter provides an overview of the current tooling ecosystem.

One good resource for learning web development is MDN web docs<sup>1</sup>.

## 43.3 Example: a tool-based JavaScript workflow

Fig. 43.1 depicts a classic web app – when web development was less sophisticated (for better and for worse):

<sup>1</sup><https://developer.mozilla.org/en-US/docs/Learn>

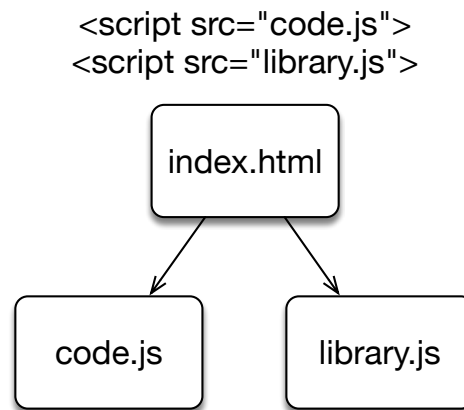


Figure 43.1: A classic, very simple web app: An HTML file refers to a JavaScript file `code.js`, which imbues the former with interactivity. `code.js` uses the library `library.js`, which must also be loaded by the HTML file.

- `index.html` contains the HTML file that is opened in web browsers.
- `code.js` contains the JavaScript code loaded and used by `index.html`.
- That code depends on the library `library.js` that is loaded via another script tag and accessed via a global variable. Notably, the HTML file needs to be aware that `code.js` depends on `library.js`.

Since then, JavaScript workflows have become more complex. Fig. 43.2 shows such a workflow – one that is based on the JavaScript bundler *webpack*.

Let's examine the pieces (data, tools, technologies) involved in this workflow:

- The app itself consists of multiple modules, written in *TypeScript* – a language that is a statically typed superset of JavaScript.
- The library used by the app is now downloaded as a *package* from a software registry, the *npm registry*. The tool for interacting with that registry, is called *npm*.
- All code files of the web app are compiled to plain JS (via a *loader*, a plugin for *webpack*).
- The tool *webpack* combines all plain JavaScript files into a single JavaScript script file. This process is called *bundling*. Bundling is done for two reasons:
  - Downloading a single file is usually faster in web browsers.
  - During bundling, you can leave out code that isn't used. Depending on how exactly that is done, this process is called *dead code elimination*, *tree-shaking* or other names.

The basic structure is still the same: The HTML file loads a JavaScript script file via a `<script>` element. However:

- The code is now modular (without the HTML file having to be aware of the modules).
- We have eliminated dead code.
- We only load a single file.
- We use a package manager for the libraries our code depends on. And the libraries aren't accessed via global variables, anymore (but via module specifiers).

In modern browsers, you can also deliver the bundle as a module (not as a script file).

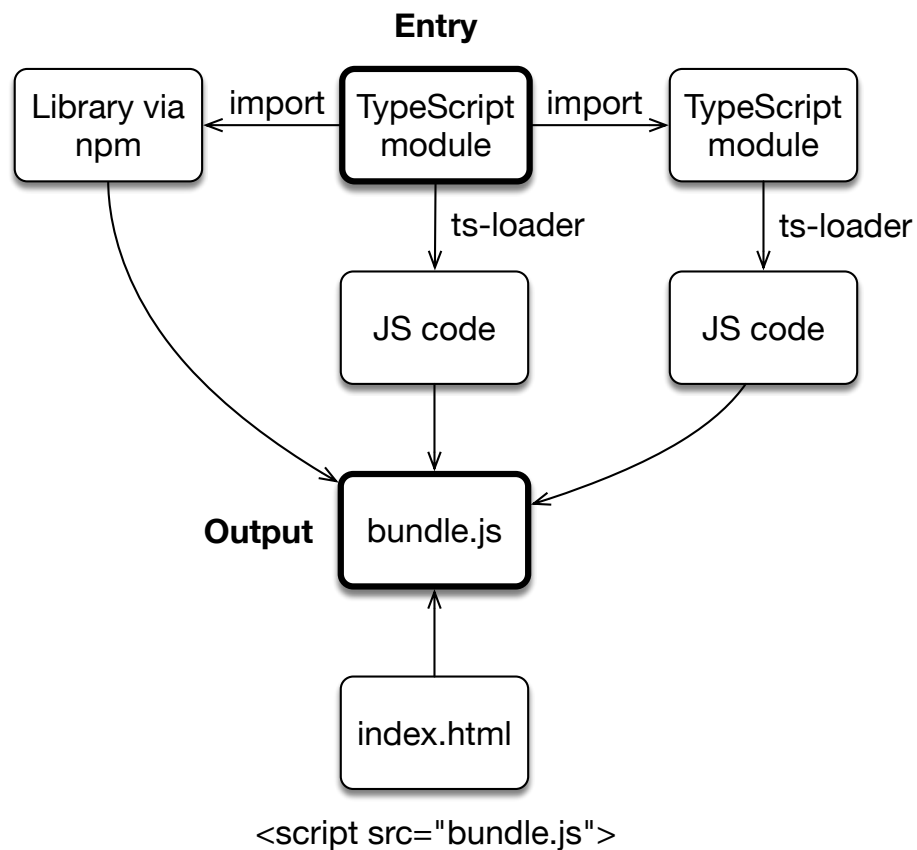


Figure 43.2: This is the workflow when developing a web app with the bundler *webpack*. Your web app consists of multiple modules. You tell webpack, in which one execution starts (the so-called *entry point*). It then analyzes the imports of the entry point, the imports of the imports, etc., to determine what code is needed to run the app. All of that code is put into a single script file.



## 43.4 An overview of JavaScript tools

Now that we have seen one workflow, let's look at various categories of tools that are popular in the world of JavaScript. You'll see categories of tools and lots of names of specific tools. The former are much more important. The names change, as tools come into and out of style, but I wanted you to see at least some of them.

### 43.4.1 Building: getting from JavaScript you write to JavaScript you deploy

*Building* JavaScript means getting from the JavaScript you write to the JavaScript you deploy. The following tools are often involved in this process:

- Compilers: compile a variety of languages to JavaScript. Two compilers that are popular in the JavaScript community, are:
  - Babel: compiles upcoming and modern JavaScript features to older versions of the language. That means you can use new features in your code and still run it on older browsers.
  - TypeScript: is a superset of JavaScript. Roughly, it is the latest version of JavaScript plus static typing.

You may occasionally see the term *transpiler*. A transpiler is a compiler that compiles source code to source code.

- Bundlers: combine all of the code of an app (proper app code plus libraries) into a single file. That is done, because single files can usually be downloaded faster – which results in less traffic and faster startups. Post-processing the code of the app also enables bundlers to only include code that is really used (via techniques such as *dead code elimination* and *tree-shaking*).

If parts of the app should be loaded on demand, multiple files are created (so-called *chunking*).

- Popular bundlers include: webpack, browserify, Rollup, Parcel.
- Minifiers: Reducing the size of the JavaScript that needs be downloaded can save money and leads to your app starting up faster. Three common techniques for doing so are: dead code elimination, compression (e.g. via gzip; the server compresses the data, the browser decompresses it) and minification. Minification makes JavaScript files smaller by renaming variables, removing comments, etc.
  - Popular tools include: UglifyJS<sup>2</sup>, babel-minify<sup>3</sup> and Closure Compiler<sup>4</sup>.

All of these build steps are usually managed via tools that are called *task runners* and *build tools* (think “make” in Unix). There are:

- Dedicated task runners: grunt, gulp, broccoli, etc.
- Tools that can be used as simple task runners: npm (via its “scripts”) and webpack (via plugins).

---

<sup>2</sup><http://lisperator.net/uglifyjs/>

<sup>3</sup><https://github.com/babel/minify>

<sup>4</sup><https://developers.google.com/closure/compiler/>

### 43.4.2 Static checking

*Static checking* means analyzing source code *statically* (without running it). That can be used to detect a variety of problems. Tools include:

- Linters: check the source code for problematic patterns, unused variables, etc. Linters are especially useful if you are still learning the language, because they point out if you are doing something wrong.
  - Popular linters include: JSLint, JSHint, ESLint, TSLint
- Code style checkers: check if code is formatted properly. They consider indentation, spaces after brackets, spaces after commas, etc.
  - Example: JSCS (JavaScript Code Style checker)
- Code formatters: automatically format your code for you, according to rules that you can customize.
  - Example: Prettier
- Type checkers: add static type checking to JavaScript.
  - Popular type checkers: TypeScript, Flow.

### 43.4.3 Testing

JavaScript has many testing frameworks. For example:

- Unit testing: Jasmine, mocha, AVA, Jest, Karma, etc.
- Integration testing: Jenkins, Travis CI, etc.
- UI testing: CasperJS, Protractor, Nightwatch.js, TestCafé, etc.

### 43.4.4 Package managers

The most popular package manager for JavaScript is npm. It started as a package manager for Node.js, but has since also become dominant for client-side web development and tools of any kind.

There are alternatives to npm, but they are all based in one way or another on npm's software registry:

- Yarn<sup>5</sup>: is a different take on npm; some of the features it pioneered are now also supported by npm.
- pnpm<sup>6</sup>: focuses on saving space when installing packages locally.

### 43.4.5 Libraries

- Various helpers: lodash (which was originally based on the Underscore.js library) is one of the most popular general helper libraries for JavaScript.
- Data structures: Immutable.js provides immutable data structures for JavaScript. Other similar libraries exist.
- Date libraries: JavaScript's built-in support for dates is awful. Two popular date libraries are: Moment.js and date-fns.

---

<sup>5</sup><https://yarnpkg.com/en/>

<sup>6</sup><https://github.com/pnpm/pnpm>

- Internationalization: For internationalization, it is good to be aware of a related standard that is supported by most modern engines: ECMA-402, the ECMAScript Internationalization API (global variable `Intl`<sup>7</sup>).
- Implementing and accessing services: The following are two popular options that are supported by a variety of libraries and tools.
  - REST (Representative State Transfer) is one popular option for services and based on HTTP(S).
  - GraphQL<sup>8</sup> is more sophisticated and supports a query language.

## 43.5 Tools not related to JavaScript

Given that JavaScript is just one of several kinds of artifacts involved in web development, more tools exist. These are but a few examples:

- CSS:
  - Minifiers: reduce the size of CSS by removing comments etc.
  - Preprocessors: let you write compact CSS – sometimes augmented with control flow constructs etc. – that is expanded into deployable, more verbose CSS.
  - Frameworks: provide help with layout, decent-looking user interface components, etc.
  - Etc.
- Images: automatic optimization, etc.

## 43.6 Further reading on JavaScript

In this book, there are occasionally tips for further reading on advanced topics. Additionally, I deliberately omitted the following advanced topic:

- ECMAScript proxies. For more info, consult “Exploring ES6”<sup>9</sup>.

Most of my books on JavaScript are free to read online at ExploringJS.com<sup>10</sup>.

---

<sup>7</sup>[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Intl](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl)

<sup>8</sup><https://graphql.org/>

<sup>9</sup>[http://exploringjs.com/es6/ch\\_proxies.html](http://exploringjs.com/es6/ch_proxies.html)

<sup>10</sup><http://exploringjs.com>



## **Part XI**

# **Appendices**



# Appendix A

## Index

(Work in progress...)

Array-like object, 267

ordinary function, 173

real function, 173

receiver, 220

specialized function, 174