

JavaScript

FOR IMPATIENT PROGRAMMERS



Dr. Axel Rauschmayer

Contents

I	Background	7
1	About this book	9
1.1	What's in this book?	9
1.2	What is not covered by this book?	9
1.3	This book isn't finished, yet	9
1.4	Can I buy a print edition?	10
1.5	Will there be a free online version?	10
1.6	Acknowledgements	10
2	Tips for reading this book	11
2.1	What are the "advanced" chapters and sections about?	11
2.2	What should I read if I'm <i>really</i> impatient?	11
2.3	How do I submit feedback and corrections?	11
2.4	I'm occasionally seeing type annotations – how do those work?	11
3	History and evolution of JavaScript	13
3.1	How JavaScript was created	13
3.2	Standardization	13
3.3	Evolving JavaScript: don't break the web	16
4	FAQ: JavaScript	17
4.1	Why does JavaScript fail silently so often?	17
II	First steps	19
5	The big picture	21
5.1	What are you learning in this book?	21
5.2	The structure of browsers and Node.js	21
5.3	Trying out JavaScript code	22
5.4	Further reading	25
6	Syntax	27
6.1	An overview of JavaScript's syntax	27
6.2	(Advanced)	31
6.3	Identifiers	31
6.4	Statement vs. expression	32
6.5	Syntactically ambiguous constructs	33

6.6	Semicolons	35
6.7	Automatic semicolon insertion (ASI)	36
6.8	Semicolons: best practices	38
6.9	Strict mode	39
7	Assertion API	41
7.1	Assertions in software development	41
7.2	How assertions are used in this book	41
7.3	Normal comparison versus deep comparison	42
7.4	Quick reference: module assert	42
8	Getting started with quizzes and exercises	45
8.1	Quizzes	45
8.2	Exercises	45
8.3	Unit tests in JavaScript	46
III	Variables and values	49
9	Variables and assignment	51
9.1	let	51
9.2	const	51
9.3	Deciding between let and const	52
9.4	Variables are block-scoped	53
10	Values	55
10.1	What's a type?	55
10.2	JavaScript's type hierarchy	55
10.3	The types of the language specification	55
10.4	Primitive values versus objects	56
10.5	Classes and constructor functions	58
10.6	Constructor functions associated with primitive types	59
10.7	The operators typeof and instanceof: what's the type of a value?	59
10.8	Converting between types	61
11	Operators	63
11.1	Two important rules for operators	63
11.2	The plus operator (+)	64
11.3	Assignment operators	64
11.4	Equality: == versus ===	65
11.5	Ordering operators	67
11.6	Various other operators	68
IV	Primitive values	69
12	The non-values undefined and null	71
12.1	undefined vs. null	71
12.2	Occurrences of undefined and null	72
12.3	Checking for undefined or null	72
12.4	undefined and null don't have properties	73

13 Booleans	75
13.1 Converting to boolean	75
13.2 Falsy and truthy values	76
13.3 Conditional operator (?:)	78
13.4 Binary logical operators: And (&&), Or ()	79
13.5 Logical Not (!)	81
14 Numbers	83
14.1 JavaScript only has floating point numbers	83
14.2 Number literals	83
14.3 Number operators	84
14.4 Converting to number	86
14.5 Error values	87
14.6 Error value: NaN	87
14.7 Error value: Infinity	88
14.8 The precision of numbers: careful with decimal fractions	89
14.9 (Advanced)	89
14.10 Background: floating point precision	89
14.11 Integers in JavaScript	90
14.12 Bitwise operators	92
14.13 Quick reference: numbers	94
15 Math	101
15.1 Data properties	101
15.2 Exponents, roots, logarithms	102
15.3 Rounding	103
15.4 Trigonometric Functions	104
15.5 asm.js helpers	106
15.6 Various other functions	106
15.7 Sources	107
16 Strings	109
16.1 Plain string literals	109
16.2 Accessing characters and code points	110
16.3 String concatenation via +	110
16.4 Converting to string	111
16.5 Comparing strings	113
16.6 JavaScript characters vs. Unicode code points	113
16.7 Quick reference: Strings	115
17 Using template literals and tagged templates	123
17.1 Disambiguation: "template"	123
17.2 Template literals	124
17.3 Tagged templates	124
17.4 Raw string literals	126
17.5 (Advanced)	126
17.6 Multi-line template literals and indentation	127
17.7 Simple templating via template literals	128
17.8 Further reading	130

18 Symbols	131
18.1 Use cases for symbols	131
18.2 Publicly known symbols	133
18.3 Converting symbols	134
18.4 Further reading	135
V Control flow and data flow	137
19 Control flow statements	139
19.1 Controlling loops: break and continue	139
19.2 if statements	141
19.3 switch statements	142
19.4 while loops	145
19.5 do-while loops	146
19.6 for loops	146
19.7 for-of loops	147
19.8 for-await-of loops	148
19.9 for-in loops (avoid)	148
20 Callable values	151
20.1 Kinds of functions	151
20.2 Named function expressions	154
20.3 Arrow functions	155
20.4 Hoisting	157
20.5 Returning values from functions	158
20.6 Parameter handling	158
20.7 Understanding JavaScript's callable values (advanced)	162
21 Where are the remaining chapters?	165

Part I

Background

Chapter 1

About this book

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)¹, which are free to read 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<http://exploringjs.com/>

1.4 Can I buy a print edition?

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

1.6 Acknowledgements

- Cover by Fran Caye².
- Thanks for reviewing:
 - Johannes Weber (@jowe³)

Generated: 2018-11-06 09:01

²<http://francaye.net>

³<https://twitter.com/jowe>

Chapter 2

Tips for reading this book

This FAQ 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 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 chapter](#) at the end of this book.

Chapter 3

History and evolution of JavaScript

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).

3.2.1 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)

3.2.2 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.

3.2.3 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 (“strawman”), 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.

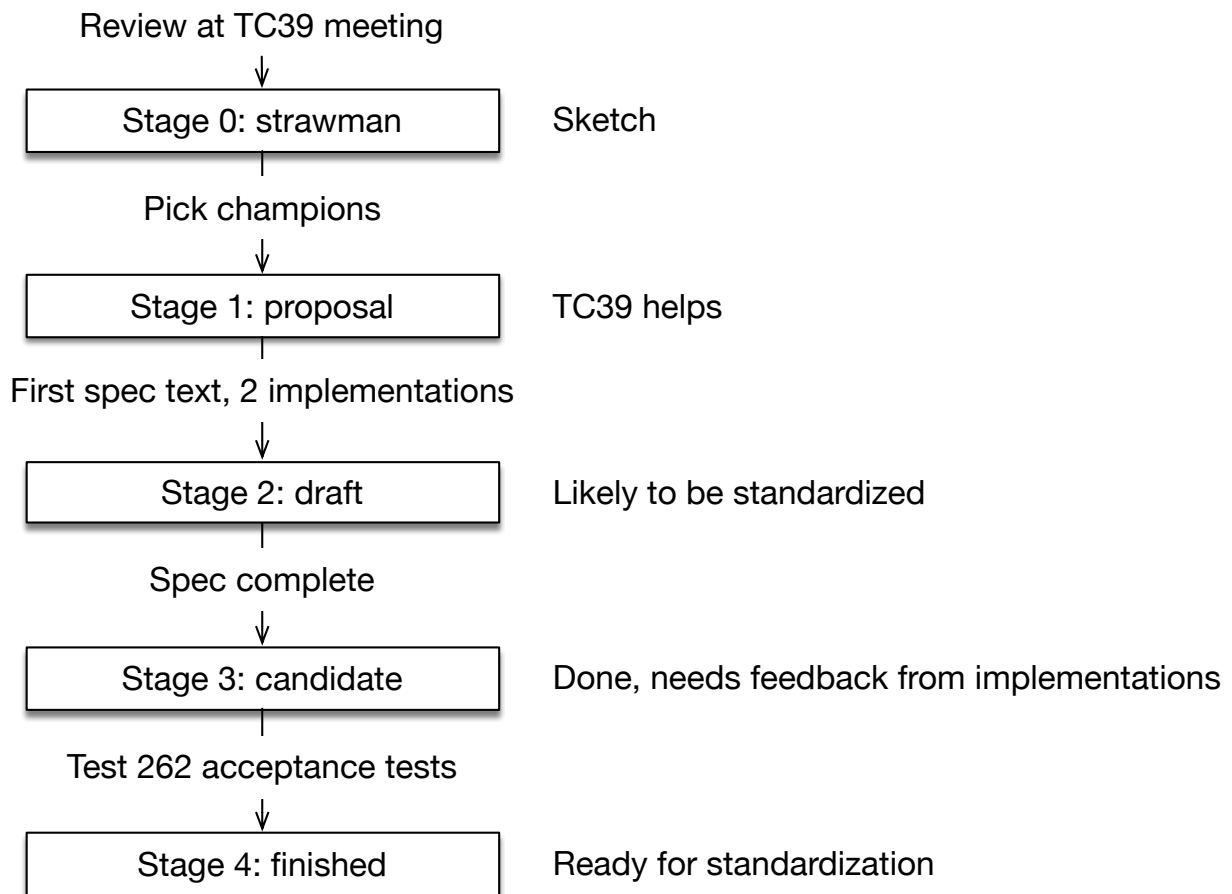


Figure 3.1: Each ECMAScript feature proposal goes through stages that are numbered from 0 to 4. *Champions* are TC39 members that support the authors of a feature. Test 262 is a suite of tests that checks JavaScript engines for compliance with the language specification.

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 ES2019¹”.

¹http://exploringjs.com/es2018-es2019/ch_tc39-process.html

3.2.4 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.

3.3 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.

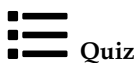
Let’s assume we create a new version of JavaScript that is not backward compatible and fixes all of its flaws. As a result, we’d encounter the following problems:

- 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 ES6²”.



Quiz

See quiz app.

²http://exploringjs.com/es6/ch_one-javascript.html

Chapter 4

FAQ: JavaScript

4.1 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.

```
> '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.

Part II

First steps

Chapter 5

The big picture

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?

5.1 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.

5.2 The structure of browsers and Node.js

The structures of the two JavaScript platforms *web browser* and *Node.js* are similar (fig. 5.1):

- 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.

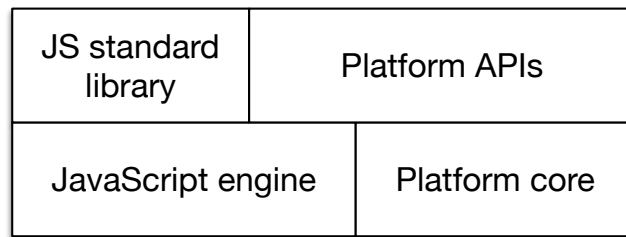


Figure 5.1: The structure of the two JavaScript platforms *web browser* and *Node.js*.

5.2.1 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`).

5.3 Trying out JavaScript code

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

5.3.1 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. 5.2 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 Safari¹
- Google Chrome²
- Microsoft Edge³
- Mozilla Firefox⁴

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

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

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

⁴https://developer.mozilla.org/en-US/docs/Tools/Web_Console/Opening_the_Web_Console

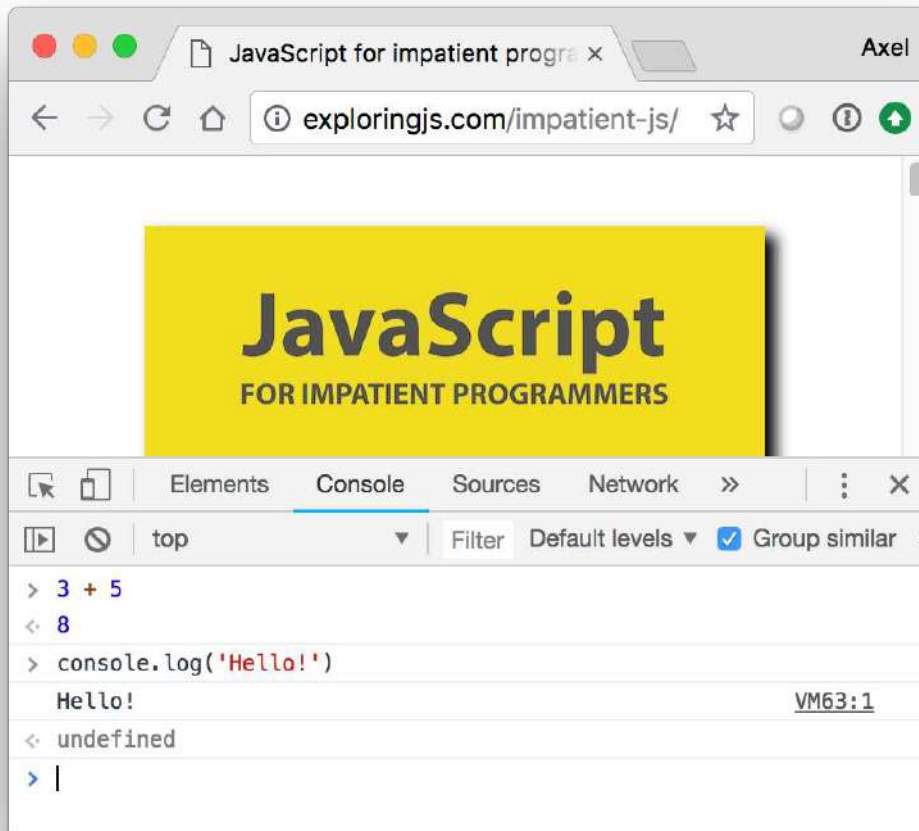
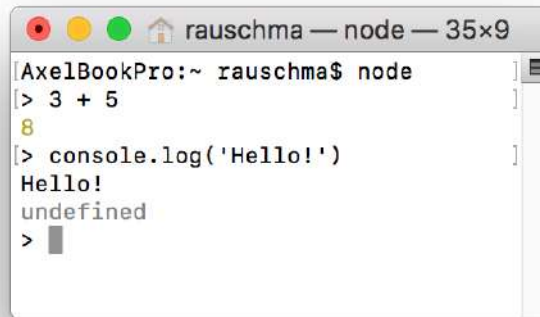


Figure 5.2: The console of the web browser “Google Chrome” is open while visiting a web page.

5.3.2 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. 5.3: The text after `>` is input from the user; everything else is output from Node.js.

A screenshot of a terminal window titled "rauschma — node — 35x9". The prompt is "AxelBookPro:~ rauschma\$ node". The user enters "> 3 + 5" and the output is "8". The user enters "> console.log('Hello!')" and the output is "Hello!". The prompt "undefined" is shown, and the user enters ">".

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

Figure 5.3: 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
```

5.3.3 Other options

Other options include:

- There are many web apps that let you experiment with JavaScript in web browsers. For example, Babel's REPL⁵.
- There are also native apps and IDE plugins for running JavaScript.

⁵<https://babeljs.io/repl>

5.4 Further reading

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

Chapter 6

Syntax

6.1 An overview of JavaScript's syntax

6.1.1 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';
```

6.1.2 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';
```

6.1.3 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
```

6.1.4 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

6.1.5 Where to put semicolons?

At the end of a statement:

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

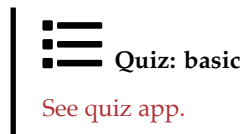
But not if that statement ends with a curly brace:

```
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() {
  // ...
};
```



6.2 (Advanced)

All remaining sections of this chapter are advanced.

6.3 Identifiers

6.3.1 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;
```

6.3.2 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.).

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

6.4.1 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
```


6.5 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.

6.5.1 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?

6.5.2 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!).

6.5.3 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.

6.5.4 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.

An IIFE (pronounced "iffy") 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.

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.

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.

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

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');
```

6.5.5 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');
```

6.5.6 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);
```

6.5.7 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);
```

6.6 Semicolons

6.6.1 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.

6.6.2 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) {}
```

6.7 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.

6.7.1 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.

6.7.2 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.

6.8 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 Prettier¹ 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 ESLint² has a rule³ that warns about critical cases if you either require or forbid semicolons.

¹<https://prettier.io>

²<https://eslint.org>

³<https://eslint.org/docs/rules/semi>

6.9 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.

6.9.1 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';  
}
```


6.9.2 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 7

Assertion API

7.1 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 version¹ of `assert`.

7.2 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.

7.2.1 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:

¹https://nodejs.org/api/assert.html#assert_strict_mode

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

7.2.2 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](#)

7.3 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']);
```

7.4 Quick reference: module `assert`

For the full documentation, see the Node.js docs².

7.4.1 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`

²<https://nodejs.org/api/assert.html>

```
assert.notEqual(3+3, 22);
```

7.4.2 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]);
```

7.4.3 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

```
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`,  
  }  
);
```

7.4.4 Other tool functions

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

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



Quiz

See quiz app.

Chapter 8

Getting started with quizzes and exercises

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.

8.1 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.

8.2 Exercises

8.2.1 Installing the exercises

To install the exercises:

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

8.2.2 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.

8.3 Unit tests in JavaScript

All exercises in this book are tests that are run via the test framework Mocha¹. This section gives a brief introduction.

8.3.1 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)

8.3.1.1 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.

8.3.1.2 Part 2: the tests

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

```
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](#).

¹<https://mochajs.org>

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`

8.3.2 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.

8.3.2.1 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();
    }
  });
});
```

8.3.2.2 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);
  });
});
```

8.3.2.3 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 9

Variables and assignment

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 JavaScript¹".

9.1 `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;
```

9.2 `const`

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

```
const i = 0; // must initialize  
  
assert.throws(  
  () => { i = i + 1 },  
  {  
    name: 'TypeError',  
  },
```

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

```

    message: 'Assignment to constant variable.',
  }
);

```

9.2.1 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.',
  }
);

```

9.2.2 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);
}

```

9.3 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``

9.4 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.

9.4.1 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.



Quiz

See [quiz app](#).

Chapter 10

Values

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](#).

10.1 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` }.

10.2 JavaScript's type hierarchy

Fig. 10.1 shows JavaScript's type hierarchy. What do we learn from that diagram?

- 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.

10.3 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')`).

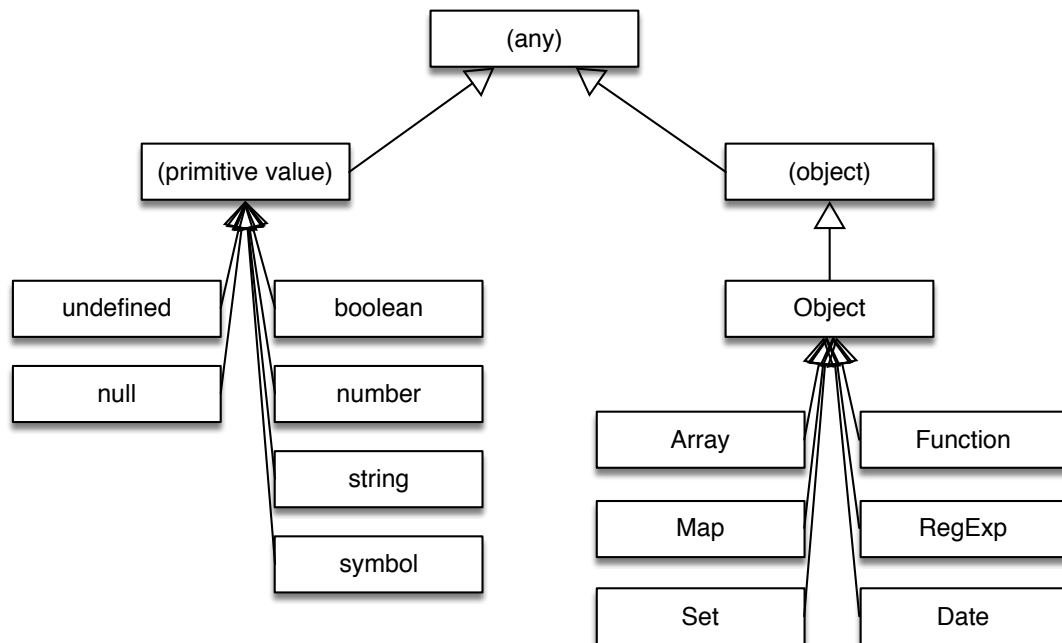


Figure 10.1: A partial hierarchy of JavaScript’s types. Missing are the classes for errors, the constructor functions associated with primitive types, and more. The diagram hints at the fact that not all objects are instances of `Object`.

- `object`: the type of all objects (different from `Object`, the type of all instances of class `Object` and its subclasses).

10.4 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.

10.4.1 Primitive values

10.4.1.1 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'",
  }
);
```

10.4.1.2 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.

10.4.2 Objects

Two common literals for creating objects, are:

- Object literals:

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

- Array literals:

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

10.4.2.1 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');
```

10.4.2.2 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).

10.5 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.

10.6 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);
```

10.7 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.

10.7.1 `typeof`

Table 10.1: The results of the `typeof` operator.

x	typeof x
undefined	'undefined'
null	'object'
Boolean	'boolean'

x	typeof x
Number	'number'
String	'string'
Symbol	'symbol'
Function	'function'
All other objects	'object'

Tbl. 10.1 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`

10.7.2 `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``

10.8 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.

10.8.1 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
```

10.8.2 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 11

Operators

11.1 Two important rules for operators

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

11.1.1 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);
```

11.1.2 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'
```

11.2 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.)

11.3 Assignment operators

11.3.1 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.

11.3.2 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>');
```

11.3.3 All compound assignment operators

- Arithmetic operators:

```
+= -= *= /= %= **=
```

`+=` also works for string concatenation

- Bitwise operators:

```
<<= >>= >>>= &= ^= |=
```

11.4 Equality: == versus ===

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

11.4.1 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
```

11.4.2 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
```

11.4.3 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.

11.4.3.1 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.

11.4.3.2 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).

11.5 Ordering operators

Table 11.1: JavaScript's ordering operators.

Operator	name
<	less than

Operator	name
<code><=</code>	Less than or equal
<code>></code>	Greater than
<code>>=</code>	Greater than or equal

JavaScript's ordering operators (tbl. 11.1) 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](#).

11.6 Various other operators

- Comma operator¹: `a, b`
- void operator²: `void 0`
- Operators for booleans, strings, numbers, objects: are covered elsewhere in this book.

■ ■ ■ Quiz

[See quiz app.](#)

¹http://speakingjs.com/es5/ch09.html#comma_operator

²http://speakingjs.com/es5/ch09.html#void_operator

Part IV

Primitive values

Chapter 12

The non-values `undefined` and `null`

12.1 `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.

12.1.1 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.

12.2 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.

12.2.1 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);
```

12.2.2 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}'
```

12.3 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](#)).

12.4 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;
}

```

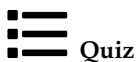
If we apply getFoo() to various value, we can see that it only fails for undefined and null:

```

> 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 13

Booleans

The primitive type *boolean* comprises two values:

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

13.1 Converting to boolean

Table 13.1: Converting values to booleans.

x	Boolean(x)
undefined	false
null	false
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. 13.1 describes how various values are converted to boolean.

13.1.1 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.

13.2 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. 13.1](#), 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
```

13.2.1 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.

13.2.2 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](#).

13.2.3 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');
}
```

13.2.4 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`

13.3 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'
```

13.4 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'
```

13.4.1 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'
''
```

13.4.2 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'
```

13.4.3 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``

13.5 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
```



Quiz

See [quiz app](#).

Chapter 14

Numbers

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

14.1 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.

14.2 Number literals

Let's examine literals for numbers.

14.2.1 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);
```

14.2.2 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
```

14.2.3 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
```

14.3 Number operators

14.3.1 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
```

14.3.2 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
```

14.3.3 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);
```

14.3.3.1 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`

14.4 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`

14.5 Error values

Two number values are returned when errors happen:

- NaN
- Infinity

14.6 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
```

14.6.1 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));
```

14.6.2 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
```

14.7 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
```

14.7.1 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);
```


14.7.2 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``

14.8 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.](#)

14.9 (Advanced)

All remaining sections of this chapter are advanced.

14.10 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:

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

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^{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^{exponent} .
 - Can be represented: 1/2
 - Can be represented: 1/4
 - Cannot be represented: 1/10 (=1/(2×5))
 - * Why? We can't get a five into the denominator.

14.11 Integers in JavaScript

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

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

14.11.1 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. 14.1 shows the results of these functions for various inputs.

Table 14.1: 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

14.11.2 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 (`>>>`): unsigned, $[0, 2^{32})$
 - Range of all other bitwise operators: signed, $[-2^{31}, 2^{31})$

14.11.3 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);
}
```

14.11.3.1 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`
```

14.12 Bitwise operators

14.12.1 Binary bitwise operators

Table 14.2: Binary bitwise operators.

Operation	Name
<code>num1 & num2</code>	Bitwise And
<code>num1 num2</code>	Bitwise Or

Operation	Name
num1 ^ num2	Bitwise Xor

The binary operators (tbl. 14.2) combine the bits of their operands to produce their results:

```
> (0b1010 & 0b11).toString(2)
'10'
> (0b1010 | 0b11).toString(2)
'1011'
> (0b1010 ^ 0b11).toString(2)
'1001'
```

14.12.2 Bitwise Not

Table 14.3: The bitwise Not operator.

Operation	Name
~num	Bitwise Not, ones' complement

The bitwise Not operator (tbl. 14.3) inverts each binary digit of its operand:

```
> bin(~0b100)
'111111111111111111111111111111011'
```

`bin()` returns a binary representation of its argument and is implemented as follows.

```
function bin(n) {
  // >>> ensures highest bit isn't interpreted as a sign
  return (n >>> 0).toString(2).padStart(32, '0');
}
```

14.12.3 Bitwise shift operators

Table 14.4: Bitwise shift operators.

Operation	Name
num << count	Left shift
num >> count	Signed right shift
num >>> count	Unsigned right shift

The shift operators (tbl. 14.4) move binary digits to the left or to the right:

```
> (0b10 << 1).toString(2)
'100'
```

```
>> preserves highest bit, >>> doesn't:
> bin(0b1000000000000000000000000000000010 >> 1)
'110000000000000000000000000000001'
> bin(0b1000000000000000000000000000000010 >>> 1)
'010000000000000000000000000000001'
```

14.13 Quick reference: numbers

14.13.1 Converting to number

Tbl. 14.5 shows what happens if you convert various values to numbers via `Number()`.

Table 14.5: Converting values to numbers.

x	Number(x)
undefined	NaN
null	0
boolean	false → 0, true → 1
number	x (no change)
string	' ' → 0 other → parse number, ignoring whitespace
object	configurable (<code>.valueOf()</code> , <code>.toString()</code> , <code>Symbol.toPrimitive</code>)

14.13.2 Arithmetic operators

JavaScript has the following arithmetic operators:

- Binary arithmetic operators (tbl. 14.6)
- Prefix and suffix arithmetic operators (tbl. 14.7)

Table 14.6: Binary arithmetic operators.

Operator	Name		Example
<code>_ + _</code>	Addition	ES1	<code>3 + 4 → 7</code>
<code>_ - _</code>	Subtraction	ES1	<code>9 - 1 → 8</code>
<code>_ * _</code>	Multiplication	ES1	<code>3 * 2.25 → 6.75</code>
<code>_ / _</code>	Division	ES1	<code>5.625 / 5 → 1.125</code>
<code>_ % _</code>	Remainder	ES1	<code>8 % 5 → 3</code> <code>-8 % 5 → -3</code>
<code>_ ** _</code>	Exponentiation	ES2016	<code>6 ** 2 → 36</code>

14.13.4 `Number.*` data properties

- `Number.EPSILON`: number ^[ES6]
The difference between 1 and the next representable floating point number. In general, a machine epsilon¹ provides an upper bound for rounding errors in floating point arithmetic.
 - Approximately: $2.2204460492503130808472633361816 \times 10^{-16}$
- `Number.MAX_SAFE_INTEGER`: number ^[ES6]
The largest integer that JavaScript can represent uniquely. The same as $(2^{53} - 1)$.
- `Number.MAX_VALUE`: number ^[ES1]
The largest positive finite JavaScript number.
 - Approximately: $1.7976931348623157 \times 10^{308}$
- `Number.MIN_SAFE_INTEGER`: number ^[ES6]
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 ^[ES1]
The smallest positive JavaScript number. Approximately 5×10^{-324} .
- `Number.NaN`: number ^[ES1]
The same as the global variable `NaN`.
- `Number.NEGATIVE_INFINITY`: number ^[ES1]
The same as `-Number.POSITIVE_INFINITY`.
- `Number.POSITIVE_INFINITY`: number ^[ES1]
The same as the global variable `Infinity`.

14.13.5 `Number.*` methods

- `Number.isFinite(num: number): boolean` ^[ES6]
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):

¹https://en.wikipedia.org/wiki/Machine_epsilon


```
> isFinite('123')
true
> Number.isFinite('123')
false
```

- `Number.isInteger(num: number): boolean` ^[ES6]

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` ^[ES6]

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` ^[ES6]

Returns true if num is a number and uniquely represents an integer.

- `Number.parseFloat(str: string): number` ^[ES6]

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` ^[ES6]

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
```

14.13.6 `Number.prototype.*`

- `toExponential(fractionDigits?: number): string` ^[ES3]

Returns a string that represents the receiver 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` ^[ES3]

Returns an exponent-free representation of the receiver, rounded to `fractionDigits` digits.

```
> 0.0000003.toString()
'3e-7'

> 0.0000003.toFixed(10)
'0.0000003000'
> 0.0000003.toFixed()
'0'
```

If the receiver is 10^{21} or greater, even `.toFixed()` uses an exponent:

```
> (10 ** 21).toFixed()
'1e+21'
```

- `toPrecision(precision?: number): string` ^[ES3]

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` ^[ES1]

Returns a string representation of the receiver.

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
```

14.13.7 Sources

- Wikipedia
- TypeScript's built-in typings²
- MDN web docs for JavaScript³

²<https://github.com/Microsoft/TypeScript/blob/master/lib/>

³<https://developer.mozilla.org/en-US/docs/Web/JavaScript>

- ECMAScript language specification⁴



Quiz: advanced

See quiz app.

⁴<https://tc39.github.io/ecma262/>

Chapter 15

Math

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.

15.1 Data properties

- **Math.E:** number ^[ES1]
Euler's number, base of the natural logarithms, approximately 2.7182818284590452354.
- **Math.LN10:** number ^[ES1]
The natural logarithm of 10, approximately 2.302585092994046.
- **Math.LN2:** number ^[ES1]
The natural logarithm of 2, approximately 0.6931471805599453.
- **Math.LOG10E:** number ^[ES1]
The logarithm of e to base 10, approximately 0.4342944819032518.
- **Math.LOG2E:** number ^[ES1]
The logarithm of e to base 2, approximately 1.4426950408889634.
- **Math.PI:** number ^[ES1]
The mathematical constant π , ratio of a circle's circumference to its diameter, approximately 3.1415926535897932.
- **Math.SQRT1_2:** number ^[ES1]
The square root of $1/2$, approximately 0.7071067811865476.
- **Math.SQRT2:** number ^[ES1]
The square root of 2, approximately 1.4142135623730951.

15.2 Exponents, roots, logarithms

- `Math.cbrt(x: number): number` ^[ES6]

Returns the cube root of x .

```
> Math.cbrt(8)
2
```

- `Math.exp(x: number): number` ^[ES1]

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` ^[ES6]

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` ^[ES1]

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` ^[ES6]

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` ^[ES6]

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` ^[ES6]

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` ^[ES1]

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` ^[ES1]

Returns the square root of x . The inverse of `x ** 2`.

```
> Math.sqrt(9)
3
```

15.3 Rounding

Rounding means converting an arbitrary number to an integer (a number without a decimal fraction). Tbl. 15.1 lists the available functions and what they return for a few representative inputs.

Table 15.1: 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` ^[ES1]

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` ^[ES1]

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` ^[ES1]

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` ^[ES6]

Removes the decimal fraction of x and returns the resulting integer.

```
> Math.trunc(1.9)
1
> Math.trunc(2.1)
2
```

15.4 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` ^[ES1]

Returns the arc cosine (inverse cosine) of x .

```
> Math.acos(0)
1.5707963267948966
> Math.acos(1)
0
```

- `Math.acosh(x: number): number` ^[ES6]

Returns the inverse hyperbolic cosine of x .

- `Math.asin(x: number): number` ^[ES1]

Returns the arc sine (inverse sine) of x .

```
> Math.asin(0)
0
> Math.asin(1)
1.5707963267948966
```

- `Math.asinh(x: number): number` ^[ES6]

Returns the inverse hyperbolic sine of x .

- `Math.atan(x: number): number` ^[ES1]
Returns the arc tangent (inverse tangent) of x .
- `Math.atanh(x: number): number` ^[ES6]
Returns the inverse hyperbolic tangent of x .
- `Math.atan2(y: number, x: number): number` ^[ES1]
Returns the arc tangent of the quotient y/x .
- `Math.cos(x: number): number` ^[ES1]
Returns the cosine of x .

```
> Math.cos(0)
1
> Math.cos(Math.PI)
-1
```
- `Math.cosh(x: number): number` ^[ES6]
Returns the hyperbolic cosine of x .
- `Math.hypot(...values: number[]): number` ^[ES6]
Returns the square root of the sum of the squares of `values` (Pythagoras' theorem):

```
> Math.hypot(3, 4)
5
```
- `Math.sin(x: number): number` ^[ES1]
Returns the sine of x .

```
> Math.sin(0)
0
> Math.sin(Math.PI / 2)
1
```
- `Math.sinh(x: number): number` ^[ES6]
Returns the hyperbolic sine of x .
- `Math.tan(x: number): number` ^[ES1]
Returns the tangent of x .

```
> Math.tan(0)
0
> Math.tan(1)
1.5574077246549023
```
- `Math.tanh(x: number): number`; ^[ES6]
Returns the hyperbolic tangent of x .

15.5 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` ^[ES6]
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` ^[ES6]
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.

15.6 Various other functions

- `Math.abs(x: number): number` ^[ES1]
Returns the absolute value of `x`.

```
> Math.abs(3)
3
> Math.abs(-3)
3
> Math.abs(0)
0
```
- `Math.clz32(x: number): number` ^[ES6]
Counts the leading zero bits in the 32-bit integer `x`. Used in DSP algorithms.

```
> Math.clz32(0b010000000000000000000000000000)
1
> Math.clz32(0b001000000000000000000000000000)
2
> Math.clz32(2)
30
> Math.clz32(1)
31
```
- `Math.max(...values: number[]): number` ^[ES1]
Converts `values` to numbers and returns the largest one.

```
> Math.max(3, -5, 24)
24
```

- `Math.min(...values: number[]): number` ^[ES1]
Converts values to numbers and returns the smallest one.

```
> Math.min(3, -5, 24)
-5
```
- `Math.random(): number` ^[ES1]
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` ^[ES6]
Returns the sign of a number:

```
> Math.sign(-8)
-1
> Math.sign(0)
0
> Math.sign(3)
1
```

15.7 Sources

- Wikipedia
- TypeScript's built-in typings¹
- MDN web docs for JavaScript²
- ECMAScript language specification³

¹<https://github.com/Microsoft/TypeScript/blob/master/lib/>

²<https://developer.mozilla.org/en-US/docs/Web/JavaScript>

³<https://tc39.github.io/ecma262/>

Chapter 16

Strings

Strings are primitive values in JavaScript and immutable. That is, string-related operations always produce new strings and never change existing strings.

16.1 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)

16.1.1 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!\""
```

16.2 Accessing characters and code points

16.2.1 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);
```

16.2.2 Accessing Unicode code points via for-of and spreading

Iterating over strings via for-of or spreading (...) visits Unicode *code points* (whose range is 21 bits). Each code point is represented by 1–2 JavaScript characters (whose range is 16 bits). For more information, see [the section on code points](#) 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']);
```

16.3 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``

16.4 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
```

16.4.1 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}'
```

16.4.2 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');
```

16.4.3 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"
}
```


16.5 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 API¹ (Intl).

16.6 JavaScript characters vs. Unicode code points

Unicode's atomic unit of text is called *code point*. In many ways, code points are Unicode characters, but you occasionally need multiple code points to represent a single text symbol (a so-called *grapheme cluster*; details soon).

The range of Unicode code points is 21 bits. To represent them in JavaScript strings, one or two JavaScript characters (whose range is 16 bits) 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);
```

16.6.1 Working with Unicode 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{1F600}'
'☺'
```

Converting from code points:

```
> String.fromCodePoint(0x1F600)
'☺'
```

¹https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Intl

Converting to code points:

```
> '👍'.codePointAt(0).toString(16)
'1f600'
```

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
```

16.6.2 Caveat: grapheme clusters

A *grapheme cluster* is what 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
```

For more information, consult “Let’s Stop Ascribing Meaning to Code Points²” by Manish Goregaokar.

²<https://manishearth.github.io/blog/2017/01/14/stop-ascribing-meaning-to-unicode-code-points/>

16.7 Quick reference: Strings

Strings are immutable, none of the string methods ever modify the receiver.

16.7.1 Converting to string

Tbl. 16.1 describes how various values are converted to strings.

Table 16.1: 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()

16.7.2 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)

16.7.3 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');
```

16.7.4 String.prototype.*: finding and matching

- endsWith(searchString: string, endPos=this.length): boolean ^[ES6]

Returns true if the receiver 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` ^[ES6]

Returns `true` if the receiver 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` ^[ES1]

Returns the lowest index at which `searchString` appears within the receiver, 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` ^[ES1]

Returns the highest index at which `searchString` appears within the receiver, 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` ^[ES3]

If `regExp` is a regular expression with flag `/g` not set, then `.match()` returns the first match for `regExp` within the receiver. 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 groups³ (ES2018) become prop-

³http://exploringjs.com/es2018-es2019/ch_regexp-named-capture-groups.html

erties 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
```

- `match(regExp: RegExp): string[] | null` ^[ES3]

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` ^[ES3]

Returns the index at which `regExp` occurs within the receiver. 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` ^[ES6]

Returns `true` if `searchString` occurs in the receiver at index `startPos`. Returns `false`, otherwise.

```
> '.gitignore'.startsWith('.')
true
> 'abcde'.startsWith('bc', 1)
true
```

16.7.5 String.prototype.*: extracting

- `slice(start=0, end=this.length): string` ^[ES3]

Returns the substring of the receiver 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[]` ^[ES3]

Splits the receiver 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` ^[ES1]

Use `.slice()` instead of this method. `.substring()` wasn't implemented consistently in older engines and doesn't support negative indices.

16.7.6 String.prototype.*: combining

- `concat(...strings: string[]): string` ^[ES3]

Returns the concatenation of the receiver and strings. `'a'+ 'b'` is equivalent to `'a'.concat('b')` and more concise.

```
> 'ab'.concat('cd', 'ef', 'gh')
'abcdefgh'
```

- `padEnd(len: number, fillString=' '): string` ^[ES2017]

Appends `fillString` to the receiver until it has the desired length `len`.

```
> '#'.padEnd(2)
'# '
> 'abc'.padEnd(2)
'abc'
> '#'.padEnd(5, 'abc')
'#abca'
```

- `padStart(len: number, fillString=' '): string` ^[ES2017]

Prepends `fillString` to the receiver until it has the desired length `len`.

```
> '#'.padStart(2)
'# '
> 'abc'.padStart(2)
'abc'
> '#'.padStart(5, 'abc')
'abca#'
```

- `repeat(count=0): string` ^[ES6]

Returns a string that is the receiver, repeated count times.

```
> '*'.repeat()
''
> '*'.repeat(3)
'***'
```

16.7.7 String.prototype.*: transforming

- `normalize(form: 'NFC'|'NFD'|'NFKC'|'NFKD' = 'NFC')`: string^[ES6]

Normalizes the receiver according to the Unicode Normalization Forms⁴.

- `replace(searchValue: string | RegExp, replaceValue: string)`: string^[ES3]

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 2018-04 b'.replace(/([0-9]{4})-([0-9]{2})/, '|$2|')
'a |04| b'
> 'a 2018-04 b'.replace(/([0-9]{4})-([0-9]{2})/, '|$&|')
'a |2018-04| b'
> 'a 2018-04 b'.replace(/([0-9]{4})-([0-9]{2})/, '|$`|')
'a |a | b'
```

Named capture groups⁵ (ES2018) are supported, too:

- `$(name)` becomes the capture of named group `name`

Example:

```
> 'a 2018-04 b'.replace(/(?<year>[0-9]{4})-(?<month>[0-9]{2})/, '|$(month)|')
'a |04| b'
```

⁴<https://unicode.org/reports/tr15/>

⁵http://exploringjs.com/es2018-es2019/ch_regexp-named-capture-groups.html

- `replace(searchValue: string | RegExp, replacer: (...args: any[]) => string): string` ^[ES3]

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
- (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 2018-04 b'.replace(regexp, replacer),
  'a |2018-04| b');
```

Named capture groups⁶ (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 2018-04 b'.replace(regexp, replacer),
  'a |04| b');
```

- `toUpperCase(): string` ^[ES1]

Returns a copy of the receiver 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` ^[ES1]

Returns a copy of the receiver 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` ^[ES5]

Returns a copy of the receiver in which all leading and trailing whitespace is gone.

⁶http://exploringjs.com/es2018-es2019/ch_regexp-named-capture-groups.html


```
> '\r\n# \t'.trim()
'#'
```

16.7.8 String.prototype.*: chars, char codes, code points

- `charAt(pos: number): string` ^[ES1]

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` ^[ES1]

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` ^[ES6]

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`.

16.7.9 Sources

- TypeScript’s built-in typings⁷
- MDN web docs for JavaScript⁸
- ECMAScript language specification⁹



Exercise: Using string methods

```
`exercises/strings/remove_extension_test.js`
```



Quiz

See quiz app.

⁷<https://github.com/Microsoft/TypeScript/blob/master/lib/>

⁸<https://developer.mozilla.org/en-US/docs/Web/JavaScript>

⁹<https://tc39.github.io/ecma262/>

Chapter 17

Using template literals and tagged templates

Before we dig into the two features *template literal* and *tagged template*, let's first examine the multiple meanings of the term *template*.

17.1 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 Handlebars¹:

```
<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(
```

¹<https://handlebarsjs.com>

```
getArgs`Count: ${5}!`,
[['Count: ', '!'], 5] );
```

Note that `getArgs()` receives both the text of the literal and the data interpolated via `${}`.

17.2 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.

17.3 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.

17.3.1 Tag function library: lit-html

`lit-html`² is a templating library that is based on tagged templates and used by the frontend framework `Polymer`³:

```
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>
`;
```

`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.

17.3.2 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})`;
```

²<https://github.com/Polymer/lit-html>

³<https://www.polymer-project.org/>

```

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');

```

17.3.3 Tag function library: graphql-tag

The library `graphql-tag`⁴ 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.

17.4 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:

```

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');

```

17.5 (Advanced)

All remaining sections are advanced

⁴<https://github.com/apollographql/graphql-tag>

17.6 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:');
// Replace spaces with mid-dots:
console.log(div('Hello!').replace(/ /g, '.'));
```

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.

17.6.1 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 Brand⁵:

```
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:

⁵<https://github.com/dmnd/dedent>

Output:

```
<div>
  Hello!
</div>
```

17.6.2 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>
```

17.7 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 tpl = (data) => `Hello ${data.name}!`;
assert.equal(tpl({name: 'Jane'}), 'Hello Jane!');
```

17.7.1 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>

```

17.7.2 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`

17.8 Further reading

- How to implement your own tag functions is described in “Exploring ES6⁶”.



Quiz

See [quiz app](#).

⁶http://exploringjs.com/es6/ch_template-literals.html

Chapter 18

Symbols

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,
};
```

18.1 Use cases for symbols

The main use cases for symbols are:

- Defining constants for the values of an enumerated type.
- Creating unique property keys.

18.1.1 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'));
```

18.1.2 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](#).

18.2 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`

18.3 Converting symbols

What happens if we convert a symbol `sym` to another primitive type? Tbl. 18.1 has the answers.

Table 18.1: 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)'

```

18.4 Further reading

- In-depth coverage of symbols (cross-realm symbols, etc.): see “Exploring ES6¹”

 Quiz

See quiz app.

¹http://exploringjs.com/es6/ch_symbols.html

Part V

Control flow and data flow

Chapter 19

Control flow statements

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.

19.1 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.

19.1.1 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'
```

19.1.2 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)' }
);
```

19.1.3 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'
```

19.2 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 ifs.

19.2.1 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.

19.3 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.

19.3.1 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');
```

19.3.2 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');

```

19.3.3 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);

```

19.3.4 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

19.4 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.

19.4.1 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;  
}
```

19.5 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');
```

19.6 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»  
}
```

19.6.1 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;
}
```

19.7 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);
}
```

19.7.1 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.

19.7.2 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);
}
```

19.7.3 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``

19.8 `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.

19.9 `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`:

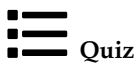
```
function getOwnPropertyNames(obj) {
  const result = [];
  for (const key in obj) {
```

```
    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 JavaScript¹”.



Quiz

See quiz app.

¹<http://speakingjs.com/es5/ch13.html#for-in>

Chapter 20

Callable values

20.1 Kinds of functions

JavaScript has two categories of functions:

- An *ordinary function* can be several things at the same time: a real function, a method and a constructor function (we'll get to what exactly these things are, in a moment).
- A *specialized function* can only be one of those things. There are three such functions:
 - An *arrow function* can only be a real function.
 - A *method* can only be a method.
 - A *class* can only be a constructor function.

Let's look at ordinary functions and specialized functions in greater detail.

20.1.1 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.

20.1.1.1 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.)

20.1.2 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. 20.1 lists the capabilities of ordinary and specialized functions.

Table 20.1: Capabilities of the four kinds of functions.

	Ordinary function	Arrow function	Method	Class
Function call	✓	✓	✓	✗
Method call	✓	lexical this	✓	✗
Constructor call	✓	✗	✗	✓

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
```

20.1.3 More kinds of real functions and methods

Warning: You are about to be faced with a long list of things and few explanations. This is just to give you a brief overview. This chapter focuses on synchronous real functions (all of which we have already seen). Later chapters will explain everything 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. 20.2).

Table 20.2: 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. 20.3 gives an overview of the syntax for creating these constructs (a Promise is a mechanism for delivering asynchronous results).

Table 20.3: Syntax for creating real functions and methods.

Sync function	Sync method	Result	Values
function f() {} f = function () {} f = () => {}	{ m() {} }	value	1
Sync generator function function* f() {} f = function* () {}	Sync gen. method { * m() {} }	iterable	0+
Async function async function f() {} f = async function () {} f = async () => {}	Async method { async m() {} }	Promise	1
Async generator function async function* f() {} f = async function* () {}	Async gen. method { async * m() {} }	async iterable	0+

For the remainder of this chapter, we'll explore real functions and their foundations.

20.2 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`.

20.3 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](#)).

20.3.1 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 });
```

20.3.2 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*).

20.3.3 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.deepStrictEqual(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.deepStrictEqual(func2(), undefined);
```

{a: 1} is interpreted as a block with the label a: and the expression statement 1.

20.4 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);
```

20.5 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);
```

20.6 Parameter handling

20.6.1 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*.

20.6.2 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]);

```

20.6.3 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]);

```

20.6.4 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, []]);

```

20.6.4.1 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;
  // ...
}
```

20.6.5 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`).

20.6.6 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 });
```

20.6.7 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.

20.6.7.1 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
```

20.6.7.2 Example: spreading into `Array.prototype.push()`

Similarly, the `Array` method `.push()` destructively adds its zero or more parameters to the end of its receiver. 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`

20.7 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

 Quiz

See quiz app.

Chapter 21

Where are the remaining chapters?

You are reading a preview of the book: For now, you have to buy it to get access to the complete contents. There will eventually be a version that is free to read online (with most of the chapters, but without exercises and quizzes). Current estimate: early 2019.