JavaScript

FOR IMPATIENT PROGRAMMERS

Dr. Axel Rauschmayer

JavaScript for impatient programmers

Dr. Axel Rauschmayer

2019

"An exhaustive resource, yet cuts out the fluff that clutters many programming books – with explanations that are understandable and to the point, as promised by the title! The quizzes and exercises are a very useful feature to check and lock in your knowledge. And you can definitely tear through the book fairly quickly, to get up and running in JavaScript."
— **Pam Selle**, thewebivore.com

"The best introductory book for modern JavaScript."
— **Tejinder Singh**, Senior Software Engineer, IBM

"This is JavaScript. No filler. No frameworks. No third-party libraries. If you want to learn JavaScript, you need this book."
— **Shelley Powers**, Software Engineer/Writer

ISBN 978-1-09-121009-7

exploringjs.com

Contents

Part I

Background

Chapter 1

About this book (ES2019 edition)

Contents

1.1 About the content

1.1.1 What's in this book?

This book makes JavaScript less challenging to learn for newcomers by offering a modern view that is as consistent as possible.

Highlights:

- Get started quickly by initially focusing on modern features.
- Test-driven exercises and quizzes available for most chapters.
- Covers all essential features of JavaScript, up to and including ES2019.
- Optional advanced sections let you dig deeper.

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

1.1.2 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 `Explor-`

ingJS.com, which are free to read online.
- This book deliberately focuses on the language. Browser-only features, etc. are not described.

1.1.3 Isn't this book too long for impatient people?

There are several ways in which you can read this book. One of them involves skipping much of the content in order to get started quickly. For details, see §2.1.1 "In which order should I read the content in this book?".

1.2 Previewing and buying this book

1.2.1 How can I preview the book, the exercises, and the quizzes?

Go to the homepage of this book:

- All essential chapters of this book can be read online.
- The first half of the test-driven exercises can be downloaded.
- The first half of the quizzes can be tried online.

1.2.2 How can I buy a digital edition of this book?

There are two digital editions of *JavaScript for impatient programmers*:

- Ebooks: PDF, EPUB, MOBI, HTML (all without DRM)
- Ebooks plus exercises and quizzes

The home page of this book describes how you can buy them.

1.2.3 How can I buy the print edition of this book?

The print edition of *JavaScript for impatient programmers* is available on Amazon.

1.3 About the author

Dr. Axel Rauschmayer specializes in JavaScript and web development. He has been developing web applications since 1995. In 1999, he was technical manager at a German internet startup that later expanded internationally. In 2006, he held his first talk on Ajax. In 2010, he received a PhD in Informatics from the University of Munich.

Since 2011, he has been blogging about web development at 2ality.com and has written several books on JavaScript. He has held trainings and talks for companies such as eBay, Bank of America, and O'Reilly Media.

He lives in Munich, Germany.

1.4 Acknowledgements

- Cover by Fran Caye

- Parts of this book were edited by Adaobi Obi Tulton.
- Thanks for answering questions, discussing language topics, etc.:
  - Allen Wirfs-Brock (`@awbjs`)
  - Benedikt Meurer (`@bmeurer`)
  - Brian Terlson (`@bterlson`)
  - Daniel Ehrenberg (`@littledan`)
  - Jordan Harband (`@ljharb`)
  - Mathias Bynens (`@mathias`)
  - Myles Borins (`@MylesBorins`)
  - Rob Palmer (`@robpalmer2`)
  - Šime Vidas (`@simevidas`)
  - And many others
- Thanks for reviewing:
  - Johannes Weber (`@jowe`)

[Generated: 2019-08-31 17:40]

Chapter 2

FAQ: Book and supplementary material

Contents

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

2.1 How to read this book

2.1.1 In which order should I read the content in this book?

This book is three books in one:

- You can use it to get started with JavaScript as quickly as possible. This "mode" is for impatient people:

  - Start reading with §5 "The big picture".
  - Skip all chapters and sections marked as "advanced", and all quick references.
- It gives you a comprehensive look at current JavaScript. In this "mode", you read everything and don't skip advanced content and quick references.
- It serves as a reference. If there is a topic that you are interested in, you can find information on it via the table of contents or via the index. Due to basic and advanced content being mixed, everything you need is usually in a single location.

The quizzes and exercises play an important part in helping you practice and retain what you have learned.

2.1.2 Why are some chapters and sections marked with "(advanced)"?

Several chapters and sections are marked with "(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.

2.1.3 Why are some chapters marked with "(bonus)"?

The bonus chapters are only available in the paid versions of this book (print and ebook). They are listed in the full table of contents.

2.2 I own a digital edition

2.2.1 How do I submit feedback and corrections?

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

2.2.2 How do I get updates for the downloads I bought at Payhip?

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

2.2.3 Can I upgrade from package "Ebooks" to package "Ebooks + exercises + quizzes"?

Yes. The instructions for doing so are on the homepage of this book.

2.3 I own the print edition

2.3.1 Can I get a discount for a digital edition?

If you bought the print edition, you can get a discount for a digital edition. The homepage of the print edition explains how.

Alas, the reverse is not possible: you cannot get a discount for the print edition if you bought a digital edition.

2.3.2 Can I submit an error or see submitted errors?

On the homepage of the print edition, you can submit errors and see submitted errors.

2.3.3 Is there an online list with the URLs in this book?

The homepage of the print edition has a list with all the URLs that you see in the footnotes of the print edition.

2.4 Notations and conventions

2.4.1 What is a type signature? Why am I seeing static types in this book?

For example, you may see:

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

That is called the *type signature* of `Number.isFinite()`. This notation, especially the static types `number` of `num` and `boolean` of the result, are not real JavaScript. The notation is 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 notation is explained in detail in a 2ality blog post, but is usually relatively intuitive.

2.4.2 What do the notes with icons mean?

Reading instructions

Explains how to best read the content.

External content

Points to additional, external, content.

Tip

Gives a tip related to the current content.

Question

Asks and answers a question pertinent to the current content (think FAQ).

Warning

Warns about pitfalls, etc.

Details

Provides additional details, complementing the current content. It is similar to a footnote.

Exercise

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

Quiz

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

Chapter 3

History and evolution of JavaScript

Contents

3.1 How JavaScript was created

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 several times:

- 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 Standardizing JavaScript

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* (with "Ecma" being a proper name, not an acronym) because the organization's activities had expanded beyond Europe. The initial all-caps acronym explains the spelling of ECMAScript.

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

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

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

3.3 Timeline of ECMAScript versions

This is a brief timeline of ECMAScript versions:

- ECMAScript 1 (June 1997): First version of the standard.
- ECMAScript 2 (June 1998): Small update to keep ECMA-262 in sync with the ISO standard.
- ECMAScript 3 (December 1999): Adds many core features – "[…] regular expressions, better string handling, new control statements [do-while, switch], try/catch exception handling, […]"
- ECMAScript 4 (abandoned in July 2008): Would have been a massive upgrade (with static typing, modules, namespaces, and more), but ended up being too ambitious and dividing the language's stewards.
- 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. The shorter release life cycle resulted in fewer new features compared to the large ES6.
- ECMAScript 2017 (June 2017). Second yearly release.
- Subsequent ECMAScript versions (ES2018, etc.) are always ratified in June.

3.4 Ecma Technical Committee 39 (TC39)

TC39 is the committee that evolves JavaScript. Its member 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.

Every two months, TC39 has meetings that member-appointed delegates and invited experts attend. The minutes of those meetings are public in a GitHub repository.

3.5 The TC39 process

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

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

In response to these issues, TC39 instituted the new *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.

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

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

Up to and including ES6, it was most common to think about JavaScript in terms of ECMAScript versions – for example, "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.

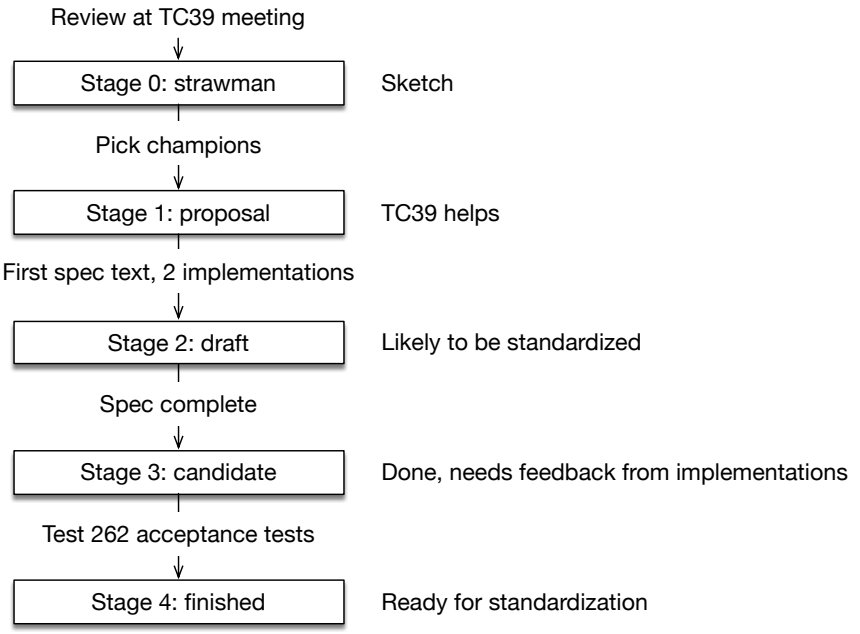

Figure 3.1: Each ECMAScript feature proposal goes through stages that are numbered from 0 to 4. *Champions* are TC39 members that support the authors of a feature. Test 262 is a suite of tests that checks JavaScript engines for compliance with the language specification.

3.6 FAQ: TC39 process

3.6.1 How is [my favorite proposed feature] doing?

If you are wondering what stages various proposed features are in, consult the GitHub repository `proposals`.

3.6.2 Is there an official list of ECMAScript features?

Yes, the TC39 repo lists finished proposals and mentions in which ECMAScript versions they were introduced.

3.7 Evolving JavaScript: Don't break the web

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

Let's assume we create a new version of JavaScript that is not backward compatible and fix 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 you 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 a 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 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.

Quiz

See quiz app.

Chapter 4

FAQ: JavaScript

Contents

4.1 What are good references for JavaScript?

Please consult §5.3 "JavaScript references".

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

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

- ECMAScript compatibility tables for various engines (by kangax, webbedspace, zloirock)
- Node.js compatibility tables (by William Kapke)
- Mozilla's MDN web docs have tables for each feature that describe relevant ECMAScript versions and browser support.
- "Can I use…" documents what features (including JavaScript language features) are supported by web browsers.

4.3 Where can I look up what features are planned for JavaScript?

Please consult the following sources:

- §3.5 "The TC39 process" describes how upcoming features are planned.
- §3.6 "FAQ: TC39 process" answers various questions regarding upcoming features.

4.4 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
```

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

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

This question is answered in §3.7 "Evolving JavaScript: Don't break the web".

4.6 How can I quickly try out a piece of JavaScript code?

§7.1 "Trying out JavaScript code" explains how to do that.

Part II

First steps

Chapter 5

The big picture

Contents

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 browser
- 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 software registry, 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 foundational layer consists of the JavaScript engine and platform-specific "core" functionality.

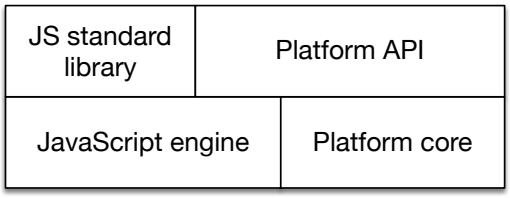

Figure 5.1: The structure of the two JavaScript platforms *web browser* and *Node.js*. The APIs "standard library" and "platform API" are hosted on top of a foundational layer with a JavaScript engine and a platform-specific "core".

- Two APIs are hosted on top of this foundation:
  - The JavaScript standard library is part of JavaScript proper and runs on top of the engine.
  - The platform API are also available from JavaScript – it provides access to platform-specific functionality. For example:
    * In browsers, you need to use the platform-specific API if you want to do anything related to the user interface: react to mouse clicks, play sounds, etc.
    * In Node.js, the platform-specific API lets you read and write files, download data via HTTP, etc.

5.3 JavaScript references

When you have a question about a JavaScript, a web search usually helps. I can recommend the following online sources:

- MDN web docs: cover various web technologies such as CSS, HTML, JavaScript, and more. An excellent reference.
- Node.js Docs: document the Node.js API.
- ExploringJS.com: My other books on JavaScript go into greater detail than this book and are free to read online. You can look up features by ECMAScript version:
  - ES1–ES5: *Speaking JavaScript*
  - ES6: *Exploring ES6*
  - ES2016–ES2017: *Exploring ES2016 and ES2017*
  - Etc.

5.4 Further reading

- A bonus chapter provides a more comprehensive look at web development.

Chapter 6

Syntax

Contents

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 a single type for numbers)
-123
1.141

// Strings (JavaScript has no type for characters)
'abc'
"abc"
```

An *assertion* describes what the result of a computation is expected to look like and throws an exception if those expectations aren't correct. For example, the following assertion states that the result of the computation 7 plus 1 must be 8:

```
assert.equal(7 + 1, 8);
```

`assert.equal()` is a method call (the object is `assert`, the method is `.equal()`) with two arguments: the actual result and the expected result. It is part of a Node.js assertion API that is explained later in this book.

Logging to the console of a browser or Node.js:

```
// Printing a value to standard out (another method call)
console.log('Hello!');

// Printing error information to standard error
console.error('Something went wrong!');
```

Operators:

```
// Operators for booleans
assert.equal(true && false, false); // And
```

```
assert.equal(true || false, true); // Or

// Operators for numbers
assert.equal(3 + 4, 7);
assert.equal(5 - 1, 4);
assert.equal(3 * 4, 12);
assert.equal(9 / 3, 3);

// Operators for strings
assert.equal('a' + 'b', 'ab');
assert.equal('I see ' + 3 + ' monkeys', 'I see 3 monkeys');

// Comparison operators
assert.equal(3 < 4, true);
assert.equal(3 <= 4, true);
assert.equal('abc' === 'abc', true);
assert.equal('abc' !== 'def', true);
```

Declaring variables:

```
let x; // declaring x (mutable)
x = 3 * 5; // assign a value to x

let y = 3 * 5; // declaring and assigning

const z = 8; // declaring 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 as arguments of function calls and method calls):

```
const add2 = (a, b) => { return a + b };
// Calling function add2()
assert.equal(add2(5, 2), 7);

// Equivalent to add2:
const add3 = (a, b) => a + b;
```

The previous code contains the following two arrow functions (the terms *expression* and *statement* are explained later in this chapter):

```
// An arrow function whose body is a code block
(a, b) => { return a + b }

// An arrow function whose body is an expression
(a, b) => a + b
```

Objects:

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

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

// Calling 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'];

// Getting an Array element
assert.equal(arr[1], 'b');
// Setting an 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.mjs
main.mjs
```

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

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

The module in `main.mjs` 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.mjs
import {isTextFilePath} from './file-tools.mjs';
```

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 Casing styles

Common casing styles for concatenating words are:

- Camel case: `threeConcatenatedWords`
- Underscore case (also called *snake case*): `three_concatenated_words`
- Dash case (also called *kebab case*): `three-concatenated-words`

6.1.5 Capitalization of names

In general, JavaScript uses camel case, except for constants.

Lowercase:

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

Uppercase:

- Classes: `MyClass`
- Constants: `MY_CONSTANT`
  - Constants are also often written in camel case: `myConstant`

6.1.6 More naming conventions

The following naming conventions are popular in JavaScript.

If the name of a parameter starts with an underscore (or is an underscore) it means that this parameter is not used – for example:

```
arr.map((_x, i) => i)
```

If the name of a property of an object starts with an underscore then that property is considered private:

```
class ValueWrapper {
  constructor(value) {
    this._value = value;
  }
}
```

6.1.7 Where to put semicolons?

At the end of a statement:

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

But not if that statement ends with a curly brace:

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

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

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

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

Quiz: basic

See quiz app.

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.

All JavaScript *keywords* are reserved words:

```
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 tokens are also keywords, but currently not used in the language:

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

The following literals are reserved words:

```
true false null
```

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

```
Infinity NaN undefined async
```

You shouldn't use the names of global variables (`String`, `Math`, etc.) for your own variables and parameters, either.

6.4 Statement vs. expression

In this section, we explore how JavaScript distinguishes two kinds of syntactic constructs: *statements* and *expressions*. Afterward, we'll see that that can cause problems because the same syntax can mean different things, depending on where it is used.

We pretend there are only statements and expressions

For the sake of simplicity, we pretend that there are only statements and expressions in JavaScript.

6.4.1 Statements

A *statement* is a piece of code that can be executed and performs some kind of action. For example, `if` is a statement:

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

One more example of a statement: a function declaration.

```
function twice(x) {
  return x + x;
}
```

6.4.2 Expressions

An *expression* is a piece of code that can be *evaluated* to produce a value. 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.

Let's look at more examples of expressions. We enter expressions and the REPL evaluates them for us:

```
> 'ab' + 'cd'
'abcd'
> Number('123')
123
> true || false
true
```

6.4.3 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:

```
function max(x, y) {
  if (x > y) {
    return x;
  } else {
    return y;
  }
}
```

- The arguments of a function call or a method call must be expressions:

```
console.log('ab' + 'cd', Number('123'));
```

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 a statement.

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

```
function f() {
  console.log(bar()); // bar() is expression
  bar(); // bar(); is (expression) statement
}
```

6.5 Ambiguous syntax

JavaScript has several programming constructs that are syntactically ambiguous: the same syntax is interpreted differently, depending on whether it is used in statement context or in expression context. This section explores the phenomenon and the pitfalls it causes.

6.5.1 Same syntax: function declaration and function expression

A *function declaration* is a statement:

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

A *function expression* is an expression (right-hand side of =):

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

6.5.2 Same syntax: object literal and block

In the following code, `{}` is an *object literal*: an expression that creates an empty object.

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

This is an empty code block (a statement):

```
{
}
```

6.5.3 Disambiguation

The ambiguities are only a problem in statement context: If the JavaScript parser encounters ambiguous syntax, it doesn't know if it's a plain statement or an expression statement. For example:

- If a statement starts with `function`: Is it a function declaration or a function expression?
- If a statement starts with `{`: Is it an object literal or a code block?

To resolve the ambiguity, statements starting with `function` or `{` are never interpreted as expressions. If you want an expression statement to start with either one of these tokens, you must wrap it in parentheses:

```
(function (x) { console.log(x) })('abc');

// Output:
// 'abc'
```

In this code:

1. We first create a function via a function expression:

   ```
   function (x) { console.log(x) }
   ```

2. Then we invoke that function: `('abc')`

The code fragment shown in (1) is only interpreted as an expression because we wrap it in parentheses. If we didn't, we would get a syntax error because then JavaScript expects a function declaration and complains about the missing function name. Additionally, you can't put a function call immediately after a function declaration.

Later in this book, we'll see more examples of pitfalls caused by syntactic ambiguity:

- Assigning via object destructuring
- Returning an object literal from an arrow function

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 = () => {}; // semicolon!
```

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

- Return statement without operand: `return;`
- Start of code block: `{`
- Expression statement `'jane';` with label `first:`
- End of code block: `}`
- 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. The following are three examples. There are more.

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'); // comma operator
assert.equal(propKey, 'ol');

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

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 Prettier can be configured to not use semicolons. It then automatically fixes problems. For example, if it encounters a line that starts with a square bracket, it prefixes that line with a semicolon.
- The static checker ESLint has a rule that you tell your preferred style (always semicolons or as few semicolons as possible) and that warns you about critical issues.

6.9 Strict mode vs. sloppy mode

Starting with ECMAScript 5, JavaScript has two *modes* in which JavaScript can be executed:

- Normal "sloppy" mode is the default in scripts (code fragments that are a precursor to modules and supported by browsers).
- Strict mode is the default in modules and classes, and can be switched on in scripts (how, is explained later). In this mode, several pitfalls of normal mode are removed and more exceptions are thrown.

You'll rarely encounter sloppy mode in modern JavaScript code, which is almost always located in modules. In this book, I assume that strict mode is always switched on.

6.9.1 Switching on strict mode

In 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 Improvements in strict mode

Let's look at three things that strict mode does better than sloppy mode. Just in this one section, all code fragments are executed in sloppy mode.

6.9.2.1 Sloppy mode pitfall: changing an undeclared variable creates a global variable

In non-strict mode, changing an undeclared variable creates a global variable.

```
function sloppyFunc() {
  undeclaredVar1 = 123;
}
sloppyFunc();
// Created global variable `undeclaredVar1`:
assert.equal(undeclaredVar1, 123);
```

Strict mode does it better and throws a `ReferenceError`. That makes it easier to detect typos.

```
function strictFunc() {
  'use strict';
  undeclaredVar2 = 123;
}
assert.throws(
  () => strictFunc(),
  {
    name: 'ReferenceError',
    message: 'undeclaredVar2 is not defined',
  });
```

The `assert.throws()` states that its first argument, a function, throws a `ReferenceError` when it is called.

6.9.2.2 Function declarations are block-scoped in strict mode, function-scoped in sloppy mode

In strict mode, a variable created via a function declaration only exists within the innermost enclosing block:

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

In sloppy mode, function declarations are function-scoped:

```
function sloppyFunc() {
  {
    function foo() { return 123 }
  }
  return foo(); // works
}
assert.equal(sloppyFunc(), 123);
```

6.9.2.3 Sloppy mode doesn't throw exceptions when changing immutable data

In strict mode, you get an exception if you try to change immutable data:

```
function strictFunc() {
  'use strict';
  true.prop = 1; // TypeError
}
assert.throws(
  () => strictFunc(),
  {
    name: 'TypeError',
    message: "Cannot create property 'prop' on boolean 'true'",
  });
```

In sloppy mode, the assignment fails silently:

```
function sloppyFunc() {
  true.prop = 1; // fails silently
  return true.prop;
}
assert.equal(sloppyFunc(), undefined);
```

Further reading: sloppy mode

For more information on how sloppy mode differs from strict mode, see MDN.

Quiz: advanced

See quiz app.

Chapter 7

Consoles: interactive JavaScript command lines

Contents

7.1 Trying out JavaScript code

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

7.1.1 Browser consoles

Web browsers have so-called *consoles*: interactive command lines to which you can print text via `console.log()` and where you can run pieces of code. How to open the console differs from browser to browser. Fig. 7.1 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 a few commonly used web browsers:

- Apple Safari
- Google Chrome
- Microsoft Edge
- Mozilla Firefox

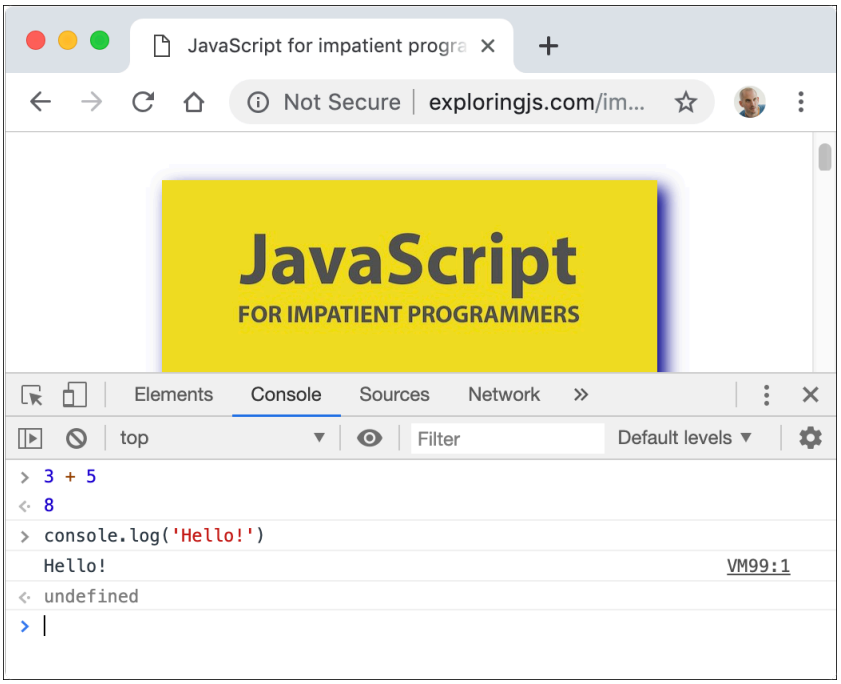

Figure 7.1: The console of the web browser "Google Chrome" is open (in the bottom half of window) while visiting a web page.

7.1.2 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. 7.2: The text after > is input from the user; everything else is output from Node.js.

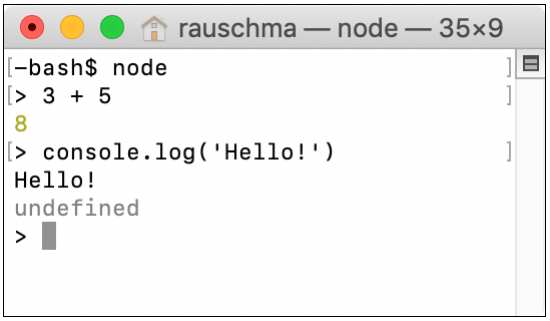

Figure 7.2: Starting and using the Node.js REPL (interactive command line).

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
```

7.1.3 Other options

Other options include:

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

Consoles often run in non-strict mode

In modern JavaScript, most code (e.g., modules) is executed in strict mode. However, consoles often run in non-strict mode. Therefore, you may occasionally get slightly different results when using a console to execute code from this book.

7.2 The `console.*` API: printing data and more

In browsers, the console is something you can bring up that is normally hidden. For Node.js, the console is the terminal that Node.js is currently running in.

The full `console.*` API is documented on MDN web docs and on the Node.js website. It is not part of the JavaScript language standard, but much functionality is supported by both browsers and Node.js.

In this chapter, we only look at the following two methods for printing data ("printing" means displaying in the console):

- `console.log()`
- `console.error()`

7.2.1 Printing values: `console.log()` (stdout)

There are two variants of this operation:

```
console.log(...values: any[]): void
console.log(pattern: string, ...values: any[]): void
```

7.2.1.1 Printing multiple values

The first variant prints (text representations of) values on the console:

```
console.log('abc', 123, true);
// Output:
// abc 123 true
```

At the end, `console.log()` always prints a newline. Therefore, if you call it with zero arguments, it just prints a newline.

7.2.1.2 Printing a string with substitutions

The second variant performs string substitution:

```
console.log('Test: %s %j', 123, 'abc');
// Output:
// Test: 123 "abc"
```

These are some of the directives you can use for substitutions:

- `%s` converts the corresponding value to a string and inserts it.

  ```
  console.log('%s %s', 'abc', 123);
  // Output:
  // abc 123
  ```

- `%o` inserts a string representation of an object.

  ```
  console.log('%o', {foo: 123, bar: 'abc'});
  // Output:
  // { foo: 123, bar: 'abc' }
  ```

- `%j` converts a value to a JSON string and inserts it.

  ```
  console.log('%j', {foo: 123, bar: 'abc'});
  // Output:
  // {"foo":123,"bar":"abc"}
  ```

- `%%` inserts a single `%`.

```
console.log('%s%%', 99);
// Output:
// 99%
```

7.2.2 Printing error information: `console.error()` (stderr)

`console.error()` works the same as `console.log()`, but what it logs is considered error information. For Node.js, that means that the output goes to stderr instead of stdout on Unix.

7.2.3 Printing nested objects via `JSON.stringify()`

`JSON.stringify()` is occasionally useful for printing nested objects:

```
console.log(JSON.stringify({first: 'Jane', last: 'Doe'}, null, 2));
```

Output:

```
{
  "first": "Jane",
  "last": "Doe"
}
```

Chapter 8

Assertion API

Contents

8.1 Assertions in software development

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

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

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

8.2 How assertions are used in this book

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

8.2.1 Documenting results in code examples via assertions

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

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

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

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

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

The motivation behind using assertions is:

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

8.2.2 Implementing test-driven exercises via assertions

The exercises for this book are test-driven, via the test framework AVA. 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', t => {
  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 §9 “Getting started with quizzes and exercises”.

8.3 Normal comparison vs. deep comparison

The strict `equal()` uses `===` to compare values. Therefore, an object is only equal to itself – even if another object has the same content (because `===` does not compare the contents of objects, only their identities):

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

`deepEqual()` is a better choice for comparing objects:

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

This method works for Arrays, too:

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

8.4 Quick reference: module `assert`

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

8.4.1 Normal equality

- `function equal(actual: any, expected: any, message?: string): void`

  `actual === expected` must be `true`. If not, an `AssertionError` is thrown.

  ```
  assert.equal(3+3, 6);
  ```

- `function notEqual(actual: any, expected: any, message?: string): void`

  `actual !== expected` must be `true`. If not, an `AssertionError` is thrown.

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

The optional last parameter `message` can be used to explain what is asserted. If the assertion fails, the message is used to set up the `AssertionError` that is thrown.

```
let e;
try {
  const x = 3;
  assert.equal(x, 8, 'x must be equal to 8')
} catch (err) {
  assert.equal(
    String(err),
    'AssertionError [ERR_ASSERTION]: x must be equal to 8');
}
```

8.4.2 Deep equality

- `function deepEqual(actual: any, expected: any, message?: string): void`

  `actual` must be deeply equal to `expected`. If not, an `AssertionError` is thrown.

  ```
  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`

  `actual` must not be deeply equal to `expected`. If it is, an `AssertionError` is thrown.

  ```
  assert.notDeepEqual([1,2,3], [1,2]);
  ```

8.4.3 Expecting exceptions

If you want to (or expect to) receive an exception, you need `throws()`: This function calls its first parameter, the function `block`, and only succeeds if it throws an exception. Additional parameters can be used to specify what that exception must look like.

- `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`,
  }
);
```

8.4.4 Another tool function

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

  Always throws an `AssertionError` when it is called. That is occasionally useful for unit testing.

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

Quiz

See quiz app.

Chapter 9

Getting started with quizzes and exercises

Contents

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

9.1 Quizzes

Installation:

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

Running the quiz app:

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

9.2 Exercises

9.2.1 Installing the exercises

To install the exercises:

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

9.2.2 Running exercises

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

9.3 Unit tests in JavaScript

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

9.3.1 A typical test

Typical test code is split into two parts:

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

Take, for example, the following two files:

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

9.3.1.1 Part 1: the code

The code itself resides in `id.mjs`:

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

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

9.3.1.2 Part 2: the tests

Don't worry about the exact details of tests

You don't need to worry about the exact details of tests: They are always implemented for you. Therefore, you only need to read them, but not write them.

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

```
// npm t demos/quizzes-exercises/id_test.mjs

import test from 'ava'; // (A)
```

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

test('My test', t => { // (D)
  assert.equal(id('abc'), 'abc'); // (E)
});
```

The core of this test file is line E – an assertion: `assert.equal()` specifies that the expected result of `id('abc')` is `'abc'`.

As for the other lines:

- The comment at the very beginning shows the shell command for running the test.
- Line A: We import the test framework.
- Line B: We import the assertion library. AVA has built-in assertions, but module `assert` lets us remain compatible with plain Node.js.
- Line C: We import the function to test.
- Line D: We define a test. This is done by calling the function `test()`:
  - First parameter: the name of the test.
  - Second parameter: the test code, which is provided via an arrow function. The parameter `t` gives us access to AVA's testing API (assertions, etc.).

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

```
npm t demos/quizzes-exercises/id_test.mjs
```

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

```
npm test demos/quizzes-exercises/id_test.mjs
```

Exercise: Your first exercise

The following exercise gives you a first taste of what exercises are like:

- `exercises/quizzes-exercises/first_module_test.mjs`

9.3.2 Asynchronous tests in AVA

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 AVA that it isn't finished yet when it returns. The following subsections examine three ways of doing so.

9.3.2.1 Asynchronicity via callbacks

If we call `test.cb()` instead of `test()`, AVA switches to callback-based asynchronicity. When we are done with our asynchronous work, we have to call `t.end()`:

```
test.cb('divideCallback', t => {
  divideCallback(8, 4, (error, result) => {
    if (error) {
      t.end(error);
    } else {
      assert.strictEqual(result, 2);
      t.end();
    }
  });
});
```

9.3.2.2 Asynchronicity via Promises

If a test returns a Promise, AVA switches to Promise-based asynchronicity. A test is considered successful if the Promise is fulfilled and failed if the Promise is rejected.

```
test('dividePromise 1', t => {
  return dividePromise(8, 4)
  .then(result => {
    assert.strictEqual(result, 2);
  });
});
```

9.3.2.3 Async functions as test "bodies"

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('dividePromise 2', async t => {
  const result = await dividePromise(8, 4);
  assert.strictEqual(result, 2);
  // No explicit return necessary!
});
```

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

Part III

Variables and values

Chapter 10

Variables and assignment

Contents

These are JavaScript's main ways of declaring variables:

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

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

10.1 `let`

Variables declared via `let` are mutable:

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

You can also declare and assign at the same time:

```
let i = 0;
```

10.2 `const`

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

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

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

10.2.1 `const` and immutability

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

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

// Allowed: changing properties of `obj`
obj.prop = obj.prop + 1;
assert.equal(obj.prop, 1);

// Not allowed: assigning to `obj`
assert.throws(
  () => { obj = {} },
  {
    name: 'TypeError',
    message: 'Assignment to constant variable.',
  }
);
```

10.2.2 `const` and loops

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

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

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

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

10.3 Deciding between `const` and `let`

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

- `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.mjs`

10.4 The scope of a variable

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

```
{ // // Scope A. Accessible: x
  const x = 0;
  assert.equal(x, 0);
  { // Scope B. Accessible: x, y
    const y = 1;
    assert.equal(x, 0);
    assert.equal(y, 1);
    { // Scope C. Accessible: x, y, z
      const z = 2;
      assert.equal(x, 0);
      assert.equal(y, 1);
```

```
        assert.equal(z, 2);
      }
    }
  }
  // Outside. Not accessible: x, y, z
  assert.throws(
    () => console.log(x),
    {
      name: 'ReferenceError',
      message: 'x is not defined',
    }
  );
```

- Scope A is the *(direct) scope* of `x`.
- Scopes B and C are *inner scopes* of scope A.
- Scope A is an *outer scope* of scope B and scope C.

Each variable is accessible in its direct scope and all scopes nested within that scope.

The variables declared via `const` and `let` are called *block-scoped* because their scopes are always the innermost surrounding blocks.

10.4.1 Shadowing variables

You can't declare the same variable twice at the same level:

```
assert.throws(
  () => {
    eval('let x = 1; let x = 2;');
  },
  {
    name: 'SyntaxError',
    message: "Identifier 'x' has already been declared",
  });
```

Why `eval()`?

`eval()` delays parsing (and therefore the `SyntaxError`), until the callback of `assert.throws()` is executed. If we didn't use it, we'd already get an error when this code is parsed and `assert.throws()` wouldn't even be executed.

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: basic**
>
> See quiz app.

10.5 (Advanced)

All remaining sections are advanced.

10.6 Terminology: static vs. dynamic

These two adjectives describe phenomena in programming languages:

- *Static* means that something is related to source code and can be determined without executing code.
- *Dynamic* means at runtime.

Let's look at examples for these two terms.

10.6.1 Static phenomenon: scopes of variables

Variable scopes are a static phenomenon. Consider the following code:

```
function f() {
  const x = 3;
  // ···
}
```

x is *statically* (or *lexically*) *scoped*. That is, its scope is fixed and doesn't change at runtime.

Variable scopes form a static tree (via static nesting).

10.6.2 Dynamic phenomenon: function calls

Function calls are a dynamic phenomenon. Consider the following code:

```
function g(x) {}
function h(y) {
  if (Math.random()) g(y); // (A)
}
```

Whether or not the function call in line A happens, can only be decided at runtime.

Function calls form a dynamic tree (via dynamic calls).

10.7 Global variables and the global object

JavaScript's variable scopes are nested. They form a tree:

- The outermost scope is the root of the tree.
- The scopes directly contained in that scope are the children of the root.
- And so on.

The root is also called the *global scope*. In web browsers, the only location where one is directly in that scope is at the top level of a script. The variables of the global scope are called *global variables* and accessible everywhere. There are two kinds of global variables:

- *Global declarative variables* are normal variables.
  - They can only be created while at the top level of a script, via `const`, 'let, and class declarations.
- *Global object variables* are stored in properties of the so-called *global object*.
  - They are created in the top level of a script, via `var` and function declarations.
  - The global object can be accessed via the global variable `globalThis`. It can be used to create, read, and delete global object variables.
  - Other than that, global object variables work like normal variables.

The following HTML fragment demonstrates `globalThis` and the two kinds of global variables.

```
<script>
  const declarativeVariable = 'd';
  var objectVariable = 'o';
</script>
<script>
  // All scripts share the same top-level scope:
  console.log(declarativeVariable); // 'd'
  console.log(objectVariable); // 'o'

  // Not all declarations create properties of the global object:
  console.log(globalThis.declarativeVariable); // undefined
  console.log(globalThis.objectVariable); // 'o'
</script>
```

Each ECMAScript module has its own scope. Therefore, variables that exist at the top level of a module are not global. Fig. 10.1 illustrates how the various scopes are related.

10.7.1 `globalThis`

`globalThis` is new

`globalThis` is a new feature. Be sure that the JavaScript engines you are targeting support it. If they don't, switch to one of the alternatives mentioned below.

The global variable `globalThis` is the new standard way of accessing the global object. It got its name from the fact that it has the same value as `this` in global scope.

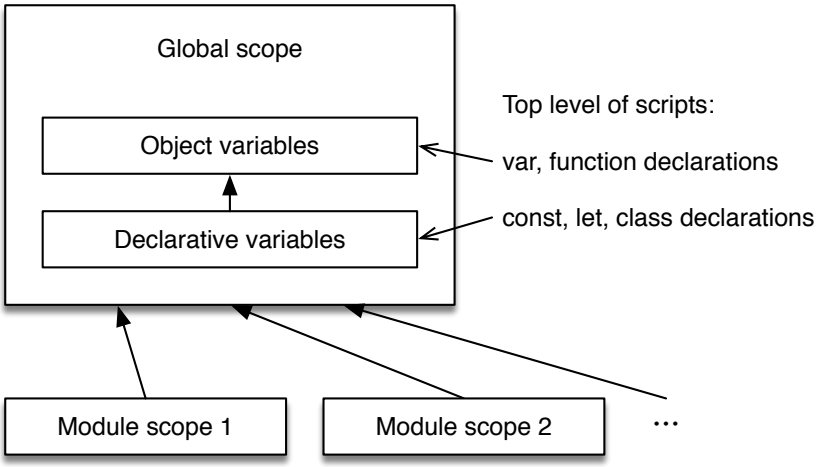

Figure 10.1: The global scope is JavaScript's outermost scope. It has two kinds of variables: *object variables* (managed via the *global object*) and normal *declarative variables*. Each ECMAScript module has its own scope which is contained in the global scope.

`globalThis` does not always directly point to the global object

For example, in browsers, there is an indirection. That indirection is normally not noticable, but it is there and can be observed.

10.7.1.1 Alternatives to `globalThis`

Older ways of accessing the global object depend on the platform:

- Global variable `window`: is the classic way of referring to the global object. But it doesn't work in Node.js and in Web Workers.
- Global variable `self`: is available in Web Workers and browsers in general. But it isn't supported by Node.js.
- Global variable `global`: is only available in Node.js.

10.7.1.2 Use cases for `globalThis`

The global object is now considered a mistake that JavaScript can't get rid of, due to backward compatibility. It affects performance negatively and is generally confusing.

ECMAScript 6 introduced several features that make it easier to avoid the global object – for example:

- `const`, `let`, and class declarations don't create global object properties when used in global scope.
- Each ECMAScript module has its own local scope.

It is usually better to access global object variables via variables and not via properties of `globalThis`. The former has always worked the same on all JavaScript platforms.

Tutorials on the web occasionally access global variables `globVar` via `window.globVar`. But the prefix "`window.`" is not necessary and I recommend to omit it:

```
window.encodeURIComponent(str); // no
encodeURIComponent(str); // yes
```

Therefore, there are relatively few use cases for `globalThis` – for example:

- *Polyfills* that add new features to old JavaScript engines.
- Feature detection, to find out what features a JavaScript engine supports.

10.8 Declarations: scope and activation

These are two key aspects of declarations:

- Scope: Where can a declared entity be seen? This is a static trait.
- Activation: When can I access an entity? This is a dynamic trait. Some entities can be accessed as soon as we enter their scopes. For others, we have to wait until execution reaches their declarations.

Tbl. 10.1 summarizes how various declarations handle these aspects.

Table 10.1: Aspects of declarations. "Duplicates" describes if a declaration can be used twice with the same name (per scope). "Global prop." describes if a declaration adds a property to the global object, when it is executed in the global scope of a script. *TDZ* means *temporal dead zone* (which is explained later). (*) Function declarations are normally block-scoped, but function-scoped in sloppy mode.

	Scope	Activation	Duplicates	Global prop.
`const`	Block	decl. (TDZ)	✘	✘
`let`	Block	decl. (TDZ)	✘	✘
`function`	Block (*)	start	✔	✔
`class`	Block	decl. (TDZ)	✘	✘
`import`	Module	same as export	✘	✘
`var`	Function	start, partially	✔	✔

`import` is described in §24.5 "ECMAScript modules". The following sections describe the other constructs in more detail.

10.8.1 `const` and `let`: temporal dead zone

For JavaScript, TC39 needed to decide what happens if you access a constant in its direct scope, before its declaration:

```
{
  console.log(x); // What happens here?
  const x;
}
```

Some possible approaches are:

1. The name is resolved in the scope surrounding the current scope.
2. You get `undefined`.
3. There is an error.

Approach 1 was rejected because there is no precedent in the language for this approach. It would therefore not be intuitive to JavaScript programmers.

Approach 2 was rejected because then `x` wouldn't be a constant – it would have different values before and after its declaration.

`let` uses the same approach 3 as `const`, so that both work similarly and it's easy to switch between them.

The time between entering the scope of a variable and executing its declaration is called the *temporal dead zone* (TDZ) of that variable:

- During this time, the variable is considered to be uninitialized (as if that were a special value it has).
- If you access an uninitialized variable, you get a `ReferenceError`.
- Once you reach a variable declaration, the variable is set to either the value of the initializer (specified via the assignment symbol) or `undefined` – if there is no initializer.

The following code illustrates the temporal dead zone:

```
if (true) { // entering scope of `tmp`, TDZ starts
  // `tmp` is uninitialized:
  assert.throws(() => (tmp = 'abc'), ReferenceError);
  assert.throws(() => console.log(tmp), ReferenceError);

  let tmp; // TDZ ends
  assert.equal(tmp, undefined);
}
```

The next example shows that the temporal dead zone is truly *temporal* (related to time):

```
if (true) { // entering scope of `myVar`, TDZ starts
  const func = () => {
    console.log(myVar); // executed later
  };

  // We are within the TDZ:
  // Accessing `myVar` causes `ReferenceError`

  let myVar = 3; // TDZ ends
  func(); // OK, called outside TDZ
}
```

Even though `func()` is located before the declaration of `myVar` and uses that variable, we can call `func()`. But we have to wait until the temporal dead zone of `myVar` is over.

10.8.2 Function declarations and early activation

More information on functions

In this section, we are using functions – before we had a chance to learn them properly. Hopefully, everything still makes sense. Whenever it doesn't, please see §23 "Callable values".

A function declaration is always executed when entering its scope, regardless of where it is located within that scope. That enables you to call a function `foo()` before it is declared:

```
assert.equal(foo(), 123); // OK
function foo() { return 123; }
```

The early activation of `foo()` means that the previous code is equivalent to:

```
function foo() { return 123; }
assert.equal(foo(), 123);
```

If you declare a function via `const` or `let`, then it is not activated early. In the following example, you can only use `bar()` after its declaration.

```
assert.throws(
  () => bar(), // before declaration
  ReferenceError);

const bar = () => { return 123; };

assert.equal(bar(), 123); // after declaration
```

10.8.2.1 Calling ahead without early activation

Even if a function `g()` is not activated early, it can be called by a preceding function `f()` (in the same scope) if we adhere to the following rule: `f()` must be invoked after the declaration of `g()`.

```
const f = () => g();
const g = () => 123;

// We call f() after g() was declared:
assert.equal(f(), 123);
```

The functions of a module are usually invoked after its complete body is executed. Therefore, in modules, you rarely need to worry about the order of functions.

Lastly, note how early activation automatically keeps the aforementioned rule: when entering a scope, all function declarations are executed first, before any calls are made.

10.8.2.2 A pitfall of early activation

If you rely on early activation to call a function before its declaration, then you need to be careful that it doesn't access data that isn't activated early.

```
funcDecl();

const MY_STR = 'abc';
function funcDecl() {
  assert.throws(
    () => MY_STR,
    ReferenceError);
}
```

The problem goes away if you make the call to `funcDecl()` after the declaration of `MY_STR`.

10.8.2.3 The pros and cons of early activation

We have seen that early activation has a pitfall and that you can get most of its benefits without using it. Therefore, it is better to avoid early activation. But I don't feel strongly about this and, as mentioned before, often use function declarations because I like their syntax.

10.8.3 Class declarations are not activated early

Even though they are similar to function declarations in some ways, class declarations are not activated early:

```
assert.throws(
  () => new MyClass(),
  ReferenceError);

class MyClass {}

assert.equal(new MyClass() instanceof MyClass, true);
```

Why is that? Consider the following class declaration:

```
class MyClass extends Object {}
```

The operand of `extends` is an expression. Therefore, you can do things like this:

```
const identity = x => x;
class MyClass extends identity(Object) {}
```

Evaluating such an expression must be done at the location where it is mentioned. Anything else would be confusing. That explains why class declarations are not activated early.

10.8.4 `var`: hoisting (partial early activation)

`var` is an older way of declaring variables that predates `const` and `let` (which are preferred now). Consider the following `var` declaration.

```
var x = 123;
```

This declaration has two parts:

- Declaration `var x`: The scope of a `var`-declared variable is the innermost surrounding function and not the innermost surrounding block, as for most other declarations. Such a variable is already active at the beginning of its scope and initialized with `undefined`.
- Assignment `x = 123`: The assignment is always executed in place.

The following code demonstrates the effects of `var`:

```
function f() {
  // Partial early activation:
  assert.equal(x, undefined);
  if (true) {
    var x = 123;
    // The assignment is executed in place:
    assert.equal(x, 123);
  }
  // Scope is function, not block:
  assert.equal(x, 123);
}
```

10.9 Closures

Before we can explore closures, we need to learn about bound variables and free variables.

10.9.1 Bound variables vs. free variables

Per scope, there is a set of variables that are mentioned. Among these variables we distinguish:

- *Bound variables* are declared within the scope. They are parameters and local variables.
- *Free variables* are declared externally. They are also called *non-local variables*.

Consider the following code:

```
function func(x) {
  const y = 123;
  console.log(z);
}
```

In the body of `func()`, `x` and `y` are bound variables. `z` is a free variable.

10.9.2 What is a closure?

What is a closure then?

> A *closure* is a function plus a connection to the variables that exist at its "birth place".

What is the point of keeping this connection? It provides the values for the free variables of the function – for example:

```
function funcFactory(value) {
  return () => {
    return value;
  };
}

const func = funcFactory('abc');
assert.equal(func(), 'abc'); // (A)
```

`funcFactory` returns a closure that is assigned to `func`. Because `func` has the connection to the variables at its birth place, it can still access the free variable `value` when it is called in line A (even though it "escaped" its scope).

All functions in JavaScript are closures

Static scoping is supported via closures in JavaScript. Therefore, every function is a closure.

10.9.3 Example: A factory for incrementors

The following function returns *incrementors* (a name that I just made up). An incrementor is a function that internally stores a number. When it is called, it updates that number by adding the argument to it and returns the new value.

```
function createInc(startValue) {
  return (step) => { // (A)
    startValue += step;
    return startValue;
  };
}
const inc = createInc(5);
assert.equal(inc(2), 7);
```

We can see that the function created in line A keeps its internal number in the free variable `startValue`. This time, we don't just read from the birth scope, we use it to store data that we change and that persists across function calls.

We can create more storage slots in the birth scope, via local variables:

```
function createInc(startValue) {
  let index = -1;
  return (step) => {
    startValue += step;
    index++;
    return [index, startValue];
  };
}
const inc = createInc(5);
assert.deepEqual(inc(2), [0, 7]);
assert.deepEqual(inc(2), [1, 9]);
```

```
assert.deepEqual(inc(2), [2, 11]);
```

10.9.4 Use cases for closures

What are closures good for?

- For starters, they are simply an implementation of static scoping. As such, they provide context data for callbacks.
- They can also be used by functions to store state that persists across function calls. `createInc()` is an example of that.
- And they can provide private data for objects (produced via literals or classes). The details of how that works are explained in *Exploring ES6*.

Quiz: advanced

See quiz app.

10.10 Further reading

For more information on how variables are handled under the hood (as described in the ECMAScript specification), consult a bonus chapter.

Chapter 11

Values

Contents

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

Supporting tool: `===`

In this chapter, we'll occasionally use the strict equality operator. `a === b` evaluates to `true` if `a` and `b` are equal. What exactly that means is explained in §12.4.2 "Strict equality (`===` and `!==`)".

11.1 What's a type?

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

11.2 JavaScript's type hierarchy

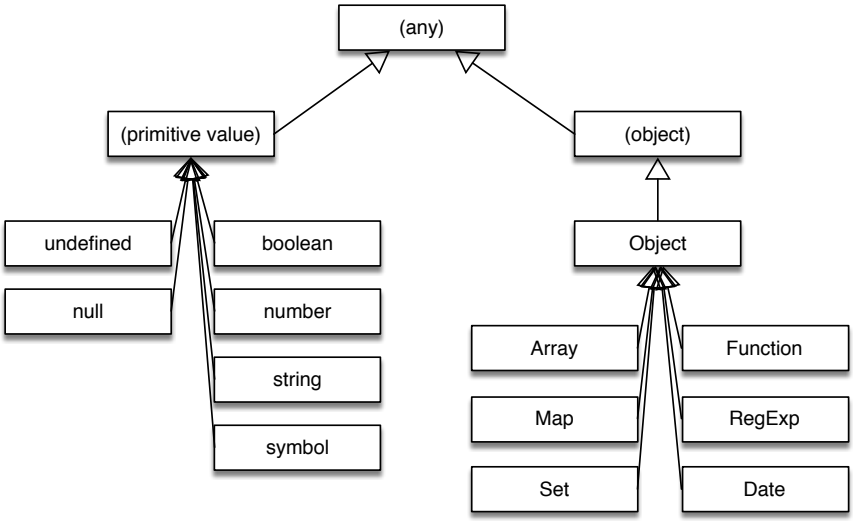

Figure 11.1: A partial hierarchy of JavaScript's types. Missing are the classes for errors, the classes associated with primitive types, and more. The diagram hints at the fact that not all objects are instances of `Object`.

Fig. 11.1 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.
- The diagram differentiates objects and instances of class `Object`. Each instance of `Object` is also an object, but not vice versa. However, virtually all objects that you'll encounter in practice are instances of `Object` – for example, objects created via object literals. More details on this topic are explained in §26.4.3.4 "Objects that aren't instances of `Object`".

11.3 The types of the language specification

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

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

11.4 Primitive values vs. 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), primitive values are not second-class citizens. The difference between them and objects is more subtle. In a nutshell:

- Primitive values: are atomic building blocks of data in JavaScript.
  - They are *passed by value*: when primitive values are assigned to variables or passed to functions, their contents are copied.
  - They are *compared by value*: when comparing two primitive values, their contents are compared.
- Objects: are compound pieces of data.
  - They are *passed by identity* (my term): when objects are assigned to variables or passed to functions, their *identities* (think pointers) are copied.
  - They are *compared by identity* (my term): when comparing two objects, their identities are compared.

Other than that, primitive values and objects are quite similar: they both have *properties* (key-value entries) and can be used in the same locations.

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

11.4.1 Primitive values (short: primitives)

11.4.1.1 Primitives are immutable

You can't change, add, or remove properties of primitives:

```
let str = 'abc';
assert.equal(str.length, 3);
assert.throws(
  () => { str.length = 1 },
  /^TypeError: Cannot assign to read only property 'length'/
);
```

11.4.1.2 Primitives are *passed by value*

Primitives are *passed by value*: variables (including parameters) store the contents of the primitives. When assigning a primitive value to a variable or passing it as an argument to a function, its content is copied.

```
let x = 123;
let y = x;
assert.equal(y, 123);
```

11.4.1.3 Primitives are *compared by value*

Primitives are *compared by value*: when comparing two primitive values, we compare their contents.

```
assert.equal(123 === 123, true);
assert.equal('abc' === 'abc', true);
```

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

11.4.2 Objects

Objects are covered in detail in §25 "Single objects" and the following chapter. Here, we mainly focus on how they differ from primitive values.

Let's first explore two common ways of creating objects:

- Object literal:

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

  The object literal starts and ends with curly braces `{}`. It creates an object with two properties. The first property has the key `'first'` (a string) and the value `'Jane'`. The second property has the key `'last'` and the value `'Doe'`. For more information on object literals, consult §25.2.1 "Object literals: properties".

- Array literal:

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

  The Array literal starts and ends with square brackets `[]`. It creates an Array with two *elements*: `'foo'` and `'bar'`. For more information on Array literals, consult [content not included].

11.4.2.1 Objects are mutable by default

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

```
const obj = {};

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

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

11.4.2.2 Objects are *passed by identity*

Objects are *passed by identity* (my term): variables (including parameters) store the *identities* of objects.

The identity of an object is like a pointer (or a transparent reference) to the object's actual data on the *heap* (think shared main memory of a JavaScript engine).

When assigning an object to a variable or passing it as an argument to a function, its identity is copied. Each object literal creates a fresh object on the heap and returns its identity.

```
const a = {}; // fresh empty object
// Pass the identity in `a` to `b`:
const b = a;

// Now `a` and `b` point to the same object
// (they "share" that object):
assert.equal(a === b, true);

// 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).

Details: passing by identity

"Passing by identity" means that the identity of an object (a transparent reference) is passed by value. This approach is also called "passing by sharing".

11.4.2.3 Objects are *compared by identity*

Objects are *compared by identity* (my term): two variables are only equal if they contain the same object identity. They are not equal if they refer to different objects with the same content.

```
const obj = {}; // fresh empty object
assert.equal(obj === obj, true); // same identity
assert.equal({} === {}, false); // different identities, same content
```

11.5 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) ···
```

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.

Rule of thumb: `typeof` is for primitive values; `instanceof` is for objects

11.5.1 `typeof`

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

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

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

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

Exercises: Two exercises on `typeof`

- `exercises/values/typeof_exrc.mjs`
- Bonus: `exercises/values/is_object_test.mjs`

11.5.2 `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/values/instanceof_exrc.mjs`

11.6 Classes and constructor functions

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

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

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

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

11.6.1 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);
  ```

- `Number.prototype` provides the properties for numbers – for example, method `.toString()`:

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

- `Number` is a namespace/container object for 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.notEqual(new Number(123), 123);
  assert.equal(new Number(123).valueOf(), 123);
  ```

11.6.1.1 Wrapping primitive values

The constructor functions related to primitive types are also called *wrapper types* because they provide the canonical way of converting primitive values to objects. In the process, primitive values are "wrapped" in objects.

```
const prim = true;
assert.equal(typeof prim, 'boolean');
assert.equal(prim instanceof Boolean, false);

const wrapped = Object(prim);
assert.equal(typeof wrapped, 'object');
assert.equal(wrapped instanceof Boolean, true);

assert.equal(wrapped.valueOf(), prim); // unwrap
```

Wrapping rarely matters in practice, but it is used internally in the language specification, to give primitives properties.

11.7 Converting between types

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

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

11.7.1 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:

```
> typeof Object(123)
'object'
```

11.7.2 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.mjs`

Quiz

See quiz app.

Chapter 12

Operators

Contents

12.1 Making sense of operators

JavaScript's operators may seem quirky. With the following two rules, they are easier to understand:

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

12.1.1 Operators coerce their operands to appropriate types

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

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

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

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

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

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

12.1.2 Most operators only work with primitive values

As mentioned before, most operators only work with primitive values. If an operand is an object, it is usually coerced to a primitive value – for example:

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

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

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

Next, it concatenates the two strings:

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

12.2 The plus operator (+)

The plus operator works as follows in JavaScript:

- First, it converts both operands to primitive values. Then it switches to one of two modes:
  - String mode: If one of the two primitive values is a string, then it converts the other one to a string, concatenates both strings, and returns the result.
  - Number mode: Otherwise, It converts both operands to numbers, adds them, and returns the result.

String mode lets us use + to assemble strings:

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

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

```
> 4 + true
5
```

`Number(true)` is 1.

12.3 Assignment operators

12.3.1 The plain assignment operator

The plain assignment operator is used to change storage locations:

```
x = value; // assign to a previously declared variable
obj.propKey = value; // assign to a property
arr[index] = value; // assign to an Array element
```

Initializers in variable declarations can also be viewed as a form of assignment:

```
const x = value;
let y = value;
```

12.3.2 Compound assignment operators

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

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

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

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

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

12.3.3 A list of all compound assignment operators

- Arithmetic operators:

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

  `+=` also works for string concatenation

- Bitwise operators:

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

12.4 Equality: == vs. ===

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

Other names for == and ===

- == is also called *double equals*. Its official name in the language specification is *abstract equality comparison*.
- === is also called *triple equals*.

12.4.1 Loose equality (== and !=)

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

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

Others less so:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

12.4.3 Recommendation: always use strict equality

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

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

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

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

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

I prefer either of the following two alternatives:

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

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

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

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

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

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

I prefer either of the following two alternatives:

```
if (x === undefined || x === null) ···
if (!x) ···
```

A downside of the second alternative is that it accepts values other than `undefined` and `null`, but it is a well-established pattern in JavaScript (to be explained in detail in §14.3 "Truthiness-based existence checks").

The following three conditions are also roughly equivalent:

```
if (x != null) ···
if (x !== undefined && x !== null) ···
if (x) ···
```

12.4.4 Even stricter than ===: `Object.is()`

Method `Object.is()` compares two values:

```
> Object.is(123, 123)
true
> Object.is(123, '123')
false
```

It is even stricter than ===. For example, it considers `NaN`, the error value for computations involving numbers, to be equal to itself:

```
> Object.is(NaN, NaN)
true
> NaN === NaN
false
```

That is occasionally useful. For example, you can use it to implement an improved version of the Array method `.indexOf()`:

```
const myIndexOf = (arr, elem) => {
  return arr.findIndex(x => Object.is(x, elem));
};
```

`myIndexOf()` finds `NaN` in an Array, while `.indexOf()` doesn't:

```
> myIndexOf([0,NaN,2], NaN)
1
> [0,NaN,2].indexOf(NaN)
-1
```

The result `-1` means that `.indexOf()` couldn't find its argument in the Array.

12.5 Ordering operators

Table 12.1: JavaScript's ordering operators.

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

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

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

<= and >= are based on strict equality.

> ⚠ **The ordering operators don't work well for human languages**
>
> The ordering operators don't work well for comparing text in a human language, e.g., when capitalization or accents are involved. The details are explained in §18.5 "Comparing strings".

12.6 Various other operators

Operators for booleans, strings, numbers, objects: are covered elsewhere in this book.

The next two subsections discuss two operators that are rarely used.

12.6.1 Comma operator

The comma operator has two operands, evaluates both of them and returns the second one:

```
> 'a', 'b'
'b'
```

For more information on this operator, see *Speaking JavaScript*.

12.6.2 `void` operator

The `void` operator evaluates its operand and returns `undefined`:

```
> void (3 + 2)
undefined
```

For more information on this operator, see *Speaking JavaScript*.

Quiz

See quiz app.

Part IV

Primitive values

Chapter 13

The non-values `undefined` and `null`

Contents

Many programming languages have one "non-value" called `null`. It indicates that a variable does not currently point to an object – for example, when it hasn't been initialized yet.

In contrast, JavaScript has two of them: `undefined` and `null`.

13.1 `undefined` vs. `null`

Both values are very similar and often used interchangeably. How they differ is therefore subtle. The language itself makes the following distinction:

- `undefined` means "not initialized" (e.g., a variable) or "not existing" (e.g., a property of an object).
- `null` means "the intentional absence of any object value" (a quote from the language specification).

Programmers may make the following distinction:

- `undefined` is the non-value used by the language (when something is uninitialized, etc.).

- `null` means "explicitly switched off". That is, it helps implement a type that comprises both meaningful values and a meta-value that stands for "no meaningful value". Such a type is called *option type* or *maybe type* in functional programming.

13.2 Occurrences of `undefined` and `null`

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

13.2.1 Occurrences of `undefined`

Uninitialized variable `myVar`:

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

Parameter `x` is not provided:

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

Property `.unknownProp` is missing:

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

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

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

13.2.2 Occurrences of `null`

The prototype of an object is either an object or, at the end of a chain of prototypes, `null`. `Object.prototype` does not have a prototype:

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

If you match a regular expression (such as `/a/`) against a string (such as `'x'`), you either get an object with matching data (if matching was successful) or `null` (if matching failed):

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

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

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

13.3 Checking for `undefined` or `null`

Checking for either:

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

Does `x` have a value?

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

Is `x` either `undefined` or `null`?

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

Truthy means "is `true` if coerced to boolean". *Falsy* means "is `false` if coerced to boolean". Both concepts are explained properly in §14.2 "Falsy and truthy values".

13.4 `undefined` and `null` don't have properties

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

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

If we apply `getFoo()` to various values, 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
```

13.5 The history of `undefined` and `null`

In Java (which inspired many aspects of JavaScript), 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`.

In JavaScript, each variable can hold both object values and primitive values. Therefore, if `null` means "not an object", JavaScript also needs an initialization value that means "neither an object nor a primitive value". That initialization value is `undefined`.

Quiz

See quiz app.

Chapter 14

Booleans

Contents

The primitive type *boolean* comprises two values – `false` and `true`:

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

14.1 Converting to boolean

The meaning of "converting to [type]"

"Converting to [type]" is short for "Converting arbitrary values to values of type [type]".

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.

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

Table 14.1: 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`

14.2 Falsy and truthy values

When checking the condition of an `if` statement, a `while` loop, or a `do-while` loop, JavaScript works differently than you may expect. Take, for example, the following condition:

```
if (value) {}
```

In many programming languages, this condition is equivalent to:

```
if (value === true) {}
```

However, in JavaScript, it is equivalent to:

```
if (Boolean(value) === true) {}
```

That is, JavaScript checks if `value` is `true` when converted to boolean. This kind of check is so common that the following 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.

Each value is either truthy or falsy. Consulting tbl. 14.1, we can make an exhaustive list of falsy values:

- `undefined`, `null`
- `false`
- `0`, `NaN`

- `''`

All other values (including all objects) are truthy:

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

14.2.1 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.

> **Exercise: Truthiness**
>
> `exercises/booleans/truthiness_exrc.mjs`

14.3 Truthiness-based existence checks

In JavaScript, if you read something that doesn't exist (e.g., a missing parameter or a missing property), you usually get `undefined` as a result. In these cases, an existence check amounts to comparing a value with `undefined`. For example, the following code checks if object `obj` has the property `.prop`:

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

Due to `undefined` being falsy, we can shorten this check to:

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

```
}
```

14.3.1 Pitfall: truthiness-based existence checks are imprecise

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

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

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

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

However, it is also skipped if:

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

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

14.3.2 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 short. It correctly throws errors for `undefined` and `null`.

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

An alternative is to check for `undefined`:

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

14.3.3 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');
}
```

14.4 Conditional operator (? :)

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

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

It is evaluated as follows:

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

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

Examples:

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

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

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

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

14.5 Binary logical operators: And (x && y), Or (x || y)

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, then 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, `console.log()` is executed:

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

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

14.5.1 Logical And (x && y)

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

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

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

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

Examples:

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

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

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

14.5.2 Logical Or (||)

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

1. Evaluate `a`.
2. Is the result truthy? Return it.

3. Otherwise, evaluate `b` and return the result.

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

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

Examples:

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

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

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

14.5.3 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.mjs`

14.6 Logical Not (!)

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

1. Evaluate `x`.
2. Is it truthy? Return `false`.

3. Otherwise, return `true`.

Examples:

```
> !false
true
> !true
false

> !0
true
> !123
false

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

Quiz

See quiz app.

Chapter 15

Numbers

Contents

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

15.1 JavaScript only has floating point numbers

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

```
98
123.45
```

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

Integers are simply floating point numbers without a decimal fraction:

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

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

15.2 Number literals

Let's examine literals for numbers.

15.2.1 Integer literals

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

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

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

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

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

15.2.2 Floating point literals

Floating point numbers can only be expressed in base 10.

Fractions:

```
> 35.0
35
```

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

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

15.2.3 Syntactic pitfall: properties of integer literals

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

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

There are four ways to work around this pitfall:

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

15.3 Arithmetic operators

15.3.1 Binary arithmetic operators

Tbl. 15.1 lists JavaScript's binary arithmetic operators.

Table 15.1: Binary arithmetic operators.

Operator	Name		Example
`n + m`	Addition	ES1	`3 + 4 → 7`
`n - m`	Subtraction	ES1	`9 - 1 → 8`
`n * m`	Multiplication	ES1	`3 * 2.25 → 6.75`
`n / m`	Division	ES1	`5.625 / 5 → 1.125`
`n % m`	Remainder	ES1	`8 % 5 → 3`
			`-8 % 5 → -3`
`n ** m`	Exponentiation	ES2016	`4 ** 2 → 16`

15.3.1.1 `%` is a remainder operator

`%` is a remainder operator, not a modulo operator. Its result has the sign of the first operand:

```
> 5 % 3
2
> -5 % 3
-2
```

For more information on the difference between remainder and modulo, see the blog post "Remainder operator vs. modulo operator (with JavaScript code)" on 2ality.

15.3.2 Unary plus (+) and negation (-)

Tbl. 15.2 summarizes the two operators *unary plus* (+) and *negation* (-).

Table 15.2: The operators unary plus (+) and negation (-).

Operator	Name		Example
`+n`	Unary plus	ES1	`+(-7)` → `-7`
`-n`	Unary negation	ES1	`-(-7)` → `7`

Both operators coerce their operands to numbers:

```
> +'5'
5
> +'-12'
-12
> -'9'
-9
```

Thus, unary plus lets us convert arbitrary values to numbers.

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

The incrementation operator ++ exists in a prefix version and a suffix version. In both versions, it destructively adds one to its operand. Therefore, its operand must be a storage location that 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 version.

Tbl. 15.3 summarizes the incrementation and decrementation operators.

Table 15.3: Incrementation operators and decrementation operators.

Operator	Name		Example
`v++`	Increment	ES1	`let v=0; [v++, v]` → `[0, 1]`
`++v`	Increment	ES1	`let v=0; [++v, v]` → `[1, 1]`

Operator	Name		Example
`v--`	Decrement	ES1	`let v=1; [v--, v]→[1, 0]`
`--v`	Decrement	ES1	`let v=1; [--v, v]→[0, 0]`

Next, we'll look at examples of these operators in use.

Prefix ++ and prefix `--` change their operands and then return them.

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

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

Suffix ++ and suffix `--` return their operands and then change them.

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

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

15.3.3.1 Operands: not just variables

You can also apply these operators to property values:

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

And to Array elements:

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

Exercise: Number operators

`exercises/numbers-math/is_odd_test.mjs`

15.4 Converting to number

These are three ways of converting values to numbers:

- `Number(value)`

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

Recommendation: use the descriptive `Number()`. Tbl. 15.4 summarizes how it works.

Table 15.4: Converting values to numbers.

`x`	`Number(x)`
`undefined`	`NaN`
`null`	`0`
boolean	`false` → `0`, `true` → `1`
number	`x` (no change)
string	`''` → `0`
	other → parsed number, ignoring leading/trailing whitespace
object	configurable (e.g. via `.valueOf()`)

Examples:

```
assert.equal(Number(123.45), 123.45);

assert.equal(Number(''), 0);
assert.equal(Number('\n 123.45 \t'), 123.45);
assert.equal(Number('xyz'), NaN);
```

How objects are converted to numbers can be configured – for example, by overriding `.valueOf()`:

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

Exercise: Converting to number

`exercises/numbers-math/parse_number_test.mjs`

15.5 Error values

Two number values are returned when errors happen:

- `NaN`
- `Infinity`

15.6 Error value: `NaN`

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

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

When is NaN returned?

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

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

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

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

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

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

15.6.1 Checking for NaN

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

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

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

```
const x = NaN;

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

In the last line, we use the comparison quirk to detect NaN.

15.6.2 Finding NaN in Arrays

Some Array methods can't find NaN:

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

Others can:

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

Alas, there is no simple rule of thumb. You have to check for each method how it handles `NaN`.

15.7 Error value: `Infinity`

When is the error value `Infinity` returned?

Infinity is returned if a number is too large:

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

Infinity is returned if there is a division by zero:

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

15.7.1 `Infinity` as a default value

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

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

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

15.7.2 Checking for `Infinity`

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

```
const x = Infinity;

assert.equal(x === Infinity, true);
assert.equal(Number.isFinite(x), false);
```

Exercise: Comparing numbers

`exercises/numbers-math/find_max_test.mjs`

15.8 The precision of numbers: careful with decimal fractions

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

```
> 0.1 + 0.2
0.30000000000000004
> 1.3 * 3
3.9000000000000004
> 1.4 * 100000000000000
139999999999999.98
```

You therefore need to take rounding errors into consideration when performing arithmetic in JavaScript.

Read on for an explanation of this phenomenon.

Quiz: basic

See quiz app.

15.9 (Advanced)

All remaining sections of this chapter are advanced.

15.10 Background: floating point precision

In JavaScript, computations with numbers don't always produce correct results – for example:

```
> 0.1 + 0.2
0.30000000000000004
```

To understand why, we need to explore how JavaScript represents floating point numbers internally. It uses three integers to do so, which take up a total of 64 bits of storage (double precision):

Component	Size	Integer range
Sign	1 bit	[0, 1]
Fraction	52 bits	$[0, 2^{52}-1]$
Exponent	11 bits	[−1023, 1024]

The floating point number represented by these integers is computed as follows:

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

This representation can't encode a zero because its second component (involving the fraction) always has a leading 1. Therefore, a zero is encoded via the special exponent −1023 and a fraction 0.

15.10.1 A simplified representation of floating point numbers

To make further discussions easier, we simplify the previous representation:

- Instead of base 2 (binary), we use base 10 (decimal) because that's what most people are more familiar with.
- The *fraction* is a natural number that is interpreted as a fraction (digits after a point). We switch to a *mantissa*, an integer that is interpreted as itself. As a consequence, the exponent is used differently, but its fundamental role doesn't change.
- As the mantissa is an integer (with its own sign), we don't need a separate sign, anymore.

The new representation works like this:

$$\text{mantissa} \times 10^{\text{exponent}}$$

Let's try out this representation for a few floating point numbers.

- For the integer −123, we mainly need the mantissa:

```
> -123 * (10 ** 0)
-123
```

- For the number 1.5, we imagine there being a point after the mantissa. We use a negative exponent to move that point one digit to the left:

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

- For the number 0.25, we move the point two digits to the left:

```
> 25 * (10 ** -2)
0.25
```

Representations with negative exponents can also be written as fractions with positive exponents in the denominators:

```
> 15 * (10 ** -1) === 15 / (10 ** 1)
true
> 25 * (10 ** -2) === 25 / (10 ** 2)
true
```

These fractions help with understanding why there are numbers that our encoding cannot represent:

- `1/10` can be represented. It already has the required format: a power of 10 in the denominator.
- `1/2` can be represented as `5/10`. We turned the 2 in the denominator into a power of 10 by multiplying the numerator and denominator by 5.

- `1/4` can be represented as `25/100`. We turned the 4 in the denominator into a power of 10 by multiplying the numerator and denominator by 25.
- `1/3` cannot be represented. There is no way to turn the denominator into a power of 10. (The prime factors of 10 are 2 and 5. Therefore, any denominator that only has these prime factors can be converted to a power of 10, by multiplying both the numerator and denominator by enough twos and fives. If a denominator has a different prime factor, then there's nothing we can do.)

To conclude our excursion, we switch back to base 2:

- `0.5 = 1/2` can be represented with base 2 because the denominator is already a power of 2.
- `0.25 = 1/4` can be represented with base 2 because the denominator is already a power of 2.
- `0.1 = 1/10` cannot be represented because the denominator cannot be converted to a power of 2.
- `0.2 = 2/10` cannot be represented because the denominator cannot be converted to a power of 2.

Now we can see why `0.1 + 0.2` doesn't produce a correct result: internally, neither of the two operands can be represented precisely.

The only way to compute precisely with decimal fractions is by internally switching to base 10. For many programming languages, base 2 is the default and base 10 an option. For example, Java has the class `BigDecimal` and Python has the module `decimal`. There are tentative plans to add something similar to JavaScript: the ECMAScript proposal "Decimal" is currently at stage 0.

15.11 Integers in JavaScript

JavaScript doesn't have a special type for integers. Instead, they are simply normal (floating point) numbers without a decimal fraction:

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

In this section, we'll look at a few tools for working with these pseudo-integers.

15.11.1 Converting to integer

The recommended way of converting numbers to integers is to use one of the rounding methods of the `Math` object:

- `Math.floor(n)`: returns the largest integer `i` ≤ `n`

  ```
  > Math.floor(2.1)
  2
  > Math.floor(2.9)
  2
  ```

- `Math.ceil(n)`: returns the smallest integer `i` ≥ `n`

```
> Math.ceil(2.1)
3
> Math.ceil(2.9)
3
```

- `Math.round(n)`: returns the integer that is "closest" to `n` with `__.5` being rounded up – for example:

```
> Math.round(2.4)
2
> Math.round(2.5)
3
```

- `Math.trunc(n)`: removes any decimal fraction (after the point) that `n` has, therefore turning it into an integer.

```
> Math.trunc(2.1)
2
> Math.trunc(2.9)
2
```

For more information on rounding, consult §16.3 "Rounding".

15.11.2 Ranges of integers in JavaScript

These are important ranges of integers in JavaScript:

- **Safe integers:** can be represented "safely" by JavaScript (more on what that means in the next subsection)
  - 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 operators** (bitwise Or, etc.)
  - Precision: 32 bits
  - Range of unsigned right shift (`>>>`): unsigned, $[0, 2^{32})$
  - Range of all other bitwise operators: signed, $[-2^{31}, 2^{31})$

15.11.3 Safe integers

This is the range of integers that are *safe* in JavaScript (53 bits plus a sign):

$$[-2^{53}-1, 2^{53}-1]$$

An integer is *safe* if it is represented by exactly one JavaScript number. Given that JavaScript numbers are encoded as a fraction multiplied by 2 to the power of an exponent, higher integers can also be represented, but then there are gaps between them.

For example (18014398509481984 is 2^{54}):

```
> 18014398509481984
18014398509481984
> 18014398509481985
18014398509481984
> 18014398509481986
18014398509481984
> 18014398509481987
18014398509481988
```

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

```
assert.equal(Number.MAX_SAFE_INTEGER, (2 ** 53) - 1);
assert.equal(Number.MIN_SAFE_INTEGER, -Number.MAX_SAFE_INTEGER);

assert.equal(Number.isSafeInteger(5), true);
assert.equal(Number.isSafeInteger('5'), false);
assert.equal(Number.isSafeInteger(5.1), false);
assert.equal(Number.isSafeInteger(Number.MAX_SAFE_INTEGER), true);
assert.equal(Number.isSafeInteger(Number.MAX_SAFE_INTEGER+1), false);
```

Exercise: Detecting safe integers

`exercises/numbers-math/is_safe_integer_test.mjs`

15.11.3.1 Safe computations

Let's look at computations involving unsafe integers.

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

```
> 9007199254740990 + 3
9007199254740992
```

The following result is safe, but incorrect. The first operand is unsafe; the second operand is safe:

```
> 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)
```

That is, both operands and the result must be safe.

15.12 Bitwise operators

15.12.1 Internally, bitwise operators work with 32-bit integers

Internally, JavaScript's bitwise operators work with 32-bit integers. They produce their results in the following steps:

- Input (JavaScript numbers): The 1–2 operands are first converted to JavaScript numbers (64-bit floating point numbers) and then to 32-bit integers.
- Computation (32-bit integers): The actual operation processes 32-bit integers and produces a 32-bit integer.
- Output (JavaScript number): Before returning the result, it is converted back to a JavaScript number.

15.12.1.1 The types of operands and results

For each bitwise operator, this book mentions the types of its operands and its result. Each type is always one of the following two:

Type	Description	Size	Range
Int32	signed 32-bit integer	32 bits incl. sign	$[-2^{31}, 2^{31})$
Uint32	unsigned 32-bit integer	32 bits	$[0, 2^{32})$

Considering the previously mentioned steps, I recommend to pretend that bitwise operators internally work with unsigned 32-bit integers (step "computation") and that Int32 and Uint32 only affect how JavaScript numbers are converted to and from integers (steps "input" and "output").

15.12.1.2 Displaying JavaScript numbers as unsigned 32-bit integers

While exploring the bitwise operators, it occasionally helps to display JavaScript numbers as unsigned 32-bit integers in binary notation. That's what `b32()` does (whose implementation is shown later):

```
assert.equal(
  b32(-1),
  '11111111111111111111111111111111');
assert.equal(
  b32(1),
  '00000000000000000000000000000001');
assert.equal(
  b32(2 ** 31),
  '10000000000000000000000000000000');
```

15.12.2 Binary bitwise operators

Table 15.7: Binary bitwise operators.

Operation	Name	Type signature	
`num1 & num2`	Bitwise And	Int32 × Int32 → Int32	ES1
`num1 | num2`	Bitwise Or	Int32 × Int32 → Int32	ES1
`num1 ^ num2`	Bitwise Xor	Int32 × Int32 → Int32	ES1

The binary bitwise operators (tbl. 15.7) combine the bits of their operands to produce their results:

```
> (0b1010 & 0b0011).toString(2).padStart(4, '0')
'0010'
> (0b1010 | 0b0011).toString(2).padStart(4, '0')
'1011'
> (0b1010 ^ 0b0011).toString(2).padStart(4, '0')
'1001'
```

15.12.3 Bitwise Not

Table 15.8: The bitwise Not operator.

Operation	Name	Type signature	
`~num`	Bitwise Not, ones' complement	Int32 → Int32	ES1

The bitwise Not operator (tbl. 15.8) inverts each binary digit of its operand:

```
> b32(~0b100)
'11111111111111111111111111111011'
```

15.12.4 Bitwise shift operators

Table 15.9: Bitwise shift operators.

Operation	Name	Type signature	
`num << count`	Left shift	Int32 × Uint32 → Int32	ES1
`num >> count`	Signed right shift	Int32 × Uint32 → Int32	ES1
`num >>> count`	Unsigned right shift	Uint32 × Uint32 → Uint32	ES1

The shift operators (tbl. 15.9) move binary digits to the left or to the right:

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

`>>` preserves highest bit, `>>>` doesn't:

```
> b32(0b10000000000000000000000000000010 >> 1)
'11000000000000000000000000000001'
> b32(0b10000000000000000000000000000010 >>> 1)
'01000000000000000000000000000001'
```

15.12.5 `b32()`: displaying unsigned 32-bit integers in binary notation

We have now used `b32()` a few times. The following code is an implementation of it:

```
/**
 * Return a string representing n as a 32-bit unsigned integer,
 * in binary notation.
 */
function b32(n) {
  // >>> ensures highest bit isn't interpreted as a sign
  return (n >>> 0).toString(2).padStart(32, '0');
}
assert.equal(
  b32(6),
  '00000000000000000000000000000110');
```

`n >>> 0` means that we are shifting `n` zero bits to the right. Therefore, in principle, the `>>>` operator does nothing, but it still coerces `n` to an unsigned 32-bit integer:

```
> 12 >>> 0
12
> -12 >>> 0
4294967284
> (2**32 + 1) >>> 0
1
```

15.13 Quick reference: numbers

15.13.1 Global functions for numbers

JavaScript has the following four global functions for numbers:

- `isFinite()`
- `isNaN()`
- `parseFloat()`
- `parseInt()`

However, it is better to use the corresponding methods of `Number` (`Number.isFinite()`, etc.), which have fewer pitfalls. They were introduced with ES6 and are discussed below.

15.13.2 Static properties of `Number`

- `.EPSILON: number` [ES6]

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

  - Approximately: $2.220446049250313080847263336181 6 \times 10^{-16}$

- `.MAX_SAFE_INTEGER: number` [ES6]

  The largest integer that JavaScript can represent unambiguously ($2^{53}–1$).

- `.MAX_VALUE: number` [ES1]

  The largest positive finite JavaScript number.

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

- `.MIN_SAFE_INTEGER: number` [ES6]

  The smallest integer that JavaScript can represent unambiguously ($-2^{53}+1$).

- `.MIN_VALUE: number` [ES1]

  The smallest positive JavaScript number. Approximately 5×10^{-324}.

- `.NaN: number` [ES1]

  The same as the global variable `NaN`.

- `.NEGATIVE_INFINITY: number` [ES1]

  The same as `-Number.POSITIVE_INFINITY`.

- `.POSITIVE_INFINITY: number` [ES1]

  The same as the global variable `Infinity`.

15.13.3 Static methods of `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
```

- `.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
```

- `.isNaN(num: number): boolean` [ES6]

  Returns `true` if `num` is the value `NaN`:

```
> Number.isNaN(NaN)
true
> Number.isNaN(123)
false
> Number.isNaN('abc')
false
```

- `.isSafeInteger(num: number): boolean` [ES6]

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

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

  Coerces its parameter to string and parses it as a floating point number. 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 illegal trailing characters and can hide problems).

  ```
  > Number.parseFloat(' 123.4#')
  123.4
  > Number(' 123.4#')
  NaN
  ```

- `.parseInt(str: string, radix=10): number` [ES6]

  Coerces its parameter to string and parses it as an integer, ignoring leading whitespace and illegal trailing characters:

  ```
  > Number.parseInt('  123#')
  123
  ```

  The parameter `radix` specifies the base of the number to be parsed:

  ```
  > Number.parseInt('101', 2)
  5
  > Number.parseInt('FF', 16)
  255
  ```

  Do not use this method to convert numbers to integers: coercing to string is inefficient. And stopping before the first non-digit is not a good algorithm for removing the fraction of a number. Here is an example where it goes wrong:

  ```
  > Number.parseInt(1e21, 10) // wrong
  1
  ```

  It is better to use one of the rounding functions of `Math` to convert a number to an integer:

  ```
  > Math.trunc(1e21) // correct
  1e+21
  ```

15.13.4 Methods of `Number.prototype`

(`Number.prototype` is where the methods of numbers are stored.)

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

  Returns a string that represents the number via exponential notation. With `fractionDigits`, you can specify, how many digits should be shown of the number that is multiplied with the exponent (the default is to show as many digits as necessary).

  Example: number too small to get a positive exponent via `.toString()`.

  ```
  > 1234..toString()
  '1234'

  > 1234..toExponential() // 3 fraction digits
  '1.234e+3'
  > 1234..toExponential(5)
  '1.23400e+3'
  > 1234..toExponential(1)
  '1.2e+3'
  ```

  Example: fraction not small enough to get a negative exponent via `.toString()`.

  ```
  > 0.003.toString()
  '0.003'
  > 0.003.toExponential()
  '3e-3'
  ```

- `.toFixed(fractionDigits=0): string` [ES3]

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

  ```
  > 0.00000012.toString() // with exponent
  '1.2e-7'

  > 0.00000012.toFixed(10) // no exponent
  '0.0000001200'
  > 0.00000012.toFixed()
  '0'
  ```

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

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

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

  Works like `.toString()`, but `precision` specifies how many digits should be shown. 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 number.

  By default, you get a base 10 numeral as a result:

  ```
  > 123.456.toString()
  '123.456'
  ```

  If you want the numeral to have a different base, you can specify it via `radix`:

  ```
  > 4..toString(2) // binary (base 2)
  '100'
  > 4.5.toString(2)
  '100.1'

  > 255..toString(16) // hexadecimal (base 16)
  'ff'
  > 255.66796875.toString(16)
  'ff.ab'

  > 1234567890..toString(36)
  'kf12oi'
  ```

  `parseInt()` provides the inverse operation: it converts a string that contains an integer (no fraction!) numeral with a given base, to a number.

  ```
  > parseInt('kf12oi', 36)
  1234567890
  ```

15.13.5 Sources

- Wikipedia
- TypeScript's built-in typings
- MDN web docs for JavaScript
- ECMAScript language specification

Quiz: advanced

See quiz app.

Chapter 16

Math

Contents

`Math` is an object with data properties and methods for processing numbers. You can see it as a poor man's module: It was created long before JavaScript had modules.

16.1 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.

16.2 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. Therefore, this function returns more precise values whenever `.exp()` returns values 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. Therefore, you can provide this function with a more precise argument whenever the argument for `.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
  ```

16.3 Rounding

Rounding means converting an arbitrary number to an integer (a number without a decimal fraction). The following functions implement different approaches to rounding.

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

  Returns the smallest (closest to $-\infty$) integer `i` with $\texttt{x} \leq \texttt{i}$.

  ```
  > Math.ceil(2.1)
  3
  > Math.ceil(2.9)
  3
  ```

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

  Returns the largest (closest to $+\infty$) integer `i` with $\texttt{i} \leq \texttt{x}$.

```
> Math.floor(2.1)
2
> Math.floor(2.9)
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(2.4)
  2
  > Math.round(2.5)
  3
  ```

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

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

  ```
  > Math.trunc(2.1)
  2
  > Math.trunc(2.9)
  2
  ```

Tbl. 16.1 shows the results of the rounding functions for a few representative inputs.

Table 16.1: Rounding functions of `Math`. Note how things change with negative numbers because "larger" always means "closer to positive infinity".

	-2.9	-2.5	-2.1	2.1	2.5	2.9
Math.floor	-3	-3	-3	2	2	2
Math.ceil	-2	-2	-2	3	3	3
Math.round	-3	-2	-2	2	3	3
Math.trunc	-2	-2	-2	2	2	2

16.4 Trigonometric Functions

All angles are specified in radians. Use the following two functions to convert between degrees and radians.

```
function degreesToRadians(degrees) {
  return degrees / 180 * Math.PI;
}
assert.equal(degreesToRadians(90), Math.PI/2);

function radiansToDegrees(radians) {
  return radians / Math.PI * 180;
}
assert.equal(radiansToDegrees(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`.

16.5 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(0b01000000000000000000000000000000)
  1
  > Math.clz32(0b00100000000000000000000000000000)
  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 $0 \leq i < max$:

  ```
  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
  ```

16.6 Sources

- Wikipedia
- TypeScript's built-in typings
- MDN web docs for JavaScript
- ECMAScript language specification

Chapter 17

Unicode – a brief introduction (advanced)

Contents

Unicode is a standard for representing and managing text in most of the world's writing systems. Virtually all modern software that works with text, supports Unicode. The standard is maintained by the Unicode Consortium. A new version of the standard is published every year (with new emojis, etc.). Unicode version 1.0.0 was published in October 1991.

17.1 Code points vs. code units

Two concepts are crucial for understanding Unicode:

- *Code points* are numbers that represent Unicode characters.
- *Code units* are numbers that encode code points, to store or transmit Unicode text. One or more code units encode a single code point. Each code unit has the same size, which depends on the *encoding format* that is used. The most popular format, UTF-8, has 8-bit code units.

17.1.1 Code points

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

- Plane 0: **Basic Multilingual Plane (BMP)**, 0x0000–0xFFFF
  - Contains characters for almost all modern languages (Latin characters, Asian characters, etc.) and many symbols.
- Plane 1: Supplementary Multilingual Plane (SMP), 0x10000–0x1FFFF
  - Supports historic writing systems (e.g., Egyptian hieroglyphs and cuneiform) and additional modern writing systems.
  - Supports emojis and many other symbols.
- Plane 2: Supplementary Ideographic Plane (SIP), 0x20000–0x2FFFF
  - Contains additional CJK (Chinese, Japanese, Korean) ideographs.
- Plane 3–13: Unassigned
- Plane 14: Supplementary Special-Purpose Plane (SSP), 0xE0000–0xEFFFF
  - Contains non-graphical characters such as tag characters and glyph variation selectors.
- Plane 15–16: Supplementary Private Use Area (S PUA A/B), 0x0F0000–0x10FFFF
  - Available for character assignment by parties outside the ISO and the Unicode Consortium. Not standardized.

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

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

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

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

17.1.2 Encoding Unicode code points: UTF-32, UTF-16, UTF-8

The main ways of encoding code points are three *Unicode Transformation Formats* (UTFs): UTF-32, UTF-16, UTF-8. The number at the end of each format indicates the size (in bits) of its code units.

17.1.2.1 UTF-32 (Unicode Transformation Format 32)

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

17.1.2.2 UTF-16 (Unicode Transformation Format 16)

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

- The BMP (first 16 bits of Unicode) is stored in single code units.
- Astral planes: The BMP comprises 0x10_000 code points. Given that Unicode has a total of 0x110_000 code points, we still need to encode the remaining 0x100_000 code points (20 bits). The BMP has two ranges of unassigned code points that provide the necessary storage:
  - Most significant 10 bits (*leading surrogate*): 0xD800-0xDBFF
  - Least significant 10 bits (*trailing surrogate*): 0xDC00-0xDFFF

In other words, the two hexadecimal digits at the end contribute 8 bits. But we can only use those 8 bits if a BMP starts with one of the following 2-digit pairs:

- D8, D9, DA, DB
- DC, DD, DE, DF

Per surrogate, we have a choice between 4 pairs, which is where the remaining 2 bits come from.

As a consequence, each UTF-16 code unit is always either a leading surrogate, a trailing surrogate, or encodes a BMP code point.

These are two examples of UTF-16-encoded code points:

- Code point 0x03C0 (π) is in the BMP and can therefore be represented by a single UTF-16 code unit: 0x03C0.
- Code point 0x1F642 (🙂) is in an astral plane and represented by two code units: 0xD83D and 0xDE42.

17.1.2.3 UTF-8 (Unicode Transformation Format 8)

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

Code points	Code units
0000–007F	0bbbbbbb (7 bits)
0080–07FF	110bbbbb, 10bbbbbb (5+6 bits)
0800–FFFF	1110bbbb, 10bbbbbb, 10bbbbbb (4+6+6 bits)
10000–1FFFFF	11110bbb, 10bbbbbb, 10bbbbbb, 10bbbbbb (3+6+6+6 bits)

Notes:

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

Three examples:

Character	Code point	Code units
A	0x0041	01000001
π	0x03C0	11001111, 10000000
☺	0x1F642	11110000, 10011111, 10011001, 10000010

17.2 Encodings used in web development: UTF-16 and UTF-8

The Unicode encoding formats that are used in web development are: UTF-16 and UTF-8.

17.2.1 Source code internally: UTF-16

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

17.2.2 Strings: UTF-16

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

```
> const smiley = '☺';
> smiley.length
2
> smiley === '\uD83D\uDE42' // code units
true
```

For more information on Unicode and strings, consult §18.6 "Atoms of text: Unicode characters, JavaScript characters, grapheme clusters".

17.2.3 Source code in files: UTF-8

HTML and JavaScript are almost always encoded as UTF-8 these days.

For example, this is how HTML files usually start now:

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

For HTML modules loaded in web browsers, the standard encoding is also UTF-8.

17.3 Grapheme clusters – the real characters

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

On one hand, there are Unicode characters, as represented by code points.

On the other hand, there are *grapheme clusters*. A grapheme cluster corresponds most closely to a symbol displayed on screen or paper. It is defined as "a horizontally segmentable unit of text". Therefore, official Unicode documents also call it a *user-perceived character*. One or more code point characters are needed to encode a grapheme cluster.

For example, the Devanagari *kshi* is encoded by 4 code points. We use spreading (`...`) to split a string into an Array with code point characters (for details, consult §18.6.1 "Working with code points"):

```
> [...'क्षि']
[ 'क', '्', 'ष', 'ि' ]
```

Flag emojis are also grapheme clusters and composed of two code point characters – for example, the flag of Japan:

```
> [...'🇯🇵']
[ '🇯', '🇵' ]
```

More information on grapheme clusters

For more information, consult "Let's Stop Ascribing Meaning to Code Points" by Manish Goregaokar.

Quiz

See quiz app.

Chapter 18

Strings

Contents

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

18.1 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, where double quotes are preferred.

The next chapter covers *template literals*, which give you:

- String interpolation
- Multiple lines
- Raw string literals (backslash has no special meaning)

18.1.1 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:

```
assert.equal(
  'She said: "Let\'s go!"',
  "She said: \"Let's go!\"");
```

18.2 Accessing characters and code points

18.2.1 Accessing JavaScript characters

JavaScript has no extra data type for characters – characters are always represented 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);
```

18.2.2 Accessing Unicode code point characters via `for-of` and spreading

Iterating over strings via `for-of` or spreading (`...`) visits Unicode code point characters. Each code point character is encoded by 1–2 JavaScript characters. For more information,

see §18.6 "Atoms of text: Unicode characters, JavaScript characters, grapheme clusters".

This is how you iterate over the code point characters of a string via `for-of`:

```
for (const ch of 'x😊y') {
  console.log(ch);
}
// Output:
// 'x'
// '😊'
// 'y'
```

And this is how you convert a string into an Array of code point characters via spreading:

```
assert.deepEqual([...'x😊y'], ['x', '😊', 'y']);
```

18.3 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');
```

Concatenating via + is efficient

Using + to assemble strings is quite efficient because most JavaScript engines internally optimize it.

Exercise: Concatenating strings

`exercises/strings/concat_string_array_test.mjs`

18.4 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 generally can't convert it back via `Boolean()`:

```
> String(false)
'false'
> Boolean('false')
true
```

The only string for which `Boolean()` returns `false`, is the empty string.

18.4.1 Stringifying objects

Plain objects have a default string 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}'
```

18.4.2 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');
```

18.4.3 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 convert values to strings:

```
> 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 multiline 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"
}
```

18.5 Comparing strings

Strings can be compared via the following operators:

```
< <= > >=
```

There is one important caveat to consider: These operators compare based on the numeric values of JavaScript characters. That means that the order that JavaScript uses for strings is different from the one used in dictionaries and phone books:

```
> 'A' < 'B' // ok
true
> 'a' < 'B' // not ok
false
```

```
> 'ä' < 'b' // not ok
false
```

Properly comparing text is beyond the scope of this book. It is supported via the ECMAScript Internationalization API (`Intl`).

18.6 Atoms of text: Unicode characters, JavaScript characters, grapheme clusters

Quick recap of §17 "Unicode – a brief introduction":

- Unicode characters are represented by *code points* – numbers which have a range of 21 bits.
- In JavaScript strings, Unicode is implemented via *code units* based on the encoding format UTF-16. Each code unit is a 16-bit number. One to two of code units are needed to encode a single code point.
  - Therefore, each JavaScript character is represented by a code unit. In the JavaScript standard library, code units are also called *char codes*. Which is what they are: numbers for JavaScript characters.
- *Grapheme clusters* (*user-perceived characters*) are written symbols, as displayed on screen or paper. One or more Unicode characters are needed to encode a single grapheme cluster.

The following code demonstrates that a single Unicode character comprises one or two JavaScript characters. We count the latter via `.length`:

```
// 3 Unicode characters, 3 JavaScript characters:
assert.equal('abc'.length, 3);

// 1 Unicode character, 2 JavaScript characters:
assert.equal('😀'.length, 2);
```

The following table summarizes the concepts we have just explored:

Entity	Numeric representation	Size	Encoded via
Grapheme cluster			1+ code points
Unicode character	Code point	21 bits	1–2 code units
JavaScript character	UTF-16 code unit	16 bits	–

18.6.1 Working with code points

Let's explore JavaScript's tools for working with code points.

A *code point escape* lets you specify a code point hexadecimally. It produces one or two JavaScript characters.

```
> '\u{1F642}'
'😂'
```

`String.fromCodePoint()` converts a single code point to 1–2 JavaScript characters:

```
> String.fromCodePoint(0x1F642)
'☺'
```

`.codePointAt()` converts 1–2 JavaScript characters to a single code point:

```
> '☺'.codePointAt(0).toString(16)
'1f642'
```

You can *iterate* over a string, which visits Unicode characters (not JavaScript characters). Iteration is described later in this book. One way of iterating is via a `for-of` loop:

```
const str = '☺a';
assert.equal(str.length, 3);

for (const codePointChar of str) {
  console.log(codePointChar);
}

// Output:
// '☺'
// 'a'
```

Spreading (`...`) into Array literals is also based on iteration and visits Unicode characters:

```
> [...'☺a']
[ '☺', 'a' ]
```

That makes it a good tool for counting Unicode characters:

```
> [...'☺a'].length
2
> '☺a'.length
3
```

18.6.2 Working with code units (char codes)

Indices and lengths of strings are based on JavaScript characters (as represented by UTF-16 code units).

To specify a code unit hexadecimally, you can use a *code unit escape*:

```
> '\uD83D\uDE42'
'☺'
```

And you can use `String.fromCharCode()`. *Char code* is the standard library's name for *code unit*:

```
> String.fromCharCode(0xD83D) + String.fromCharCode(0xDE42)
'☺'
```

To get the char code of a character, use `.charCodeAt()`:

```
> '☺'.charCodeAt(0).toString(16)
'd83d'
```

18.6.3 Caveat: grapheme clusters

When working with text that may be written in any human language, it's best to split at the boundaries of grapheme clusters, not at the boundaries of Unicode characters.

TC39 is working on `Intl.Segmenter`, a proposal for the ECMAScript Internationalization API to support Unicode segmentation (along grapheme cluster boundaries, word boundaries, sentence boundaries, etc.).

Until that proposal becomes a standard, you can use one of several libraries that are available (do a web search for "JavaScript grapheme").

18.7 Quick reference: Strings

Strings are immutable; none of the string methods ever modify their strings.

18.7.1 Converting to string

Tbl. 18.2 describes how various values are converted to strings.

Table 18.2: 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()`

18.7.2 Numeric values of characters

- **Char code:** represents a JavaScript character numerically. JavaScript's name for *Unicode code unit*.
  - Size: 16 bits, unsigned
  - Convert number to character: `String.fromCharCode()` [ES1]
  - Convert character to number: string method `.charCodeAt()` [ES1]
- **Code point:** represents a Unicode character numerically.
  - Size: 21 bits, unsigned (17 planes, 16 bits each)
  - Convert number to character: `String.fromCodePoint()` [ES6]
  - Convert character to number: string method `.codePointAt()` [ES6]

18.7.3 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');
```

18.7.4 `String.prototype`: finding and matching

(`String.prototype` is where the methods of strings are stored.)

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

  Returns `true` if the string would end with `searchString` if its length were `endPos`. Returns `false` otherwise.

  ```
  > 'foo.txt'.endsWith('.txt')
  true
  > 'abcde'.endsWith('cd', 4)
  true
  ```

- `.includes(searchString: string, startPos=0): boolean` [ES6]

  Returns `true` if the string contains the `searchString` and `false` otherwise. The search starts at `startPos`.

  ```
  > 'abc'.includes('b')
  true
  > 'abc'.includes('b', 2)
  false
  ```

- `.indexOf(searchString: string, minIndex=0): number` [ES1]

  Returns the lowest index at which `searchString` appears within the string or `-1`, otherwise. Any returned index will beminIndex‘ 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 string or `-1`, otherwise. Any returned index will bemaxIndex‘ or lower.

  ```
  > 'abab'.lastIndexOf('ab', 2)
  2
  > 'abab'.lastIndexOf('ab', 1)
  0
  > 'abab'.lastIndexOf('ab')
  2
  ```

- [1 of 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 string. Or `null` if there is no match. If `regExp` is a string, it is used to create a regular expression (think parameter of `new RegExp()`) before performing the previously mentioned 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 (which is why this type extends `Array`). Named capture groups (ES2018) become properties of `.groups`. In this mode, `.match()` works like `RegExp.prototype.exec()`.

Examples:

```
> 'ababb'.match(/a(b+)/)
{ 0: 'ab', 1: 'b', index: 0, input: 'ababb', groups: undefined }
> 'ababb'.match(/a(?<foo>b+)/)
{ 0: 'ab', 1: 'b', index: 0, input: 'ababb', groups: { foo: 'b' } }
> 'abab'.match(/x/)
null
```

- [2 of 2] `.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 string. If `regExp` is a string, it is used to create a regular expression (think parameter of `new RegExp()`).

  ```
  > 'a2b'.search(/[0-9]/)
  1
  > 'a2b'.search('[0-9]')
  1
  ```

- `.startsWith(searchString: string, startPos=0): boolean` [ES6]

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

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

18.7.5 `String.prototype`: extracting

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

  Returns the substring of the string that starts at (including) index `start` and ends at (excluding) index `end`. If an index is negative, it is added to `.length` before it is used (`-1` becomes `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 string into an Array of substrings – the strings that occur between the separators. The separator can be a string:

  ```
  > 'a | b | c'.split('|')
  [ 'a ', ' b ', ' c' ]
  ```

  It can also be a regular expression:

  ```
  > 'a : b : c'.split(/ *: */)
  [ 'a', 'b', 'c' ]
  > 'a : b : c'.split(/( *):( *)/)
  [ 'a', ' ', ' ', 'b', ' ', ' ', 'c' ]
  ```

  The last invocation demonstrates that captures made by groups in the regular expression become elements of the returned Array.

  Warning: `.split('')` splits a string into JavaScript characters. That doesn't work well when dealing with astral Unicode characters (which are encoded as two JavaScript characters). For example, emojis are astral:

  ```
  > '😲X😲'.split('')
  [ '\uD83D', '\uDE42', 'X', '\uD83D', '\uDE42' ]
  ```

  Instead, it is better to use spreading:

  ```
  > [...'😲X😲']
  [ '😲', 'X', '😲' ]
  ```

- `.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.

18.7.6 `String.prototype`: combining

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

  Returns the concatenation of the string and `strings`. `'a'.concat('b')` is equivalent to `'a'+'b'`. The latter is much more popular.

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

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

  Appends (fragments of) `fillString` to the string until it has the desired length `len`. If it already has or exceeds `len`, then it is returned without any changes.

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

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

  Prepends (fragments of) `fillString` to the string until it has the desired length `len`. If it already has or exceeds `len`, then it is returned without any changes.

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

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

  Returns the string, concatenated `count` times.

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

18.7.7 `String.prototype`: transforming

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

  Normalizes the string according to the Unicode Normalization Forms.

- [1 of 2] `.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` stands for the string `'$0'`, it does not refer to the complete match)
- `$&`: becomes the complete match
- `` $` ``: becomes everything before the match
- `$'`: becomes everything after the match

Examples:

```
> 'a 2020-04 b'.replace(/([0-9]{4})-([0-9]{2})/, '|$2|')
'a |04| b'
> 'a 2020-04 b'.replace(/([0-9]{4})-([0-9]{2})/, '|$&|')
'a |2020-04| b'
> 'a 2020-04 b'.replace(/([0-9]{4})-([0-9]{2})/, '|$`|')
'a |a | b'
```

Named capture groups (ES2018) are supported, too:

- `$<name>` becomes the capture of named group `name`

Example:

```
assert.equal(
  'a 2020-04 b'.replace(
    /(?<year>[0-9]{4})-(?<month>[0-9]{2})/, '|$<month>|'),
  'a |04| b');
```

- [2 of 2] `.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 2020-04 b'.replace(regexp, replacer),
    'a |2020-04| b');
  ```

Named capture groups (ES2018) are supported, too. If there are any, an argument is added at the end with an object whose properties contain the captures:

```
const regexp = /(?<year>[0-9]{4})-(?<month>[0-9]{2})/;
const replacer = (...args) => {
  const groups=args.pop();
  return '|' + groups.month + '|';
};
assert.equal(
  'a 2020-04 b'.replace(regexp, replacer),
  'a |04| b');
```

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

  Returns a copy of the string in which all lowercase alphabetic characters are converted to uppercase. How well that works for various alphabets, depends on the JavaScript engine.

  ```
  > '-a2b-'.toUpperCase()
  '-A2B-'
  > 'αβγ'.toUpperCase()
  'ΑΒΓ'
  ```

- `.toLowerCase(): string` [ES1]

  Returns a copy of the string in which all uppercase alphabetic characters are converted to lowercase. How well that works for various alphabets, depends on the JavaScript engine.

  ```
  > '-A2B-'.toLowerCase()
  '-a2b-'
  > 'ΑΒΓ'.toLowerCase()
  'αβγ'
  ```

- `.trim(): string` [ES5]

  Returns a copy of the string in which all leading and trailing whitespace (spaces, tabs, line terminators, etc.) is gone.

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

- `.trimEnd(): string` [ES2019]

  Similar to `.trim()` but only the end of the string is trimmed:

  ```
  > '  abc  '.trimEnd()
  '  abc'
  ```

- `.trimStart(): string` [ES2019]

  Similar to `.trim()` but only the beginning of the string is trimmed:

```
> '  abc  '.trimStart()
'abc  '
```

18.7.8 Sources

- TypeScript's built-in typings
- MDN web docs for JavaScript
- ECMAScript language specification

Exercise: Using string methods

`exercises/strings/remove_extension_test.mjs`

Quiz

See quiz app.

Chapter 19

Using template literals and tagged templates

Contents

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

19.1 Disambiguation: "template"

The following three things are significantly different despite all having *template* in their names and despite all of them looking similar:

- A *text template* is a function from data to text. It is frequently used in web development and often defined via text files. For example, the following text defines a template for the library Handlebars:

```
<div class="entry">
  <h1>{{title}}</h1>
  <div class="body">
    {{body}}
  </div>
</div>
```

This template has two blanks to be filled in: `title` and `body`. It is used like this:

```
// First step: retrieve the template text, e.g. from a text file.
const tmplFunc = Handlebars.compile(TMPL_TEXT); // compile string
const data = {title: 'My page', body: 'Welcome to my page!'};
const html = tmplFunc(data);
```

- A *template literal* is similar to a string literal, but has additional features – for example, interpolation. It is delimited by backticks:

  ```
  const num = 5;
  assert.equal(`Count: ${num}!`, 'Count: 5!');
  ```

- Syntactically, a *tagged template* is a template literal that follows a function (or rather, an expression that evaluates to a function). That leads to the function being called. Its arguments are derived from the contents of the template literal.

  ```
  const getArgs = (...args) => args;
  assert.deepEqual(
    getArgs`Count: ${5}!`,
    [['Count: ', '!'], 5] );
  ```

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

19.2 Template literals

A template literal has two new features compared to a normal string literal.

First, it supports *string interpolation*: if you put a dynamically computed value inside a `${}`, it is converted to a string and inserted into the string returned by the literal.

```
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.

19.3 Tagged templates

The expression in line A is a *tagged template*. It is equivalent to invoking `tagFunc()` with the arguments listed in the Array in line B.

```
function tagFunc(...args) {
  return args;
}

const setting = 'dark mode';
const value = true;

assert.deepEqual(
  tagFunc`Setting ${setting} is ${value}!`, // (A)
  [['Setting ', ' is ', '!'], 'dark mode', true] // (B)
);
```

The function `tagFunc` before the first backtick is called a *tag function*. Its arguments are:

- *Template strings* (first argument): an Array with the text fragments surrounding the interpolations `${}`.
  - In the example: `['Setting ', ' is ', '!']`
- *Substitutions* (remaining arguments): the interpolated values.
  - In the example: `'dark mode'` and `true`

The static (fixed) parts of the literal (the template strings) are kept separate from the dynamic parts (the substitutions).

A tag function can return arbitrary values.

19.3.1 Cooked vs. raw template strings (advanced)

So far, we have only seen the *cooked interpretation* of template strings. But tag functions actually get two interpretations:

- A *cooked interpretation* where backslashes have special meaning. For example, `\t` produces a tab character. This interpretation of the template strings is stored as an Array in the first argument.
- A *raw interpretation* where backslashes do not have special meaning. For example, `\t` produces two characters – a backslash and a `t`. This interpretation of the template strings is stored in property `.raw` of the first argument (an Array).

The following tag function `cookedRaw` uses both interpretations:

```
function cookedRaw(templateStrings, ...substitutions) {
  return {
```

```
      cooked: [...templateStrings], // copy just the Array elements
      raw: templateStrings.raw,
      substitutions,
    };
  }
  assert.deepEqual(
    cookedRaw`\tab${'subst'}\newline\\`,
    {
      cooked: ['\tab', '\newline\\'],
      raw:    ['\\tab', '\\newline\\\\'],
      substitutions: ['subst'],
    });
```

The raw interpretation 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.

19.3.2 Tag function library: lit-html

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

```
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.

19.3.3 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 `${}` (line A):

```
const RE_YEAR = re`(?<year>[0-9]{4})`;
const RE_MONTH = re`(?<month>[0-9]{2})`;
const RE_DAY = re`(?<day>[0-9]{2})`;
const RE_DATE = re`/${RE_YEAR}-${RE_MONTH}-${RE_DAY}/u`; // (A)
```

```
const match = RE_DATE.exec('2017-01-27');
assert.equal(match.groups.year, '2017');
```

19.3.4 Tag function library: graphql-tag

The library graphql-tag 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.

19.4 Raw string literals

Raw string literals are implemented via the tag function `String.raw`. They are string literals where backslashes don't do anything special (such as escaping characters, etc.):

```
assert.equal(String.raw`\back`, '\\back');
```

This helps whenever data contains backslashes – for example, strings with regular expressions:

```
const regex1 = /^\./;
const regex2 = new RegExp('^\\.');
const regex3 = new RegExp(String.raw`^\.`);
```

All three regular expressions are equivalent. With a normal 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.

Raw string literals are also useful for specifying Windows filename paths:

```
const WIN_PATH = String.raw`C:\foo\bar`;
assert.equal(WIN_PATH, 'C:\\foo\\bar');
```

19.5 (Advanced)

All remaining sections are advanced

19.6 Multiline template literals and indentation

If you put multiline text in template literals, two goals are in conflict: On one hand, the template literal should be indented to fit inside the source code. On the other hand, the lines of its content should start in the leftmost column.

For example:

```
function div(text) {
  return `
    <div>
      ${text}
    </div>
  `;
}
console.log('Output:');
console.log(
  div('Hello!')
  // Replace spaces with mid-dots:
  .replace(/ /g, '·')
  // Replace \n with #\n:
  .replace(/\n/g, '#\n')
);
```

Due to the indentation, the template literal fits well into the source code. Alas, the output is also indented. And we don't want the return at the beginning and the return plus two spaces at the end.

```
Output:
#
····<div>#
··········Hello!#
····</div>#
··
```

There are two ways to fix this: via a tagged template or by trimming the result of the template literal.

19.6.1 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 shortens the indentation everywhere. It also removes the line break at the very beginning and the indentation at the very end. One such template tag is `dedent` by Desmond Brand:

```
import dedent from 'dedent';
function divDedented(text) {
  return dedent`
    <div>
      ${text}
    </div>
```

```
  `.replace(/\n/g, '#\n');
}
console.log('Output:');
console.log(divDedented('Hello!'));
```

This time, the output is not indented:

```
Output:
<div>#
  Hello!#
</div>
```

19.6.2 Fix: `.trim()`

The second fix is quicker, but also dirtier:

```
function divDedented(text) {
  return `
<div>
  ${text}
</div>
  `.trim().replace(/\n/g, '#\n');
}
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 that you don't need a custom tag function. The downside is that it looks ugly.

The output is the same as with `dedent`:

```
Output:
<div>#
  Hello!#
</div>
```

19.7 Simple templating via template literals

While template literals look like text templates, it is not immediately obvious how to use them for (text) templating: A text template gets its data from an object, while a template literal gets its data from variables. The solution is to use a template literal in the body of a function whose parameter receives the templating data – for example:

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

19.7.1 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        `.trim()
10   ).join('')}
11 </table>
12 `.trim();
```

This code contains two templating functions:

- The first one (line 1) takes `addrs`, an Array with addresses, and returns a string with a table.
- The second one (line 4) takes `addr`, an object containing an address, and returns a string with a table row. Note the `.trim()` at the end, which removes unnecessary whitespace.

The first templating function produces its result by wrapping a table element around an Array that it joins into a string (line 10). That Array is produced by mapping the second templating function to each element of `addrs` (line 3). It therefore contains strings with table rows.

The helper function `escapeHtml()` is used to escape special HTML characters (line 6 and line 7). Its implementation is shown in the next subsection.

Let us 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>
```

19.7.2 Simple HTML-escaping

The following function escapes plain text so that it is displayed verbatim in HTML:

```
function escapeHtml(str) {
  return str
    .replace(/&/g, '&amp;') // first!
    .replace(/>/g, '&gt;')
    .replace(/</g, '&lt;')
    .replace(/"/g, '&quot;')
    .replace(/'/g, '&#39;')
    .replace(/`/g, '&#96;')
    ;
}
assert.equal(
  escapeHtml('Rock & Roll'), 'Rock &amp; Roll');
assert.equal(
  escapeHtml('<blank>'), '&lt;blank&gt;');
```

Exercise: HTML templating

Exercise with bonus challenge: `exercises/template-literals/templating_test.mjs`

Quiz

See quiz app.

Chapter 20

Symbols

Contents

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`:

```
const sym = Symbol();
assert.equal(typeof sym, 'symbol');
```

And they can be property keys in objects:

```
const obj = {
  [sym]: 123,
};
```

20.1 Use cases for symbols

The main use cases for symbols, are:

- Values for constants
- Unique property keys

20.1.1 Symbols: values for constants

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 symbols:

```
const COLOR_BLUE = Symbol('Blue');
const MOOD_BLUE = Symbol('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);
```

20.1.2 Symbols: unique property keys

The keys of properties (fields) in objects are used at two levels:

- The program operates at a *base level*. The keys at that level reflect the problem that the program solves.
- Libraries and ECMAScript operate at a *meta-level*. The keys at that level are used by services operating on base-level data and code. One such key is `'toString'`.

The following code demonstrates the difference:

```
const pt = {
  x: 7,
  y: 4,
  toString() {
    return `(${this.x}, ${this.y})`;
  },
};
assert.equal(String(pt), '(7, 4)');
```

Properties `.x` and `.y` exist at the base level. They hold the coordinates of the point represented by `pt` and are used to solve a problem – computing with points. Method `.toString()` exists at a meta-level. It is used by JavaScript to convert this object to a string.

Meta-level properties must never interfere with base-level properties. That is, their keys must never overlap. That is difficult when both language and libraries contribute to the meta-level. For example, it is now impossible to give new meta-level methods simple names, such as `toString` because they might clash with existing base-level names. Python's solution to this problem is to prefix and suffix special names with two underscores: `__init__`, `__iter__`, `__hash__`, etc. However, even with this solution, libraries can't have their own meta-level properties because those might be in conflict with future language properties.

Symbols, used as property keys, help us here: Each symbol is unique and a symbol key never clashes with any other string or symbol key.

20.1.2.1 Example: a library with a meta-level method

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 = {
  _id: 'kf12oi',
```

```
  [specialMethod]() { // (A)
    return this._id;
  }
};
assert.equal(obj[specialMethod](), 'kf12oi');
```

The square brackets in line A enable us to specify that the method must have the key `specialMethod`. More details are explained in §25.5.2 "Computed property keys".

20.2 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. For more information on this topic, see [content not included].
- `Symbol.hasInstance`: customizes how `instanceof` works. If an object implements a method with that key, it can be used at the right-hand side of that operator. For example:

  ```
  const PrimitiveNull = {
    [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.mjs`
- `Symbol.hasInstance`: `exercises/symbols/has_instance_test.mjs`

20.3 Converting symbols

What happens if we convert a symbol `sym` to another primitive type? Tbl. 20.1 has the answers.

Table 20.1: The results of converting symbols to other primitive types.

Convert to	Explicit conversion	Coercion (implicit conv.)
boolean	`Boolean(sym)` → OK	`!sym` → OK
number	`Number(sym)` → `TypeError`	`sym*2` → `TypeError`
string	`String(sym)` → OK	`''+sym` → `TypeError`
	`sym.toString()` → OK	`` `${sym}` `` → `TypeError`

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 (which is a different kind of 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)'
```

Quiz

See quiz app.

Part V

Control flow and data flow

Chapter 21

Control flow statements

Contents

This chapter covers the following control flow statements:

- `if` statement (ES1)
- `switch` statement (ES3)
- `while` loop (ES1)

- `do-while` loop (ES3)
- `for` loop (ES1)
- `for-of` loop (ES6)
- `for-await-of` loop (ES2018)
- `for-in` loop (ES1)

Before we get to the actual control flow statements, let's take a look at two operators for controlling loops.

21.1 Conditions of control flow statements

`if`, `while`, and `do-while` have conditions that are, in principle, boolean. However, a condition only has to be *truthy* (`true` if coerced to boolean) in order to be accepted. In other words, the following two control flow statements are equivalent:

```
if (value) {}
if (Boolean(value) === true) {}
```

This is a list of all *falsy* values:

- `undefined`, `null`
- `false`
- `0`, `NaN`
- `''`

All other values are truthy. For more information, see §14.2 "Falsy and truthy values".

21.2 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.

21.2.1 `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'
```

21.2.2 `break` plus label: leaving any labeled statement

`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
  // ···
}
```

In the following example, we use break with a label to leave a loop differently when we succeeded (line A). Then we skip what comes directly after the loop, which is where we end up if we failed.

```
function findSuffix(stringArray, suffix) {
  let result;
  search_block: {
    for (const str of stringArray) {
      if (str.endsWith(suffix)) {
        // Success:
        result = str;
        break search_block; // (A)
      }
    } // for
    // Failure:
    result = '(Untitled)';
  } // search_block

  return { suffix, result };
    // Same as: {suffix: suffix, result: result}
}
assert.deepEqual(
  findSuffix(['foo.txt', 'bar.html'], '.html'),
  { suffix: '.html', result: 'bar.html' }
);
assert.deepEqual(
  findSuffix(['foo.txt', 'bar.html'], '.mjs'),
  { suffix: '.mjs', result: '(Untitled)' }
);
```

21.2.3 `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'
```

21.3 `if` statements

These are two simple `if` statements: one with just a "then" branch and one with both a "then" branch and an "else" branch:

```
if (cond) {
  // then branch
}

if (cond) {
  // then branch
} else {
  // else branch
}
```

Instead of the block, `else` can also be followed by another `if` statement:

```
if (cond1) {
  // ···
} else if (cond2) {
  // ···
}

if (cond1) {
  // ···
} else if (cond2) {
  // ···
} else {
  // ···
}
```

You can continue this chain with more `else if`s.

21.3.1 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 we can use any 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.

21.4 `switch` statements

A `switch` statement looks as follows:

```
switch («switch_expression») {
  «switch_body»
}
```

The body of `switch` consists of zero or more case clauses:

```
case «case_expression»:
  «statements»
```

And, optionally, a default clause:

```
default:
  «statements»
```

A `switch` is executed as follows:

- It evaluates the switch expression.
- It jumps to the first case clause whose expression has the same result as the switch expression.
- Otherwise, if there is no such clause, it jumps to the default clause.
- Otherwise, if there is no default clause, it does nothing.

21.4.1 A first example of a `switch` statement

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');
```

21.4.2 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 englishToFrench(english) {
  let french;
  switch (english) {
    case 'hello':
      french = 'bonjour';
    case 'goodbye':
      french = 'au revoir';
  }
  return french;
}
// The result should be 'bonjour'!
assert.equal(englishToFrench('hello'), 'au revoir');
```

That is, our implementation of `dayOfTheWeek()` only worked because we used `return`. We can fix `englishToFrench()` by using `break`:

```
function englishToFrench(english) {
  let french;
  switch (english) {
    case 'hello':
      french = 'bonjour';
      break;
    case 'goodbye':
      french = 'au revoir';
      break;
  }
  return french;
}
assert.equal(englishToFrench('hello'), 'bonjour'); // ok
```

21.4.3 Empty case 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);
```

21.4.4 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/number_to_month_test.mjs`
- Bonus: `exercises/control-flow/is_object_via_switch_test.mjs`

21.5 `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.

21.5.1 Examples of `while` loops

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 always evaluates to `true`, then `while` is an infinite loop:

```
while (true) {
  if (Math.random() === 0) break;
}
```

21.6 `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:');
  console.log(input);
} while (input !== ':q');
```

`prompt()` is a global function that is available in web browsers. It prompts the user to input text and returns it.

21.7 `for` loops

A `for` loop has the following syntax:

```
for («initialization»; «condition»; «post_iteration») {
  «statements»
}
```

The first line is the *head* of the loop and controls how often the *body* (the remainder of the loop) is executed. It has three parts and each of them is optional:

- `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»
}
```

21.7.1 Examples of `for` loops

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<arr.length; 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;
}
```

21.8 `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);
}
```

21.8.1 `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.

21.8.2 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);
}
```

21.8.3 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'
```

With `[index, element]`, we are using *destructuring* to access Array elements.

Exercise: `for-of`

`exercises/control-flow/array_to_string_test.mjs`

21.9 `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 the chapter on asynchronous iteration.

21.10 `for-in` loops (avoid)

Recommendation: don't use `for-in` loops

`for-in` has several pitfalls. Therefore, it is usually best to avoid it.

This is an example of using `for-in` properly, which involves boilerplate code (line A):

```
function getOwnPropertyNames(obj) {
  const result = [];
  for (const key in obj) {
    if ({}.hasOwnProperty.call(obj, key)) { // (A)
      result.push(key);
    }
  }
  return result;
}
assert.deepEqual(
  getOwnPropertyNames({ a: 1, b:2 }),
  ['a', 'b']);
assert.deepEqual(
  getOwnPropertyNames(['a', 'b']),
  ['0', '1']); // strings!
```

We can implement the same functionality without `for-in`, which is almost always better:

```
function getOwnPropertyNames(obj) {
  const result = [];
  for (const key of Object.keys(obj)) {
    result.push(key);
  }
```

```
    return result;
  }
```

Quiz

See quiz app.

Chapter 22

Exception handling

Contents

This chapter covers how JavaScript handles exceptions.

Why doesn't JavaScript throw exceptions more often?

JavaScript didn't support exceptions until ES3. That explains why they are used sparingly by the language and its standard library.

22.1 Motivation: throwing and catching exceptions

Consider the following code. It reads profiles stored in files into an Array with instances of class `Profile`:

```
function readProfiles(filePaths) {
  const profiles = [];
  for (const filePath of filePaths) {
    try {
      const profile = readOneProfile(filePath);
      profiles.push(profile);
```

```
    } catch (err) { // (A)
      console.log('Error in: '+filePath, err);
    }
  }
}
function readOneProfile(filePath) {
  const profile = new Profile();
  const file = openFile(filePath);
  // ··· (Read the data in `file` into `profile`)
  return profile;
}
function openFile(filePath) {
  if (!fs.existsSync(filePath)) {
    throw new Error('Could not find file '+filePath); // (B)
  }
  // ··· (Open the file whose path is `filePath`)
}
```

Let's examine what happens in line B: An error occurred, but the best place to handle the problem is not the current location, it's line A. There, we can skip the current file and move on to the next one.

Therefore:

- In line B, we use a `throw` statement to indicate that there was a problem.
- In line A, we use a `try-catch` statement to handle the problem.

When we throw, the following constructs are active:

```
readProfiles(···)
  for (const filePath of filePaths)
    try
      readOneProfile(···)
        openFile(···)
          if (!fs.existsSync(filePath))
            throw
```

One by one, `throw` exits the nested constructs, until it encounters a `try` statement. Execution continues in the `catch` clause of that `try` statement.

22.2 `throw`

This is the syntax of the `throw` statement:

```
throw «value»;
```

Any value can be thrown, but it's best to throw an instance of `Error` or its subclasses.

```
throw new Error('Problem!');
```

22.2.1 Options for creating error objects

- Use class `Error`. That is less limiting in JavaScript than in a more static language because you can add your own properties to instances:

```
const err = new Error('Could not find the file');
err.filePath = filePath;
throw err;
```

- Use one of JavaScript's subclasses of `Error` (which are listed later).
- Subclass `Error` yourself.

```
class MyError extends Error {
}
function func() {
  throw new MyError('Problem!');
}
assert.throws(
  () => func(),
  MyError);
```

22.3 The `try` statement

The maximal version of the `try` statement looks as follows:

```
try {
  «try_statements»
} catch (error) {
  «catch_statements»
} finally {
  «finally_statements»
}
```

You can combine these clauses as follows:

- `try-catch`
- `try-finally`
- `try-catch-finally`

Since ECMAScript 2019, you can omit the `catch` parameter `(error)`, if you are not interested in the value that was thrown.

22.3.1 The `try` block

The `try` block can be considered the body of the statement. This is where we execute the regular code.

22.3.2 The `catch` clause

If an exception reaches the `try` block, then it is assigned to the parameter of the `catch` clause and the code in that clause is executed. Next, execution normally continues after

the `try` statement. That may change if:

- There is a `return`, `break`, or `throw` inside the `catch` block.
- There is a `finally` clause (which is always executed before the `try` statement ends).

The following code demonstrates that the value that is thrown in line A is indeed caught in line B.

```
const errorObject = new Error();
function func() {
  throw errorObject; // (A)
}

try {
  func();
} catch (err) { // (B)
  assert.equal(err, errorObject);
}
```

22.3.3 The `finally` clause

The code inside the `finally` clause is always executed at the end of a `try` statement – no matter what happens in the `try` block or the `catch` clause.

Let's look at a common use case for `finally`: You have created a resource and want to always destroy it when you are done with it, no matter what happens while working with it. You'd implement that as follows:

```
const resource = createResource();
try {
  // Work with `resource`. Errors may be thrown.
} finally {
  resource.destroy();
}
```

22.3.3.1 `finally` is always executed

The `finally` clause is always executed, even if an error is thrown (line A):

```
let finallyWasExecuted = false;
assert.throws(
  () => {
    try {
      throw new Error(); // (A)
    } finally {
      finallyWasExecuted = true;
    }
  },
  Error
);
assert.equal(finallyWasExecuted, true);
```

And even if there is a `return` statement (line A):

```
let finallyWasExecuted = false;
function func() {
  try {
    return; // (A)
  } finally {
    finallyWasExecuted = true;
  }
}
func();
assert.equal(finallyWasExecuted, true);
```

22.4 Error classes

`Error` is the common superclass of all built-in error classes. It has the following subclasses (I'm quoting the ECMAScript specification):

- `RangeError`: Indicates a value that is not in the set or range of allowable values.
- `ReferenceError`: Indicate that an invalid reference value has been detected.
- `SyntaxError`: Indicates that a parsing error has occurred.
- `TypeError`: is used to indicate an unsuccessful operation when none of the other *NativeError* objects are an appropriate indication of the failure cause.
- `URIError`: Indicates that one of the global URI handling functions was used in a way that is incompatible with its definition.

22.4.1 Properties of error objects

Consider `err`, an instance of `Error`:

```
const err = new Error('Hello!');
assert.equal(String(err), 'Error: Hello!');
```

Two properties of `err` are especially useful:

- `.message`: contains just the error message.

  ```
  assert.equal(err.message, 'Hello!');
  ```

- `.stack`: contains a stack trace. It is supported by all mainstream browsers.

  ```
  assert.equal(
  err.stack,
  `
  Error: Hello!
      at ch_exception-handling.mjs:1:13
  `.trim());
  ```

Exercise: Exception handling

exercises/exception-handling/call_function_test.mjs

Quiz

See quiz app.

Chapter 23

Callable values

Contents

23.1 Kinds of functions

JavaScript has two categories of functions:

- An *ordinary function* can play several roles:
  - Real function
  - Method
  - Constructor function
- A *specialized function* can only play one of those roles – for example:
  - An *arrow function* can only be a real function.
  - A *method* can only be a method.
  - A *class* can only be a constructor function.

The next two sections explain what all of those things mean.

23.2 Ordinary functions

The following code shows three ways of doing (roughly) the same thing: creating an ordinary function.

```
// Function declaration (a statement)
function ordinary1(a, b, c) {
  // ···
}

// const plus anonymous function expression
const ordinary2 = function (a, b, c) {
  // ···
};

// const plus named function expression
const ordinary3 = function myName(a, b, c) {
  // `myName` is only accessible in here
};
```

As we have seen in §10.8 "Declarations: scope and activation", function declarations are activated early, while variable declarations (e.g., via `const`) are not.

The syntax of function declarations and function expressions is very similar. The context determines which is which. For more information on this kind of syntactic ambiguity, consult §6.5 "Ambiguous syntax".

23.2.1 Parts of a function declaration

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` statement explicitly returns a value from the function.

23.2.2 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.

  ```
  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); // (A)
  ```

  In line A, `obj` is called the *receiver* of the method call. It can be accessed via `this` inside the method.

- 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.)

Ordinary function vs. real function

In JavaScript, we distinguish:

- The entity *ordinary function*
- The role *real function*, as played by an ordinary function

In many other programming languages, the entity *function* only plays one role – *function*. Therefore, the same name *function* can be used for both.

23.2.3 Names of ordinary functions

The name of a function expression is only accessible inside the function, where the function can use it to refer to itself (e.g., for self-recursion):

```
const func = function funcExpr() { return funcExpr };
assert.equal(func(), func);

// The name `funcExpr` only exists inside the function:
assert.throws(() => funcExpr(), ReferenceError);
```

In contrast, the name of a function declaration is accessible inside the current scope:

```
function funcDecl() { return funcDecl }

// The name `funcDecl` exists in the current scope
assert.equal(funcDecl(), funcDecl);
```

23.3 Specialized functions

Specialized functions are single-purpose versions of ordinary functions. Each one of them specializes in a single role:

- The purpose of an *arrow function* is to be a real function:

  ```
  const arrow = () => { return 123 };
  assert.equal(arrow(), 123);
  ```

- The purpose of a *method* is to be a method:

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

- The purpose of a *class* is to 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 jobs 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 classes.

Tbl. 23.1 lists the capabilities of ordinary and specialized functions.

Table 23.1: Capabilities of four kinds of functions. "Lexical `this`" means that `this` is defined by the surroundings of an arrow function, not by method calls.

	Function call	Method call	Constructor call
Ordinary function	(`this === undefined`)	✔	✔
Arrow function	✔	(lexical `this`)	✘
Method	(`this === undefined`)	✔	✘
Class	✘	✘	✔

23.3.1 Specialized functions are still functions

It's important to note that arrow functions, methods, and classes are still categorized as functions:

```
> (() => {}) instanceof Function
true
> ({ method() {} }.method) instanceof Function
```

```
true
> (class SomeClass {}) instanceof Function
true
```

23.3.2 Recommendation: prefer specialized functions

Normally, you should prefer specialized functions over ordinary functions, especially classes and methods. The choice between an arrow function and an ordinary function is less clear-cut, though:

- On one hand, an ordinary function has `this` as an implicit parameter. That parameter is set to `undefined` during function calls – which is not what you want. An arrow function treats `this` like any other variable. For details, see §25.4.6 "Avoiding the pitfalls of `this`".
- On the other hand, I like the syntax of a function declaration (which produces an ordinary function). If you don't use `this` inside it, it is mostly equivalent to `const` plus arrow function:

```
function funcDecl(x, y) {
  return x * y;
}
const arrowFunc = (x, y) => {
  return x * y;
};
```

23.3.3 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.

Next, we'll first look at the syntax of arrow functions and then how they help with `this`.

23.3.3.1 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 previous example demonstrates one benefit of arrow functions – conciseness. If we perform the same task with a function expression, our code is more verbose:

```
[1,2,3].map(function (x) { return x+1 });
```

23.3.3.2 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`.
- The 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 the issue:

```
const person = {
  name: 'Jill',
  someMethod() {
    const ordinaryFunc = function () {
      assert.throws(
        () => this.name, // (A)
        /^TypeError: Cannot read property 'name' of undefined$/);
    };
    const arrowFunc = () => {
      assert.equal(this.name, 'Jill'); // (B)
    };

    ordinaryFunc();
    arrowFunc();
  },
}
```

In this code, we can observe two ways of handling `this`:

- Dynamic `this`: In line A, we try to access the `this` of `.someMethod()` from an ordinary function. There, it is *shadowed* by the function's own `this`, which is `undefined` (as filled in by the function call). Given that ordinary functions receive their `this` via (dynamic) function or method calls, their `this` is called *dynamic*.

- Lexical `this`: In line B, we again try to access the `this` of `.someMethod()`. This time, we succeed because the arrow function does not have its own `this`. `this` is resolved *lexically*, just like any other variable. That's why the `this` of arrow functions is called *lexical*.

23.3.3.3 Syntax pitfall: returning an object literal from an arrow function

If you want the expression body of an arrow function to be an object literal, you must put the literal in parentheses:

```
const func1 = () => ({a: 1});
assert.deepEqual(func1(), { a: 1 });
```

If you don't, JavaScript thinks, the arrow function has a block body (that doesn't return anything):

```
const func2 = () => {a: 1};
assert.deepEqual(func2(), undefined);
```

`{a: 1}` is interpreted as a block with the label `a:` and the expression statement `1`. Without an explicit `return` statement, the block body returns `undefined`.

This pitfall is caused by syntactic ambiguity: object literals and code blocks have the same syntax. We use the parentheses to tell JavaScript that the body is an expression (an object literal) and not a statement (a block).

For more information on shadowing `this`, consult §25.4.5 "`this` pitfall: accidentally shadowing `this`".

23.4 More kinds of functions and methods

This section is a summary of upcoming content

This section mainly serves as a reference for the current and upcoming chapters. Don't worry if you don't understand everything.

So far, all (real) functions and methods, that we have seen, were:

- Single-result
- Synchronous

Later chapters will cover other modes of programming:

- *Iteration* treats objects as containers of data (so-called *iterables*) and provides a standardized way for retrieving what is inside them. If a function or a method returns an iterable, it returns multiple values.
- *Asynchronous programming* deals with handling a long-running computation. You are notified when the computation is finished and can do something else in between. The standard pattern for asynchronously delivering single results is called *Promise*.

These modes can be combined – for example, there are synchronous iterables and asynchronous iterables.

Several new kinds of functions and methods help with some of the mode combinations:

- *Async functions* help implement functions that return Promises. There are also *async methods*.

- *Synchronous generator functions* help implement functions that return synchronous iterables. There are also *synchronous generator methods*.
- *Asynchronous generator functions* help implement functions that return asynchronous iterables. There are also *asynchronous generator methods*.

That leaves us with 4 kinds (2 × 2) of functions and methods:

- Synchronous vs. asynchronous
- Generator vs. single-result

Tbl. 23.2 gives an overview of the syntax for creating these 4 kinds of functions and methods.

Table 23.2: Syntax for creating functions and methods. The last column specifies how many values are produced by an entity.

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

23.5 Returning values from functions and methods

(Everything mentioned in this section applies to both functions and methods.)

The `return` statement explicitly returns a value 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);
```

23.6 Parameter handling

Once again, I am only mentioning functions in this section, but everything also applies to methods.

23.6.1 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*.

23.6.2 Terminology: callback

A *callback* or *callback function* is a function that is an argument of a function or method call.

The following is an example of a callback:

```
const myArray = ['a', 'b'];
const callback = (x) => console.log(x);
myArray.forEach(callback);

// Output:
// 'a'
// 'b'
```

JavaScript uses the term *callback* broadly

In other programming languages, the term *callback* often has a narrower meaning: it refers to a pattern for delivering results asynchronously, via a function-valued parameter. In this meaning, the *callback* (or *continuation*) is invoked after a function has completely finished its computation.

> Callbacks as an asynchronous pattern, are described in the chapter on asynchronous programming.

23.6.3 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]);
```

23.6.4 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]);
```

23.6.5 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, []]);
```

23.6.5.1 Enforcing a certain number of arguments via a rest parameter

You can use a rest parameter to enforce a certain number of arguments. Take, for example, the following function:

```
function createPoint(x, y) {
  return {x, y};
    // same as {x: x, y: y}
}
```

This is how we force callers to always provide two arguments:

```
function createPoint(...args) {
  if (args.length !== 2) {
    throw new Error('Please provide exactly 2 arguments!');
  }
  const [x, y] = args; // (A)
  return {x, y};
}
```

In line A, we access the elements of `args` via *destructuring*.

23.6.6 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.
- The order of the arguments 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 omit (with positional parameters, you have to fill in preceding optional parameters, with `undefined`).

23.6.7 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 });
```

23.6.8 Spreading (...) into function calls

If you put three dots (`...`) in front of the argument of a function call, then you *spread* it. That means that the argument must be an *iterable* object and the iterated values all become arguments. In other words, a single argument is expanded into multiple arguments – for example:

```
function func(x, y) {
  console.log(x);
  console.log(y);
}
const someIterable = ['a', 'b'];
func(...someIterable);
```

```
  // same as func('a', 'b')

// Output:
// 'a'
// 'b'
```

Spreading and rest parameters use the same syntax (`...`), but they serve opposite purposes:

- Rest parameters are used when defining functions or methods. They collect arguments into Arrays.
- Spread arguments are used when calling functions or methods. They turn iterable objects into arguments.

23.6.8.1 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
```

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

Similarly, the Array method `.push()` destructively adds its zero or more parameters to the end of its Array. JavaScript has no method for destructively appending an Array to another one. 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.mjs`
- Named parameters: `exercises/callables/named_parameters_test.mjs`

23.7 Dynamically evaluating code: `eval()`, `new Function()` (advanced)

Next, we'll look at two ways of evaluating code dynamically: `eval()` and `new Function()`.

23.7.1 `eval()`

Given a string `str` with JavaScript code, `eval(str)` evaluates that code and returns the result:

```
> eval('2 ** 4')
16
```

There are two ways of invoking `eval()`:

- *Directly*, via a function call. Then the code in its argument is evaluated inside the current scope.
- *Indirectly*, not via a function call. Then it evaluates its code in global scope.

"Not via a function call" means "anything that looks different than `eval(···)`":

- `eval.call(undefined, '···')`
- `(0, eval)('···')` (uses the comma operator)
- `globalThis.eval('···')`
- `const e = eval; e('···')`
- Etc.

The following code illustrates the difference:

```
globalThis.myVariable = 'global';
function func() {
  const myVariable = 'local';

  // Direct eval
  assert.equal(eval('myVariable'), 'local');

  // Indirect eval
  assert.equal(eval.call(undefined, 'myVariable'), 'global');
}
```

Evaluating code in global context is safer because the code has access to fewer internals.

23.7.2 `new Function()`

`new Function()` creates a function object and is invoked as follows:

```
const func = new Function('«param_1»', ···, '«param_n»', '«func_body»');
```

The previous statement is equivalent to the next statement. Note that «`param_1`», etc., are not inside string literals, anymore.

```
const func = function («param_1», ···, «param_n») {
  «func_body»
};
```

In the next example, we create the same function twice, first via `new Function()`, then via a function expression:

```
const times1 = new Function('a', 'b', 'return a * b');
const times2 = function (a, b) { return a * b };
```

`new Function()` creates non-strict mode functions

Functions created via `new Function()` are sloppy.

23.7.3 Recommendations

Avoid dynamic evaluation of code as much as you can:

- It's a security risk because it may enable an attacker to execute arbitrary code with the privileges of your code.
- It may be switched off – for example, in browsers, via a Content Security Policy.

Very often, JavaScript is dynamic enough so that you don't need `eval()` or similar. In the following example, what we are doing with `eval()` (line A) can be achieved just as well without it (line B).

```
const obj = {a: 1, b: 2};
const propKey = 'b';

assert.equal(eval('obj.' + propKey), 2); // (A)
assert.equal(obj[propKey], 2); // (B)
```

If you have to dynamically evaluate code:

- Prefer `new Function()` over `eval()`: it always executes its code in global context and a function provides a clean interface to the evaluated code.
- Prefer indirect `eval` over direct `eval`: evaluating code in global context is safer.

Quiz

See quiz app.

Part VI

Modularity

Chapter 24

Modules

Contents

24.1 Overview: syntax of ECMAScript modules

24.1.1 Exporting

```
// Named exports
export function f() {}
export const one = 1;
export {foo, b as bar};

// Default exports
export default function f() {} // declaration with optional name
// Replacement for `const` (there must be exactly one value)
export default 123;

// Re-exporting from another module
export * from './some-module.mjs';
export {foo, b as bar} from './some-module.mjs';
```

24.1.2 Importing

```
// Named imports
import {foo, bar as b} from './some-module.mjs';
// Namespace import
import * as someModule from './some-module.mjs';
// Default import
import someModule from './some-module.mjs';

// Combinations:
import someModule, * as someModule from './some-module.mjs';
import someModule, {foo, bar as b} from './some-module.mjs';

// Empty import (for modules with side effects)
import './some-module.mjs';
```

24.2 JavaScript source code formats

The current landscape of JavaScript modules is quite diverse: ES6 brought built-in modules, but the source code formats that came before them, are still around, too. Understanding the latter helps understand the former, so let's investigate. The next sections describe the following ways of delivering JavaScript source code:

- *Scripts* are code fragments that browsers run in global scope. They are precursors of modules.
- *CommonJS modules* are a module format that is mainly used on servers (e.g., via Node.js).
- *AMD modules* are a module format that is mainly used in browsers.
- *ECMAScript modules* are JavaScript's built-in module format. It supersedes all previous formats.

Tbl. 24.1 gives an overview of these code formats. Note that for CommonJS modules and ECMAScript modules, two filename extensions are commonly used. Which one is appropriate depends on how you want to use a file. Details are given later in this chapter.

Table 24.1: Ways of delivering JavaScript source code.

	Runs on	Loaded	Filename ext.
Script	browsers	async	`.js`
CommonJS module	servers	sync	`.js` `.cjs`
AMD module	browsers	async	`.js`
ECMAScript module	browsers and servers	async	`.js` `.mjs`

24.2.1 Code before built-in modules was written in ECMAScript 5

Before we get to built-in modules (which were introduced with ES6), all code that you'll see, will be written in ES5. Among other things:

- ES5 did not have `const` and `let`, only `var`.
- ES5 did not have arrow functions, only function expressions.

24.3 Before we had modules, we had scripts

Initially, browsers only had *scripts* – pieces of code that were executed in global scope. As an example, consider an HTML file that loads script files via the following HTML:

```
<script src="other-module1.js"></script>
<script src="other-module2.js"></script>
<script src="my-module.js"></script>
```

The main file is `my-module.js`, where we simulate a module:

```
var myModule = (function () { // Open IIFE
  // Imports (via global variables)
  var importedFunc1 = otherModule1.importedFunc1;
```

```
  var importedFunc2 = otherModule2.importedFunc2;

  // Body
  function internalFunc() {
    // ···
  }
  function exportedFunc() {
    importedFunc1();
    importedFunc2();
    internalFunc();
  }

  // Exports (assigned to global variable `myModule`)
  return {
    exportedFunc: exportedFunc,
  };
})(); // Close IIFE
```

`myModule` is a global variable that is assigned the result of immediately invoking a function expression. The function expression starts in the first line. It is invoked in the last line.

This way of wrapping a code fragment is called *immediately invoked function expression* (IIFE, coined by Ben Alman). What do we gain from an IIFE? `var` is not block-scoped (like `const` and `let`), it is function-scoped: the only way to create new scopes for `var`-declared variables is via functions or methods (with `const` and `let`, you can use either functions, methods, or blocks `{}`). Therefore, the IIFE in the example hides all of the following variables from global scope and minimizes name clashes: `importedFunc1`, `importedFunc2`, `internalFunc`, `exportedFunc`.

Note that we are using an IIFE in a particular manner: at the end, we pick what we want to export and return it via an object literal. That is called the *revealing module pattern* (coined by Christian Heilmann).

This way of simulating modules, has several issues:

- Libraries in script files export and import functionality via global variables, which risks name clashes.
- Dependencies are not stated explicitly, and there is no built-in way for a script to load the scripts it depends on. Therefore, the web page has to load not just the scripts that are needed by the page but also the dependencies of those scripts, the dependencies' dependencies, etc. And it has to do so in the right order!

24.4 Module systems created prior to ES6

Prior to ECMAScript 6, JavaScript did not have built-in modules. Therefore, the flexible syntax of the language was used to implement custom module systems *within* the language. Two popular ones are:

- CommonJS (targeting the server side)

- AMD (Asynchronous Module Definition, targeting the client side)

24.4.1 Server side: CommonJS modules

The original CommonJS standard for modules was created for server and desktop platforms. It was the foundation of the original Node.js module system, where it achieved enormous popularity. Contributing to that popularity were the npm package manager for Node and tools that enabled using Node modules on the client side (browserify, webpack, and others).

From now on, *CommonJS module* means the Node.js version of this standard (which has a few additional features). This is an example of a CommonJS module:

```
// Imports
var importedFunc1 = require('./other-module1.js').importedFunc1;
var importedFunc2 = require('./other-module2.js').importedFunc2;

// Body
function internalFunc() {
  // ···
}
function exportedFunc() {
  importedFunc1();
  importedFunc2();
  internalFunc();
}

// Exports
module.exports = {
  exportedFunc: exportedFunc,
};
```

CommonJS can be characterized as follows:

- Designed for servers.
- Modules are meant to be loaded *synchronously* (the importer waits while the imported module is loaded and executed).
- Compact syntax.

24.4.2 Client side: AMD (Asynchronous Module Definition) modules

The AMD module format was created to be easier to use in browsers than the CommonJS format. Its most popular implementation is RequireJS. The following is an example of an AMD module.

```
define(['./other-module1.js', './other-module2.js'],
  function (otherModule1, otherModule2) {
    var importedFunc1 = otherModule1.importedFunc1;
    var importedFunc2 = otherModule2.importedFunc2;

    function internalFunc() {
```

```
    // ···
  }
  function exportedFunc() {
    importedFunc1();
    importedFunc2();
    internalFunc();
  }

  return {
    exportedFunc: exportedFunc,
  };
});
```

AMD can be characterized as follows:

- Designed for browsers.
- Modules are meant to be loaded *asynchronously*. That's a crucial requirement for browsers, where code can't wait until a module has finished downloading. It has to be notified once the module is available.
- The syntax is slightly more complicated.

On the plus side, AMD modules can be executed directly. In contrast, CommonJS modules must either be compiled before deployment or custom source code must be generated and evaluated dynamically (think `eval()`). That isn't always permitted on the web.

24.4.3 Characteristics of JavaScript modules

Looking at CommonJS and AMD, similarities between JavaScript module systems emerge:

- There is one module per file.
- Such a file is basically a piece of code that is executed:
  - Local scope: The code is executed in a local "module scope". Therefore, by default, all of the variables, functions, and classes declared in it are internal and not global.
  - Exports: If you want any declared entity to be exported, you must explicitly mark it as an export.
  - Imports: Each module can import exported entities from other modules. Those other modules are identified via *module specifiers* (usually paths, occasionally full URLs).
- Modules are *singletons*: Even if a module is imported multiple times, only a single "instance" of it exists.
- No global variables are used. Instead, module specifiers serve as global IDs.

24.5 ECMAScript modules

ECMAScript modules (*ES modules* or *ESM*) were introduced with ES6. They continue the tradition of JavaScript modules and have all of their aforementioned characteristics. Additionally:

- With CommonJS, ES modules share the compact syntax and support for cyclic dependencies.
- With AMD, ES modules share being designed for asynchronous loading.

ES modules also have new benefits:

- The syntax is even more compact than CommonJS's.
- Modules have *static* structures (which can't be changed at runtime). That helps with static checking, optimized access of imports, dead code elimination, and more.
- Support for cyclic imports is completely transparent.

This is an example of ES module syntax:

```
import {importedFunc1} from './other-module1.mjs';
import {importedFunc2} from './other-module2.mjs';

function internalFunc() {
  ...
}

export function exportedFunc() {
  importedFunc1();
  importedFunc2();
  internalFunc();
}
```

From now on, "module" means "ECMAScript module".

24.5.1 ES modules: syntax, semantics, loader API

The full standard of ES modules comprises the following parts:

1. Syntax (how code is written): What is a module? How are imports and exports declared? Etc.
2. Semantics (how code is executed): How are variable bindings exported? How are imports connected with exports? Etc.
3. A programmatic loader API for configuring module loading.

Parts 1 and 2 were introduced with ES6. Work on part 3 is ongoing.

24.6 Named exports and imports

24.6.1 Named exports

Each module can have zero or more *named exports*.

As an example, consider the following two files:

```
lib/my-math.mjs
main.mjs
```

Module `my-math.mjs` has two named exports: `square` and `LIGHTSPEED`.

```
// Not exported, private to module
function times(a, b) {
  return a * b;
}
export function square(x) {
  return times(x, x);
}
export const LIGHTSPEED = 299792458;
```

To export something, we put the keyword `export` in front of a declaration. Entities that are not exported are private to a module and can't be accessed from outside.

24.6.2 Named imports

Module `main.mjs` has a single named import, `square`:

```
import {square} from './lib/my-math.mjs';
assert.equal(square(3), 9);
```

It can also rename its import:

```
import {square as sq} from './lib/my-math.mjs';
assert.equal(sq(3), 9);
```

24.6.2.1 Syntactic pitfall: named importing is not destructuring

Both named importing and destructuring look similar:

```
import {foo} from './bar.mjs'; // import
const {foo} = require('./bar.mjs'); // destructuring
```

But they are quite different:

- Imports remain connected with their exports.
- You can destructure again inside a destructuring pattern, but the `{}` in an import statement can't be nested.
- The syntax for renaming is different:

  ```
  import {foo as f} from './bar.mjs'; // importing
  const {foo: f} = require('./bar.mjs'); // destructuring
  ```

  Rationale: Destructuring is reminiscent of an object literal (including nesting), while importing evokes the idea of renaming.

Exercise: Named exports

`exercises/modules/export_named_test.mjs`

24.6.3 Namespace imports

Namespace imports are an alternative to named imports. If we namespace-import a module, it becomes an object whose properties are the named exports. This is what `main.mjs` looks like if we use a namespace import:

```
import * as myMath from './lib/my-math.mjs';
assert.equal(myMath.square(3), 9);

assert.deepEqual(
  Object.keys(myMath), ['LIGHTSPEED', 'square']);
```

24.6.4 Named exporting styles: inline versus clause (advanced)

The named export style we have seen so far was *inline*: We exported entities by prefixing them with the keyword `export`.

But we can also use separate *export clauses*. For example, this is what `lib/my-math.mjs` looks like with an export clause:

```
function times(a, b) {
  return a * b;
}
function square(x) {
  return times(x, x);
}
const LIGHTSPEED = 299792458;

export { square, LIGHTSPEED }; // semicolon!
```

With an export clause, we can rename before exporting and use different names internally:

```
function times(a, b) {
  return a * b;
}
function sq(x) {
  return times(x, x);
}
const LS = 299792458;

export {
  sq as square,
  LS as LIGHTSPEED, // trailing comma is optional
};
```

24.7 Default exports and imports

Each module can have at most one *default export*. The idea is that the module *is* the default-exported value.

Avoid mixing named exports and default exports

A module can have both named exports and a default export, but it's usually better to stick to one export style per module.

As an example for default exports, consider the following two files:

```
my-func.mjs
main.mjs
```

Module `my-func.mjs` has a default export:

```
const GREETING = 'Hello!';
export default function () {
  return GREETING;
}
```

Module `main.mjs` default-imports the exported function:

```
import myFunc from './my-func.mjs';
assert.equal(myFunc(), 'Hello!');
```

Note the syntactic difference: the curly braces around named imports indicate that we are reaching *into* the module, while a default import *is* the module.

What are use cases for default exports?

The most common use case for a default export is a module that contains a single function or a single class.

24.7.1 The two styles of default-exporting

There are two styles of doing default exports.

First, you can label existing declarations with `export default`:

```
export default function foo() {} // no semicolon!
export default class Bar {} // no semicolon!
```

Second, you can directly default-export values. In that style, `export default` is itself much like a declaration.

```
export default 'abc';
export default foo();
export default /^xyz$/;
export default 5 * 7;
export default { no: false, yes: true };
```

24.7.1.1 Why are there two default export styles?

The reason is that `export default` can't be used to label `const`: `const` may define multiple values, but `export default` needs exactly one value. Consider the following hypothetical code:

```
// Not legal JavaScript!
export default const foo = 1, bar = 2, baz = 3;
```

With this code, you don't know which one of the three values is the default export.

Exercise: Default exports

`exercises/modules/export_default_test.mjs`

24.7.2 The default export as a named export (advanced)

Internally, a default export is simply a named export whose name is `default`. As an example, consider the previous module `my-func.mjs` with a default export:

```
const GREETING = 'Hello!';
export default function () {
  return GREETING;
}
```

The following module `my-func2.mjs` is equivalent to that module:

```
const GREETING = 'Hello!';
function greet() {
  return GREETING;
}

export {
  greet as default,
};
```

For importing, we can use a normal default import:

```
import myFunc from './my-func2.mjs';
assert.equal(myFunc(), 'Hello!');
```

Or we can use a named import:

```
import {default as myFunc} from './my-func2.mjs';
assert.equal(myFunc(), 'Hello!');
```

The default export is also available via property `.default` of namespace imports:

```
import * as mf from './my-func2.mjs';
assert.equal(mf.default(), 'Hello!');
```

Isn't `default` illegal as a variable name?

`default` can't be a variable name, but it can be an export name and it can be a property name:

```
const obj = {
  default: 123,
};
assert.equal(obj.default, 123);
```

24.8 More details on exporting and importing

24.8.1 Imports are read-only views on exports

So far, we have used imports and exports intuitively, and everything seems to have worked as expected. But now it is time to take a closer look at how imports and exports are really related.

Consider the following two modules:

```
counter.mjs
main.mjs
```

`counter.mjs` exports a (mutable!) variable and a function:

```
export let counter = 3;
export function incCounter() {
  counter++;
}
```

`main.mjs` name-imports both exports. When we use `incCounter()`, we discover that the connection to `counter` is live – we can always access the live state of that variable:

```
import { counter, incCounter } from './counter.mjs';

// The imported value `counter` is live
assert.equal(counter, 3);
incCounter();
assert.equal(counter, 4);
```

Note that while the connection is live and we can read `counter`, we cannot change this variable (e.g., via `counter++`).

There are two benefits to handling imports this way:

- It is easier to split modules because previously shared variables can become exports.
- This behavior is crucial for supporting transparent cyclic imports. Read on for more information.

24.8.2 ESM's transparent support for cyclic imports (advanced)

ESM supports cyclic imports transparently. To understand how that is achieved, consider the following example: fig. 24.1 shows a directed graph of modules importing other modules. P importing M is the cycle in this case.

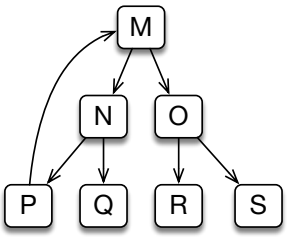

Figure 24.1: A directed graph of modules importing modules: M imports N and O, N imports P and Q, etc.

After parsing, these modules are set up in two phases:

- Instantiation: Every module is visited and its imports are connected to its exports. Before a parent can be instantiated, all of its children must be instantiated.
- Evaluation: The bodies of the modules are executed. Once again, children are evaluated before parents.

This approach handles cyclic imports correctly, due to two features of ES modules:

- Due to the static structure of ES modules, the exports are already known after parsing. That makes it possible to instantiate P before its child M: P can already look up M's exports.
- When P is evaluated, M hasn't been evaluated, yet. However, entities in P can already mention imports from M. They just can't use them, yet, because the imported values are filled in later. For example, a function in P can access an import from M. The only limitation is that we must wait until after the evaluation of M, before calling that function.

  Imports being filled in later is enabled by them being "live immutable views" on exports.

24.9 npm packages

The *npm software registry* is the dominant way of distributing JavaScript libraries and apps for Node.js and web browsers. It is managed via the *npm package manager* (short: *npm*). Software is distributed as so-called *packages*. A package is a directory containing arbitrary files and a file `package.json` at the top level that describes the package. For example, when npm creates an empty package inside a directory `foo/`, you get this `package.json`:

```
{
  "name": "foo",
  "version": "1.0.0",
```

```
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Some of these properties contain simple metadata:

- `name` specifies the name of this package. Once it is uploaded to the npm registry, it can be installed via `npm install foo`.
- `version` is used for version management and follows semantic versioning, with three numbers:
  - Major version: is incremented when incompatible API changes are made.
  - Minor version: is incremented when functionality is added in a backward compatible manner.
  - Patch version: is incremented when backward compatible changes are made.
- `description`, `keywords`, `author` make it easier to find packages.
- `license` clarifies how you can use this package.

Other properties enable advanced configuration:

- `main`: specifies the module that "is" the package (explained later in this chapter).
- `scripts`: are commands that you can execute via `npm run`. For example, the script `test` can be executed via `npm run test`.

For more information on `package.json`, consult the npm documentation.

24.9.1 Packages are installed inside a directory `node_modules/`

npm always installs packages inside a directory `node_modules`. There are usually many of these directories. Which one npm uses, depends on the directory where one currently is. For example, if we are inside a directory `/tmp/a/b/`, npm tries to find a `node_modules` in the current directory, its parent directory, the parent directory of the parent, etc. In other words, it searches the following *chain* of locations:

- `/tmp/a/b/node_modules`
- `/tmp/a/node_modules`
- `/tmp/node_modules`

When installing a package `foo`, npm uses the closest `node_modules`. If, for example, we are inside `/tmp/a/b/` and there is a `node_modules` in that directory, then npm puts the package inside the directory:

```
/tmp/a/b/node_modules/foo/
```

When importing a module, we can use a special module specifier to tell Node.js that we want to import it from an installed package. How exactly that works, is explained later. For now, consider the following example:

```
// /home/jane/proj/main.mjs
import * as theModule from 'the-package/the-module.mjs';
```

To find `the-module.mjs` (Node.js prefers the filename extension `.mjs` for ES modules), Node.js walks up the `node_module` chain and searches the following locations:

- `/home/jane/proj/node_modules/the-package/the-module.mjs`
- `/home/jane/node_modules/the-package/the-module.mjs`
- `/home/node_modules/the-package/the-module.mjs`

24.9.2 Why can npm be used to install frontend libraries?

Finding installed modules in `node_modules` directories is only supported on Node.js. So why can we also use npm to install libraries for browsers?

That is enabled via bundling tools, such as webpack, that compile and optimize code before it is deployed online. During this compilation process, the code in npm packages is adapted so that it works in browsers.

24.10 Naming modules

There are no established best practices for naming module files and the variables they are imported into.

In this chapter, I'm using the following naming style:

- The names of module files are dash-cased and start with lowercase letters:

  ```
  ./my-module.mjs
  ./some-func.mjs
  ```

- The names of namespace imports are lowercased and camel-cased:

  ```
  import * as myModule from './my-module.mjs';
  ```

- The names of default imports are lowercased and camel-cased:

  ```
  import someFunc from './some-func.mjs';
  ```

What are the rationales behind this style?

- npm doesn't allow uppercase letters in package names (source). Thus, we avoid camel case, so that "local" files have names that are consistent with those of npm packages. Using only lowercase letters also minimizes conflicts between file systems that are case-sensitive and file systems that aren't: the former distinguish files whose names have the same letters, but with different cases; the latter don't.
- There are clear rules for translating dash-cased file names to camel-cased JavaScript variable names. Due to how we name namespace imports, these rules work for both namespace imports and default imports.

I also like underscore-cased module file names because you can directly use these names for namespace imports (without any translation):

```
import * as my_module from './my_module.mjs';
```

But that style does not work for default imports: I like underscore-casing for namespace objects, but it is not a good choice for functions, etc.

24.11 Module specifiers

Module specifiers are the strings that identify modules. They work slightly differently in browsers and Node.js. Before we can look at the differences, we need to learn about the different categories of module specifiers.

24.11.1 Categories of module specifiers

In ES modules, we distinguish the following categories of specifiers. These categories originated with CommonJS modules.

- Relative path: starts with a dot. Examples:

  ```
  './some/other/module.mjs'
  '../../lib/counter.mjs'
  ```

- Absolute path: starts with a slash. Example:

  ```
  '/home/jane/file-tools.mjs'
  ```

- URL: includes a protocol (technically, paths are URLs, too). Examples:

  ```
  'https://example.com/some-module.mjs'
  'file:///home/john/tmp/main.mjs'
  ```

- Bare path: does not start with a dot, a slash or a protocol, and consists of a single filename without an extension. Examples:

  ```
  'lodash'
  'the-package'
  ```

- Deep import path: starts with a bare path and has at least one slash. Example:

  ```
  'the-package/dist/the-module.mjs'
  ```

24.11.2 ES module specifiers in browsers

Browsers handle module specifiers as follows:

- Relative paths, absolute paths, and URLs work as expected. They all must point to real files (in contrast to CommonJS, which lets you omit filename extensions and more).
- The file name extensions of modules don't matter, as long as they are served with the content type `text/javascript`.
- How bare paths will end up being handled is not yet clear. You will probably eventually be able to map them to other specifiers via lookup tables.

Note that bundling tools such as webpack, which combine modules into fewer files, are often less strict with specifiers than browsers. That's because they operate at build/-compile time (not at runtime) and can search for files by traversing the file system.

24.11.3 ES module specifiers on Node.js

> ⚠ **Support for ES modules on Node.js is still new**
>
> You may have to switch it on via a command line flag. See the Node.js documentation for details.

Node.js handles module specifiers as follows:

- Relative paths are resolved as they are in web browsers – relative to the path of the current module.
- Absolute paths are currently not supported. As a workaround, you can use URLs that start with `file:///`. You can create such URLs via `url.pathToFileURL()`.
- Only `file:` is supported as a protocol for URL specifiers.
- A bare path is interpreted as a package name and resolved relative to the closest `node_modules` directory. What module should be loaded, is determined by looking at property `"main"` of the package's `package.json` (similarly to CommonJS).
- Deep import paths are also resolved relatively to the closest `node_modules` directory. They contain file names, so it is always clear which module is meant.

All specifiers, except bare paths, must refer to actual files. That is, ESM does not support the following CommonJS features:

- CommonJS automatically adds missing filename extensions.
- CommonJS can import a directory `foo` if there is a `foo/package.json` with a `"main"` property.
- CommonJS can import a directory `foo` if there is a module `foo/index.js`.

All built-in Node.js modules are available via bare paths and have named ESM exports – for example:

```
import * as path from 'path';
import {strict as assert} from 'assert';

assert.equal(
  path.join('a/b/c', '../d'), 'a/b/d');
```

24.11.3.1 Filename extensions on Node.js

Node.js supports the following default filename extensions:

- `.mjs` for ES modules
- `.cjs` for CommonJS modules

The filename extension `.js` stands for either ESM or CommonJS. Which one it is is configured via the "closest" `package.json` (in the current directory, the parent directory, etc.). Using `package.json` in this manner is independent of packages.

In that `package.json`, there is a property `"type"`, which has two settings:

- `"commonjs"` (the default): files with the extension `.js` or without an extension are interpreted as CommonJS modules.
- `"module"`: files with the extension `.js` or without an extension are interpreted as ESM modules.

24.11.3.2 Interpreting non-file source code as either CommonJS or ESM

Not all source code executed by Node.js comes from files. You can also send it code via stdin, `--eval`, and `--print`. The command line option `--input-type` lets you specify how such code is interpreted:

- As CommonJS (the default): `--input-type=commonjs`
- As ESM: `--input-type=module`

24.12 Loading modules dynamically via `import()`

So far, the only way to import a module has been via an `import` statement. That statement has several limitations:

- You must use it at the top level of a module. That is, you can't, for example, import something when you are inside a block.
- The module specifier is always fixed. That is, you can't change what you import depending on a condition. And you can't assemble a specifier dynamically.

The `import()` operator changes that. Let's look at an example of it being used.

24.12.1 Example: loading a module dynamically

Consider the following files:

```
lib/my-math.mjs
main1.mjs
main2.mjs
```

We have already seen module `my-math.mjs`:

```
// Not exported, private to module
function times(a, b) {
  return a * b;
}
export function square(x) {
  return times(x, x);
}
export const LIGHTSPEED = 299792458;
```

This is what using `import()` looks like in `main1.mjs`:

```
const dir = './lib/';
const moduleSpecifier = dir + 'my-math.mjs';
```

```
function loadConstant() {
  return import(moduleSpecifier)
  .then(myMath => {
    const result = myMath.LIGHTSPEED;
    assert.equal(result, 299792458);
    return result;
  });
}
```

Method `.then()` is part of *Promises*, a mechanism for handling asynchronous results, which is covered later in this book.

Two things in this code weren't possible before:

- We are importing inside a function (not at the top level).
- The module specifier comes from a variable.

Next, we'll implement the exact same functionality in `main2.mjs` but via a so-called *async function*, which provides nicer syntax for Promises.

```
const dir = './lib/';
const moduleSpecifier = dir + 'my-math.mjs';

async function loadConstant() {
  const myMath = await import(moduleSpecifier);
  const result = myMath.LIGHTSPEED;
  assert.equal(result, 299792458);
  return result;
}
```

Why is `import()` an operator and not a function?

Even though it works much like a function, `import()` is an operator: in order to resolve module specifiers relatively to the current module, it needs to know from which module it is invoked. A normal function cannot receive this information as implicitly as an operator can. It would need, for example, a parameter.

24.12.2 Use cases for `import()`

24.12.2.1 Loading code on demand

Some functionality of web apps doesn't have to be present when they start, it can be loaded on demand. Then `import()` helps because you can put such functionality into modules – for example:

```
button.addEventListener('click', event => {
  import('./dialogBox.mjs')
    .then(dialogBox => {
      dialogBox.open();
    })
    .catch(error => {
```

```
        /* Error handling */
    })
});
```

24.12.2.2 Conditional loading of modules

We may want to load a module depending on whether a condition is true. For example, a module with a polyfill that makes a new feature available on legacy platforms:

```
if (isLegacyPlatform()) {
  import('./my-polyfill.mjs')
    .then(···);
}
```

24.12.2.3 Computed module specifiers

For applications such as internationalization, it helps if you can dynamically compute module specifiers:

```
import(`messages_${getLocale()}.mjs`)
  .then(···);
```

24.13 Preview: `import.meta.url`

"`import.meta`" is an ECMAScript feature proposed by Domenic Denicola. The object `import.meta` holds metadata for the current module.

Its most important property is `import.meta.url`, which contains a string with the URL of the current module file. For example:

```
'https://example.com/code/main.mjs'
```

24.13.1 `import.meta.url` and class `URL`

Class `URL` is available via a global variable in browsers and on Node.js. You can look up its full functionality in the Node.js documentation. When working with `import.meta.url`, its constructor is especially useful:

```
new URL(input: string, base?: string|URL)
```

Parameter `input` contains the URL to be parsed. It can be relative if the second parameter, `base`, is provided.

In other words, this constructor lets us resolve a relative path against a base URL:

```
> new URL('other.mjs', 'https://example.com/code/main.mjs').href
'https://example.com/code/other.mjs'
> new URL('../other.mjs', 'https://example.com/code/main.mjs').href
'https://example.com/other.mjs'
```

This is how we get a `URL` instance that points to a file `data.txt` that sits next to the current module:

```
const urlOfData = new URL('data.txt', import.meta.url);
```

24.13.2 `import.meta.url` on Node.js

On Node.js, `import.meta.url` is always a string with a `file:` URL – for example:

```
'file:///Users/rauschma/my-module.mjs'
```

24.13.2.1 Example: reading a sibling file of a module

Many Node.js file system operations accept either strings with paths or instances of `URL`. That enables us to read a sibling file `data.txt` of the current module:

```
import {promises as fs} from 'fs';

async function main() {
  const urlOfData = new URL('data.txt', import.meta.url);
  const str = await fs.readFile(urlOfData, {encoding: 'UTF-8'});
  assert.equal(str, 'This is textual data.\n');
}
main();
```

`main()` is an async function, as explained in [content not included].

`fs.promises` contains a Promise-based version of the `fs` API, which can be used with async functions.

24.13.2.2 Converting between `file:` URLs and paths

The Node.js module `url` has two functions for converting between `file:` URLs and paths:

- `fileURLToPath(url: URL|string): string`
  Converts a `file:` URL to a path.
- `pathToFileURL(path: string): URL`
  Converts a path to a `file:` URL.

If you need a path that can be used in the local file system, then property `.pathname` of `URL` instances does not always work:

```
assert.equal(
  new URL('file:///tmp/with%20space.txt').pathname,
  '/tmp/with%20space.txt');
```

Therefore, it is better to use `fileURLToPath()`:

```
import * as url from 'url';
assert.equal(
  url.fileURLToPath('file:///tmp/with%20space.txt'),
  '/tmp/with space.txt'); // result on Unix
```

Similarly, `pathToFileURL()` does more than just prepend `'file://'` to an absolute path.

24.14 Polyfills: emulating native web platform features (advanced)

Backends have polyfills, too

This section is about frontend development and web browsers, but similar ideas apply to backend development.

Polyfills help with a conflict that we are facing when developing a web application in JavaScript:

- On one hand, we want to use modern web platform features that make the app better and/or development easier.
- On the other hand, the app should run on as many browsers as possible.

Given a web platform feature X:

- A *polyfill* for X is a piece of code. If it is executed on a platform that already has built-in support for X, it does nothing. Otherwise, it makes the feature available on the platform. In the latter case, the polyfilled feature is (mostly) indistinguishable from a native implementation. In order to achieve that, the polyfill usually makes global changes. For example, it may modify global data or configure a global module loader. Polyfills are often packaged as modules.
  - The term *polyfill* was coined by Remy Sharp.
- A *speculative polyfill* is a polyfill for a proposed web platform feature (that is not standardized, yet).
  - Alternative term: *prollyfill*
- A *replica* of X is a library that reproduces the API and functionality of X locally. Such a library exists independently of a native (and global) implementation of X.
  - *Replica* is a new term introduced in this section. Alternative term: *ponyfill*
- There is also the term *shim*, but it doesn't have a universally agreed upon definition. It often means roughly the same as *polyfill*.

Every time our web applications starts, it must first execute all polyfills for features that may not be available everywhere. Afterwards, we can be sure that those features are available natively.

24.14.1 Sources of this section

- "What is a Polyfill?" by Remy Sharp
- Inspiration for the term *replica*: The Eiffel Tower in Las Vegas
- Useful clarification of "polyfill" and related terms: "Polyfills and the evolution of the Web". Edited by Andrew Betts.

Quiz

See quiz app.

Chapter 25

Single objects

Contents

In this book, JavaScript's style of object-oriented programming (OOP) is introduced in four steps. This chapter covers step 1; the next chapter covers steps 2–4. The steps are (fig. 25.1):

1. **Single objects (this chapter):** How do *objects*, JavaScript's basic OOP building blocks, work in isolation?
2. **Prototype chains (next chapter):** Each object has a chain of zero or more *prototype objects*. Prototypes are JavaScript's core inheritance mechanism.
3. **Classes (next chapter):** JavaScript's *classes* are factories for objects. The relationship between a class and its instances is based on prototypal inheritance.
4. **Subclassing (next chapter):** The relationship between a *subclass* and its *superclass* is also based on prototypal inheritance.

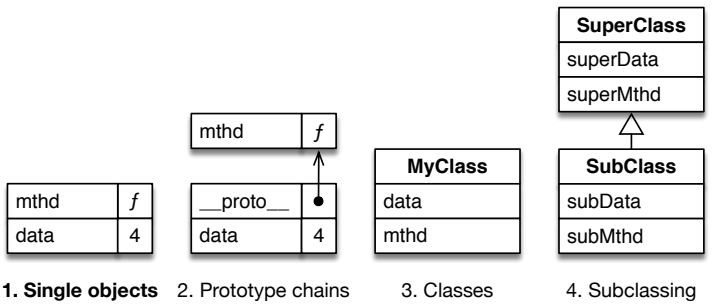

Figure 25.1: This book introduces object-oriented programming in JavaScript in four steps.

25.1 What is an object?

In JavaScript:

- An object is a set of *properties* (key-value entries).
- A property key can only be a string or a symbol.

25.1.1 Roles of objects: record vs. dictionary

Objects play two roles in JavaScript:

- Records: Objects-as-records have a fixed number of properties, whose keys are known at development time. Their values can have different types.
- Dictionaries: Objects-as-dictionaries have a variable number of properties, whose keys are not known at development time. All of their values have the same type.

These roles influence how objects are explained in this chapter:

- First, we'll explore objects-as-records. Even though property keys are strings or symbols under the hood, they will appear as fixed identifiers to us, in this part of the chapter.
- Later, we'll explore objects-as-dictionaries. Note that Maps are usually better dictionaries than objects. However, some of the operations that we'll encounter, can also be useful for objects-as-records.

25.2 Objects as records

Let's first explore the role *record* of objects.

25.2.1 Object literals: properties

Object literals are one way of creating objects-as-records. They are a stand-out feature of JavaScript: you can directly create objects – no need for classes! This is an example:

```
const jane = {
  first: 'Jane',
  last: 'Doe', // optional trailing comma
};
```

In the example, we created an object via an object literal, which starts and ends with curly braces `{}`. Inside it, we defined two *properties* (key-value entries):

- The first property has the key `first` and the value `'Jane'`.
- The second property has the key `last` and the value `'Doe'`.

We will later see other ways of specifying property keys, but with this way of specifying them, they must follow the rules of JavaScript variable names. For example, you can use `first_name` as a property key, but not `first-name`). However, reserved words are allowed:

```
const obj = {
  if: true,
  const: true,
};
```

In order to check the effects of various operations on objects, we'll occasionally use `Object.keys()` in this part of the chapter. It lists property keys:

```
> Object.keys({a:1, b:2})
[ 'a', 'b' ]
```

25.2.2 Object literals: property value shorthands

Whenever the value of a property is defined via a variable name and that name is the same as the key, you can omit the key.

```
function createPoint(x, y) {
  return {x, y};
}
assert.deepEqual(
  createPoint(9, 2),
  { x: 9, y: 2 }
);
```

25.2.3 Getting properties

This is how you *get* (read) a property (line A):

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

// Get property .first
assert.equal(jane.first, 'Jane'); // (A)
```

Getting an unknown property produces `undefined`:

```
assert.equal(jane.unknownProperty, undefined);
```

25.2.4 Setting properties

This is how you *set* (write to) a property:

```
const obj = {
  prop: 1,
};
assert.equal(obj.prop, 1);
obj.prop = 2; // (A)
assert.equal(obj.prop, 2);
```

We just changed an existing property via setting. If we set an unknown property, we create a new entry:

```
const obj = {}; // empty object
assert.deepEqual(
  Object.keys(obj), []);

obj.unknownProperty = 'abc';
```

```
assert.deepEqual(
  Object.keys(obj), ['unknownProperty']);
```

25.2.5 Object literals: methods

The following code shows how to create the method `.says()` via an object literal:

```
const jane = {
  first: 'Jane', // data property
  says(text) {   // method
    return `${this.first} says “${text}”`; // (A)
  }, // comma as separator (optional at end)
};
assert.equal(jane.says('hello'), 'Jane says “hello”');
```

During the method call `jane.says('hello')`, `jane` is called the *receiver* of the method call and assigned to the special variable `this`. That enables method `.says()` to access the sibling property `.first` in line A.

25.2.6 Object literals: accessors

There are two kinds of accessors in JavaScript:

- A *getter* is a method-like entity that is invoked by getting a property.
- A *setter* is a method-like entity that is invoked by setting a property.

25.2.6.1 Getters

A getter is created by prefixing a method definition with the modifier `get`:

```
const jane = {
  first: 'Jane',
  last: 'Doe',
  get full() {
    return `${this.first} ${this.last}`;
  },
};

assert.equal(jane.full, 'Jane Doe');
jane.first = 'John';
assert.equal(jane.full, 'John Doe');
```

25.2.6.2 Setters

A setter is created by prefixing a method definition with the modifier `set`:

```
const jane = {
  first: 'Jane',
  last: 'Doe',
  set full(fullName) {
    const parts = fullName.split(' ');
```

```
    this.first = parts[0];
    this.last = parts[1];
  },
};

jane.full = 'Richard Roe';
assert.equal(jane.first, 'Richard');
assert.equal(jane.last, 'Roe');
```

Exercise: Creating an object via an object literal

`exercises/single-objects/color_point_object_test.mjs`

25.3 Spreading into object literals (...)

Inside a function call, spreading (`...`) turns the iterated values of an *iterable* object into arguments.

Inside an object literal, a *spread property* adds the properties of another object to the current one:

```
> const obj = {foo: 1, bar: 2};
> {...obj, baz: 3}
{ foo: 1, bar: 2, baz: 3 }
```

If property keys clash, the property that is mentioned last "wins":

```
> const obj = {foo: 1, bar: 2, baz: 3};
> {...obj, foo: true}
{ foo: true, bar: 2, baz: 3 }
> {foo: true, ...obj}
{ foo: 1, bar: 2, baz: 3 }
```

All values are spreadable, even `undefined` and `null`:

```
> {...undefined}
{}
> {...null}
{}
> {...123}
{}
> {...'abc'}
{ '0': 'a', '1': 'b', '2': 'c' }
> {...['a', 'b']}
{ '0': 'a', '1': 'b' }
```

Property `.length` of strings and of Arrays is hidden from this kind of operation (it is not *enumerable*; see §25.7.3 "Property attributes and property descriptors" for more information).

25.3.1 Use case for spreading: copying objects

You can use spreading to create a copy of an object `original`:

```
const copy = {...original};
```

Caveat – copying is *shallow*: `copy` is a fresh object with duplicates of all properties (key-value entries) of `original`. But if property values are objects, then those are not copied themselves; they are shared between `original` and `copy`. Let's look at an example:

```
const original = { a: 1, b: {foo: true} };
const copy = {...original};
```

The first level of `copy` is really a copy: If you change any properties at that level, it does not affect the original:

```
copy.a = 2;
assert.deepEqual(
  original, { a: 1, b: {foo: true} }); // no change
```

However, deeper levels are not copied. For example, the value of `.b` is shared between original and copy. Changing `.b` in the copy also changes it in the original.

```
copy.b.foo = false;
assert.deepEqual(
  original, { a: 1, b: {foo: false} });
```

> **JavaScript doesn't have built-in support for deep copying**
>
> *Deep copies* of objects (where all levels are copied) are notoriously difficult to do generically. Therefore, JavaScript does not have a built-in operation for them (for now). If you need such an operation, you have to implement it yourself.

25.3.2 Use case for spreading: default values for missing properties

If one of the inputs of your code is an object with data, you can make properties optional by specifying default values that are used if those properties are missing. One technique for doing so is via an object whose properties contain the default values. In the following example, that object is `DEFAULTS`:

```
const DEFAULTS = {foo: 'a', bar: 'b'};
const providedData = {foo: 1};

const allData = {...DEFAULTS, ...providedData};
assert.deepEqual(allData, {foo: 1, bar: 'b'});
```

The result, the object `allData`, is created by copying `DEFAULTS` and overriding its properties with those of `providedData`.

But you don't need an object to specify the default values; you can also specify them inside the object literal, individually:

```
const providedData = {foo: 1};

const allData = {foo: 'a', bar: 'b', ...providedData};
assert.deepEqual(allData, {foo: 1, bar: 'b'});
```

25.3.3 Use case for spreading: non-destructively changing properties

So far, we have encountered one way of changing a property `.foo` of an object: We *set* it (line A) and mutate the object. That is, this way of changing a property is *destructive*.

```
const obj = {foo: 'a', bar: 'b'};
obj.foo = 1; // (A)
assert.deepEqual(obj, {foo: 1, bar: 'b'});
```

With spreading, we can change `.foo` *non-destructively* – we make a copy of `obj` where `.foo` has a different value:

```
const obj = {foo: 'a', bar: 'b'};
const updatedObj = {...obj, foo: 1};
assert.deepEqual(updatedObj, {foo: 1, bar: 'b'});
```

Exercise: Non-destructively updating a property via spreading (fixed key)

`exercises/single-objects/update_name_test.mjs`

25.4 Methods

25.4.1 Methods are properties whose values are functions

Let's revisit the example that was used to introduce methods:

```
const jane = {
  first: 'Jane',
  says(text) {
    return `${this.first} says “${text}”`;
  },
};
```

Somewhat surprisingly, methods are functions:

```
assert.equal(typeof jane.says, 'function');
```

Why is that? We learned in the chapter on callable values, that ordinary functions play several roles. *Method* is one of those roles. Therefore, under the hood, `jane` roughly looks as follows.

```
const jane = {
  first: 'Jane',
  says: function (text) {
    return `${this.first} says “${text}”`;
```

```
  },
};
```

25.4.2 `.call()`: specifying `this` via a parameter

Remember that each function `someFunc` is also an object and therefore has methods. One such method is `.call()` – it lets you call a function while specifying `this` via a parameter:

```
someFunc.call(thisValue, arg1, arg2, arg3);
```

25.4.2.1 Methods and `.call()`

If you make a method call, `this` is an implicit parameter that is filled in via the receiver of the call:

```
const obj = {
  method(x) {
    assert.equal(this, obj); // implicit parameter
    assert.equal(x, 'a');
  },
};

obj.method('a'); // receiver is `obj`
```

The method call in the last line sets up `this` as follows:

```
obj.method.call(obj, 'a');
```

As an aside, that means that there are actually two different dot operators:

1. One for accessing properties: `obj.prop`
2. One for making method calls: `obj.prop()`

They are different in that (2) is not just (1) followed by the function call operator `()`. Instead, (2) additionally specifies a value for `this`.

25.4.2.2 Functions and `.call()`

If you function-call an ordinary function, its implicit parameter `this` is also provided – it is implicitly set to `undefined`:

```
function func(x) {
  assert.equal(this, undefined); // implicit parameter
  assert.equal(x, 'a');
}

func('a');
```

The method call in the last line sets up `this` as follows:

```
func.call(undefined, 'a');
```

`this` being set to `undefined` during a function call, indicates that it is a feature that is only needed during a method call.

Next, we'll examine the pitfalls of using `this`. Before we can do that, we need one more tool: method `.bind()` of functions.

25.4.3 `.bind()`: pre-filling `this` and parameters of functions

`.bind()` is another method of function objects. This method is invoked as follows:

```
const boundFunc = someFunc.bind(thisValue, arg1, arg2);
```

`.bind()` returns a new function `boundFunc()`. Calling that function invokes `someFunc()` with `this` set to `thisValue` and these parameters: `arg1`, `arg2`, followed by the parameters of `boundFunc()`.

That is, the following two function calls are equivalent:

```
boundFunc('a', 'b')
someFunc.call(thisValue, arg1, arg2, 'a', 'b')
```

25.4.3.1 An alternative to `.bind()`

Another way of pre-filling `this` and parameters is via an arrow function:

```
const boundFunc2 = (...args) =>
  someFunc.call(thisValue, arg1, arg2, ...args);
```

25.4.3.2 An implementation of `.bind()`

Considering the previous section, `.bind()` can be implemented as a real function as follows:

```
function bind(func, thisValue, ...boundArgs) {
  return (...args) =>
    func.call(thisValue, ...boundArgs, ...args);
}
```

25.4.3.3 Example: binding a real function

Using `.bind()` for real functions is somewhat unintuitive because you have to provide a value for `this`. Given that it is `undefined` during function calls, it is usually set to `undefined` or `null`.

In the following example, we create `add8()`, a function that has one parameter, by binding the first parameter of `add()` to `8`.

```
function add(x, y) {
  return x + y;
}

const add8 = add.bind(undefined, 8);
assert.equal(add8(1), 9);
```

25.4.3.4 Example: binding a method

In the following code, we turn method `.says()` into the stand-alone function `func()`:

```
const jane = {
  first: 'Jane',
  says(text) {
    return `${this.first} says “${text}”`; // (A)
  },
};

const func = jane.says.bind(jane, 'hello');
assert.equal(func(), 'Jane says “hello”');
```

Setting `this` to `jane` via `.bind()` is crucial here. Otherwise, `func()` wouldn't work properly because `this` is used in line A.

25.4.4 `this` pitfall: extracting methods

We now know quite a bit about functions and methods and are ready to take a look at the biggest pitfall involving methods and `this`: function-calling a method extracted from an object can fail if you are not careful.

In the following example, we fail when we extract method `jane.says()`, store it in the variable `func`, and function-call `func()`.

```
const jane = {
  first: 'Jane',
  says(text) {
    return `${this.first} says “${text}”`;
  },
};
const func = jane.says; // extract the method
assert.throws(
  () => func('hello'), // (A)
  {
    name: 'TypeError',
    message: "Cannot read property 'first' of undefined",
  });
```

The function call in line A is equivalent to:

```
assert.throws(
  () => jane.says.call(undefined, 'hello'), // `this` is undefined!
  {
    name: 'TypeError',
    message: "Cannot read property 'first' of undefined",
  });
```

So how do we fix this? We need to use `.bind()` to extract method `.says()`:

```
const func2 = jane.says.bind(jane);
assert.equal(func2('hello'), 'Jane says “hello”');
```

The `.bind()` ensures that `this` is always `jane` when we call `func()`.

You can also use arrow functions to extract methods:

```
const func3 = text => jane.says(text);
assert.equal(func3('hello'), 'Jane says “hello”');
```

25.4.4.1 Example: extracting a method

The following is a simplified version of code that you may see in actual web development:

```
class ClickHandler {
  constructor(id, elem) {
    this.id = id;
    elem.addEventListener('click', this.handleClick); // (A)
  }
  handleClick(event) {
    alert('Clicked ' + this.id);
  }
}
```

In line A, we don't extract the method `.handleClick()` properly. Instead, we should do:

```
elem.addEventListener('click', this.handleClick.bind(this));
```

> **Exercise: Extracting a method**
>
> `exercises/single-objects/method_extraction_exrc.mjs`

25.4.5 `this` pitfall: accidentally shadowing `this`

> **Accidentally shadowing `this` is only an issue with ordinary functions**
>
> Arrow functions don't shadow `this`.

Consider the following problem: when you are inside an ordinary function, you can't access the `this` of the surrounding scope because the ordinary function has its own `this`. In other words, a variable in an inner scope hides a variable in an outer scope. That is called *shadowing*. The following code is an example:

```
const prefixer = {
  prefix: '==> ',
  prefixStringArray(stringArray) {
    return stringArray.map(
      function (x) {
        return this.prefix + x; // (A)
      });
  },
};
```

```
assert.throws(
  () => prefixer.prefixStringArray(['a', 'b']),
  /^TypeError: Cannot read property 'prefix' of undefined$/);
```

In line A, 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. The value of the former `this` is `undefined` due to the callback being function-called. That explains the error message.

The simplest way to fix this problem is via an arrow function, which doesn't have its own `this` and therefore doesn't shadow anything:

```
const prefixer = {
  prefix: '==> ',
  prefixStringArray(stringArray) {
    return stringArray.map(
      (x) => {
        return this.prefix + x;
      });
  },
};
assert.deepEqual(
  prefixer.prefixStringArray(['a', 'b']),
  ['==> a', '==> b']);
```

We can also store `this` in a different variable (line A), so that it doesn't get shadowed:

```
prefixStringArray(stringArray) {
  const that = this; // (A)
  return stringArray.map(
    function (x) {
      return that.prefix + x;
    });
},
```

Another option is to specify a fixed `this` for the callback via `.bind()` (line A):

```
prefixStringArray(stringArray) {
  return stringArray.map(
    function (x) {
      return this.prefix + x;
    }.bind(this)); // (A)
},
```

Lastly, `.map()` lets us specify a value for `this` (line A) that it uses when invoking the callback:

```
prefixStringArray(stringArray) {
  return stringArray.map(
    function (x) {
      return this.prefix + x;
    },
```

```
    this); // (A)
  },
```

25.4.6 Avoiding the pitfalls of `this`

We have seen two big `this`-related pitfalls:

1. Extracting methods
2. Accidentally shadowing `this`

One simple rule helps avoid the second pitfall:

> "Avoid the keyword `function`": Never use ordinary functions, only arrow functions (for real functions) and method definitions.

Following this rule has two benefits:

- It prevents the second pitfall because ordinary functions are never used as real functions.
- `this` becomes easier to understand because it will only appear inside methods (never inside ordinary functions). That makes it clear that `this` is an OOP feature.

However, even though I don't use (ordinary) function *expressions* anymore, I do like function *declarations* syntactically. You can use them safely if you don't refer to `this` inside them. The static checking tool ESLint can warn you during development when you do this wrong via a built-in rule.

Alas, there is no simple way around the first pitfall: whenever you extract a method, you have to be careful and do it properly – for example, by binding `this`.

25.4.7 The value of `this` in various contexts

What is the value of `this` in various contexts?

Inside a callable entity, the value of `this` depends on how the callable entity is invoked and what kind of callable entity it is:

- Function call:
  - Ordinary functions: `this === undefined` (in strict mode)
  - Arrow functions: `this` is same as in surrounding scope (lexical `this`)
- Method call: `this` is receiver of call
- `new`: `this` refers to newly created instance

You can also access `this` in all common top-level scopes:

- `<script>` element: `this === globalThis`
- ECMAScript modules: `this === undefined`
- CommonJS modules: `this === module.exports`

However, I like to pretend that you can't access `this` in top-level scopes because top-level `this` is confusing and rarely useful.

25.5 Objects as dictionaries (advanced)

Objects work best as records. But before ES6, JavaScript did not have a data structure for dictionaries (ES6 brought Maps). Therefore, objects had to be used as dictionaries, which imposed a signficant constraint: keys had to be strings (symbols were also introduced with ES6).

We first look at features of objects that are related to dictionaries but also useful for objects-as-records. This section concludes with tips for actually using objects as dictionaries (spoiler: use Maps if you can).

25.5.1 Arbitrary fixed strings as property keys

So far, we have always used objects as records. Property keys were fixed tokens that had to be valid identifiers and internally became strings:

```
const obj = {
  mustBeAnIdentifier: 123,
};

// Get property
assert.equal(obj.mustBeAnIdentifier, 123);

// Set property
obj.mustBeAnIdentifier = 'abc';
assert.equal(obj.mustBeAnIdentifier, 'abc');
```

As a next step, we'll go beyond this limitation for property keys: In this section, we'll use arbitrary fixed strings as keys. In the next subsection, we'll dynamically compute keys.

Two techniques allow us to use arbitrary strings as property keys.

First, when creating property keys via object literals, we can quote property keys (with single or double quotes):

```
const obj = {
  'Can be any string!': 123,
};
```

Second, when getting or setting properties, we can use square brackets with strings inside them:

```
// Get property
assert.equal(obj['Can be any string!'], 123);

// Set property
obj['Can be any string!'] = 'abc';
assert.equal(obj['Can be any string!'], 'abc');
```

You can also use these techniques for methods:

```
const obj = {
  'A nice method'() {
```

```
    return 'Yes!';
  },
};

assert.equal(obj['A nice method'](), 'Yes!');
```

25.5.2 Computed property keys

So far, property keys were always fixed strings inside object literals. In this section we learn how to dynamically compute property keys. That enables us to use either arbitrary strings or symbols.

The syntax of dynamically computed property keys in object literals is inspired by dynamically accessing properties. That is, we can use square brackets to wrap expressions:

```
const obj = {
  ['Hello world!']: true,
  ['f'+'o'+'o']: 123,
  [Symbol.toStringTag]: 'Goodbye', // (A)
};

assert.equal(obj['Hello world!'], true);
assert.equal(obj.foo, 123);
assert.equal(obj[Symbol.toStringTag], 'Goodbye');
```

The main use case for computed keys is having symbols as property keys (line A).

Note that the square brackets operator for getting and setting properties works with arbitrary expressions:

```
assert.equal(obj['f'+'o'+'o'], 123);
assert.equal(obj['==> foo'.slice(-3)], 123);
```

Methods can have computed property keys, too:

```
const methodKey = Symbol();
const obj = {
  [methodKey]() {
    return 'Yes!';
  },
};

assert.equal(obj[methodKey](), 'Yes!');
```

For the remainder of this chapter, we'll mostly use fixed property keys again (because they are syntactically more convenient). But all features are also available for arbitrary strings and symbols.

Exercise: Non-destructively updating a property via spreading (computed key)

exercises/single-objects/update_property_test.mjs

25.5.3 The `in` operator: is there a property with a given key?

The `in` operator checks if an object has a property with a given key:

```
const obj = {
  foo: 'abc',
  bar: false,
};

assert.equal('foo' in obj, true);
assert.equal('unknownKey' in obj, false);
```

25.5.3.1 Checking if a property exists via truthiness

You can also use a truthiness check to determine if a property exists:

```
assert.equal(
  obj.foo ? 'exists' : 'does not exist',
  'exists');
assert.equal(
  obj.unknownKey ? 'exists' : 'does not exist',
  'does not exist');
```

The previous checks work because `obj.foo` is truthy and because reading a missing property returns `undefined` (which is falsy).

There is, however, one important caveat: truthiness checks fail if the property exists, but has a falsy value (`undefined`, `null`, `false`, `0`, `""`, etc.):

```
assert.equal(
  obj.bar ? 'exists' : 'does not exist',
  'does not exist'); // should be: 'exists'
```

25.5.4 Deleting properties

You can delete properties via the `delete` operator:

```
const obj = {
  foo: 123,
};
assert.deepEqual(Object.keys(obj), ['foo']);

delete obj.foo;

assert.deepEqual(Object.keys(obj), []);
```

25.5.5 Listing property keys

Table 25.1: Standard library methods for listing *own* (non-inherited) property keys. All of them return Arrays with strings and/or symbols.

	enumerable	non-e.	string	symbol
`Object.keys()`	✔		✔	
`Object.getOwnPropertyNames()`	✔	✔	✔	
`Object.getOwnPropertySymbols()`	✔	✔		✔
`Reflect.ownKeys()`	✔	✔	✔	✔

Each of the methods in tbl. 25.1 returns an Array with the own property keys of the parameter. In the names of the methods, you can see that the following distinction is made:

- A *property key* can be either a string or a symbol.
- A *property name* is a property key whose value is a string.
- A *property symbol* is a property key whose value is a symbol.

The next section describes the term *enumerable* and demonstrates each of the methods.

25.5.5.1 Enumerability

Enumerability is an *attribute* of a property. Non-enumerable properties are ignored by some operations – for example, by `Object.keys()` (see tbl. 25.1) and by spread properties. By default, most properties are enumerable. The next example shows how to change that. It also demonstrates the various ways of listing property keys.

```
const enumerableSymbolKey = Symbol('enumerableSymbolKey');
const nonEnumSymbolKey = Symbol('nonEnumSymbolKey');

// We create enumerable properties via an object literal
const obj = {
  enumerableStringKey: 1,
  [enumerableSymbolKey]: 2,
}

// For non-enumerable properties, we need a more powerful tool
Object.defineProperties(obj, {
  nonEnumStringKey: {
    value: 3,
    enumerable: false,
  },
  [nonEnumSymbolKey]: {
    value: 4,
    enumerable: false,
  },
});

assert.deepEqual(
```

```
    Object.keys(obj),
    [ 'enumerableStringKey' ]);
  assert.deepEqual(
    Object.getOwnPropertyNames(obj),
    [ 'enumerableStringKey', 'nonEnumStringKey' ]);
  assert.deepEqual(
    Object.getOwnPropertySymbols(obj),
    [ enumerableSymbolKey, nonEnumSymbolKey ]);
  assert.deepEqual(
    Reflect.ownKeys(obj),
    [
      'enumerableStringKey', 'nonEnumStringKey',
      enumerableSymbolKey, nonEnumSymbolKey,
    ]);
```

`Object.defineProperties()` is explained later in this chapter.

25.5.6 Listing property values via `Object.values()`

`Object.values()` lists the values of all enumerable properties of an object:

```
const obj = {foo: 1, bar: 2};
assert.deepEqual(
  Object.values(obj),
  [1, 2]);
```

25.5.7 Listing property entries via `Object.entries()`

`Object.entries()` lists key-value pairs of enumerable properties. Each pair is encoded as a two-element Array:

```
const obj = {foo: 1, bar: 2};
assert.deepEqual(
  Object.entries(obj),
  [
    ['foo', 1],
    ['bar', 2],
  ]);
```

Exercise: `Object.entries()`

`exercises/single-objects/find_key_test.mjs`

25.5.8 Properties are listed deterministically

Own (non-inherited) properties of objects are always listed in the following order:

1. Properties with string keys that contain integer indices (that includes Array indices):
   In ascending numeric order

2. Remaining properties with string keys:
   In the order in which they were added
3. Properties with symbol keys:
   In the order in which they were added

The following example demonstrates how property keys are sorted according to these rules:

```
> Object.keys({b:0,a:0, 10:0,2:0})
[ '2', '10', 'b', 'a' ]
```

The order of properties

The ECMAScript specification describes in more detail how properties are ordered.

25.5.9 Assembling objects via `Object.fromEntries()`

Given an iterable over [key, value] pairs, `Object.fromEntries()` creates an object:

```
assert.deepEqual(
  Object.fromEntries([['foo',1], ['bar',2]]),
  {
    foo: 1,
    bar: 2,
  }
);
```

`Object.fromEntries()` does the opposite of `Object.entries()`.

To demonstrate both, we'll use them to implement two tool functions from the library Underscore in the next subsubsections.

25.5.9.1 Example: `pick(object, ...keys)`

`pick` returns a copy of `object` that only has those properties whose keys are mentioned as arguments:

```
const address = {
  street: 'Evergreen Terrace',
  number: '742',
  city: 'Springfield',
  state: 'NT',
  zip: '49007',
};
assert.deepEqual(
  pick(address, 'street', 'number'),
  {
    street: 'Evergreen Terrace',
    number: '742',
  }
);
```

We can implement `pick()` as follows:

```
function pick(object, ...keys) {
  const filteredEntries = Object.entries(object)
    .filter(([key, _value]) => keys.includes(key));
  return Object.fromEntries(filteredEntries);
}
```

25.5.9.2 Example: `invert(object)`

`invert` returns a copy of `object` where the keys and values of all properties are swapped:

```
assert.deepEqual(
  invert({a: 1, b: 2, c: 3}),
  {1: 'a', 2: 'b', 3: 'c'}
);
```

We can implement `invert()` like this:

```
function invert(object) {
  const mappedEntries = Object.entries(object)
    .map(([key, value]) => [value, key]);
  return Object.fromEntries(mappedEntries);
}
```

25.5.9.3 A simple implementation of `Object.fromEntries()`

The following function is a simplified version of `Object.fromEntries()`:

```
function fromEntries(iterable) {
  const result = {};
  for (const [key, value] of iterable) {
    let coercedKey;
    if (typeof key === 'string' || typeof key === 'symbol') {
      coercedKey = key;
    } else {
      coercedKey = String(key);
    }
    result[coercedKey] = value;
  }
  return result;
}
```

25.5.9.4 A polyfill for `Object.fromEntries()`

The npm package `object.fromentries` is a *polyfill* for `Object.entries()`: it installs its own implementation if that method doesn't exist on the current platform.

Exercise: `Object.entries()` and `Object.fromEntries()`

exercises/single-objects/omit_properties_test.mjs

25.5.10 The pitfalls of using an object as a dictionary

If you use plain objects (created via object literals) as dictionaries, you have to look out for two pitfalls.

The first pitfall is that the `in` operator also finds inherited properties:

```
const dict = {};
assert.equal('toString' in dict, true);
```

We want `dict` to be treated as empty, but the `in` operator detects the properties it inherits from its prototype, `Object.prototype`.

The second pitfall is that you can't use the property key `__proto__` because it has special powers (it sets the prototype of the object):

```
const dict = {};

dict['__proto__'] = 123;
// No property was added to dict:
assert.deepEqual(Object.keys(dict), []);
```

So how do we avoid these pitfalls?

- Whenever you can, use Maps. They are the best solution for dictionaries.
- If you can't, use a library for objects-as-dictionaries that does everything safely.
- If you can't, use an object without a prototype.

The following code demonstrates using objects without prototypes as dictionaries:

```
const dict = Object.create(null); // no prototype

assert.equal('toString' in dict, false); // (A)

dict['__proto__'] = 123;
assert.deepEqual(Object.keys(dict), ['__proto__']);
```

We avoided both pitfalls: First, a property without a prototype does not inherit any properties (line A). Second, in modern JavaScript, `__proto__` is implemented via `Object.prototype`. That means that it is switched off if `Object.prototype` is not in the prototype chain.

Exercise: Using an object as a dictionary

exercises/single-objects/simple_dict_test.mjs

25.6 Standard methods (advanced)

`Object.prototype` defines several standard methods that can be overridden to configure how an object is treated by the language. Two important ones are:

- `.toString()`
- `.valueOf()`

25.6.1 `.toString()`

`.toString()` determines how objects are converted to strings:

```
> String({toString() { return 'Hello!' }})
'Hello!'
> String({})
'[object Object]'
```

25.6.2 `.valueOf()`

`.valueOf()` determines how objects are converted to numbers:

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

25.7 Advanced topics

The following subsections give brief overviews of a few advanced topics.

25.7.1 `Object.assign()`

`Object.assign()` is a tool method:

```
Object.assign(target, source_1, source_2, ···)
```

This expression assigns all properties of `source_1` to `target`, then all properties of `source_2`, etc. At the end, it returns `target` – for example:

```
const target = { foo: 1 };

const result = Object.assign(
  target,
  {bar: 2},
  {baz: 3, bar: 4});

assert.deepEqual(
  result, { foo: 1, bar: 4, baz: 3 });
// target was modified and returned:
assert.equal(result, target);
```

The use cases for `Object.assign()` are similar to those for spread properties. In a way, it spreads destructively.

25.7.2 Freezing objects

`Object.freeze(obj)` makes `obj` completely immutable: You can't change properties, add properties, or change its prototype – for example:

```
const frozen = Object.freeze({ x: 2, y: 5 });
assert.throws(
  () => { frozen.x = 7 },
  {
    name: 'TypeError',
    message: /^Cannot assign to read only property 'x'/,
  });
```

There is one caveat: `Object.freeze(obj)` freezes shallowly. That is, only the properties of `obj` are frozen but not objects stored in properties.

25.7.3 Property attributes and property descriptors

Just as objects are composed of properties, properties are composed of *attributes*. The value of a property is only one of several attributes. Others include:

- `writable`: Is it possible to change the value of the property?
- `enumerable`: Is the property considered by `Object.keys()`, spreading, etc.?

When you are using one of the operations for handling property attributes, attributes are specified via *property descriptors*: objects where each property represents one attribute. For example, this is how you read the attributes of a property `obj.foo`:

```
const obj = { foo: 123 };
assert.deepEqual(
  Object.getOwnPropertyDescriptor(obj, 'foo'),
  {
    value: 123,
    writable: true,
    enumerable: true,
    configurable: true,
  });
```

And this is how you set the attributes of a property `obj.bar`:

```
const obj = {
  foo: 1,
  bar: 2,
};

assert.deepEqual(Object.keys(obj), ['foo', 'bar']);

// Hide property `bar` from Object.keys()
Object.defineProperty(obj, 'bar', {
```

```
  enumerable: false,
});

assert.deepEqual(Object.keys(obj), ['foo']);
```

Enumerability is covered in greater detail earlier in this chapter. For more information on property attributes and property descriptors, consult *Speaking JavaScript*.

Quiz

See quiz app.

Chapter 26

Prototype chains and classes

Contents

In this book, JavaScript's style of object-oriented programming (OOP) is introduced in four steps. This chapter covers steps 2–4, the previous chapter covers step 1. The steps are (fig. 26.1):

1. **Single objects (previous chapter):** How do *objects*, JavaScript's basic OOP building blocks, work in isolation?
2. **Prototype chains (this chapter):** Each object has a chain of zero or more *prototype objects*. Prototypes are JavaScript's core inheritance mechanism.
3. **Classes (this chapter):** JavaScript's *classes* are factories for objects. The relationship between a class and its instances is based on prototypal inheritance.
4. **Subclassing (this chapter):** The relationship between a *subclass* and its *superclass* is also based on prototypal inheritance.

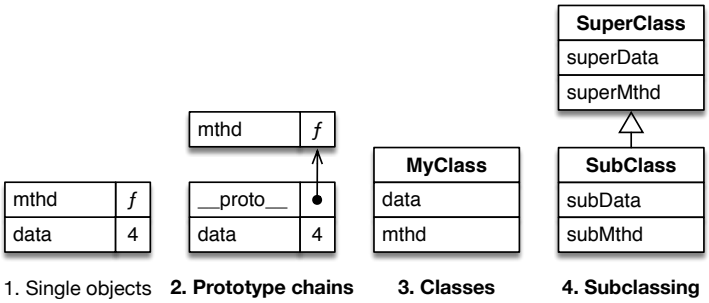

Figure 26.1: This book introduces object-oriented programming in JavaScript in four steps.

26.1 Prototype chains

Prototypes are JavaScript's only inheritance mechanism: each object has a prototype that is either `null` or an object. In the latter case, the object inherits all of the prototype's properties.

In an object literal, you can set the prototype via the special property `__proto__`:

```
const proto = {
  protoProp: 'a',
};
const obj = {
  __proto__: proto,
  objProp: 'b',
};

// obj inherits .protoProp:
assert.equal(obj.protoProp, 'a');
assert.equal('protoProp' in obj, true);
```

Given that a prototype object can have a prototype itself, we get a chain of objects – the so-called *prototype chain*. That means that inheritance gives us the impression that we are dealing with single objects, but we are actually dealing with chains of objects.

Fig. 26.2 shows what the prototype chain of `obj` looks like.

Non-inherited properties are called *own properties*. `obj` has one own property, `.objProp`.

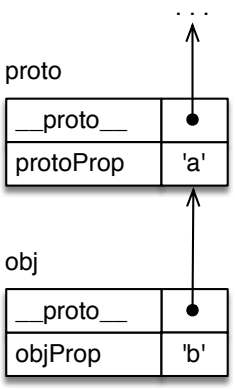

Figure 26.2: `obj` starts a chain of objects that continues with `proto` and other objects.

26.1.1 JavaScript's operations: all properties vs. own properties

Some operations consider all properties (own and inherited) – for example, getting properties:

```
> const obj = { foo: 1 };
> typeof obj.foo // own
'number'
> typeof obj.toString // inherited
'function'
```

Other operations only consider own properties – for example, `Object.keys()`:

```
> Object.keys(obj)
[ 'foo' ]
```

Read on for another operation that also only considers own properties: setting properties.

26.1.2 Pitfall: only the first member of a prototype chain is mutated

One aspect of prototype chains that may be counter-intuitive is that setting *any* property via an object – even an inherited one – only changes that very object – never one of the prototypes.

Consider the following object `obj`:

```
const proto = {
  protoProp: 'a',
};
const obj = {
  __proto__: proto,
  objProp: 'b',
};
```

In the next code snippet, we set the inherited property `obj.protoProp` (line A). That

"changes" it by creating an own property: When reading `obj.protoProp`, the own property is found first and its value *overrides* the value of the inherited property.

```
// In the beginning, obj has one own property
assert.deepEqual(Object.keys(obj), ['objProp']);

obj.protoProp = 'x'; // (A)

// We created a new own property:
assert.deepEqual(Object.keys(obj), ['objProp', 'protoProp']);

// The inherited property itself is unchanged:
assert.equal(proto.protoProp, 'a');

// The own property overrides the inherited property:
assert.equal(obj.protoProp, 'x');
```

The prototype chain of `obj` is depicted in fig. 26.3.

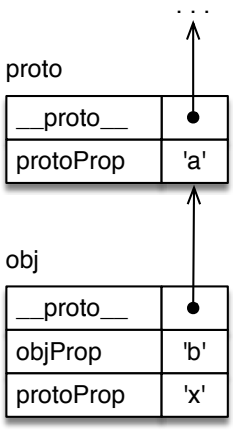

Figure 26.3: The own property `.protoProp` of `obj` overrides the property inherited from `proto`.

26.1.3 Tips for working with prototypes (advanced)

26.1.3.1 Best practice: avoid `__proto__`, except in object literals

I recommend to avoid the pseudo-property `__proto__`: As we will see later, not all objects have it.

However, `__proto__` in object literals is different. There, it is a built-in feature and always available.

The recommended ways of getting and setting prototypes are:

- The best way to get a prototype is via the following method:

```
Object.getPrototypeOf(obj: Object) : Object
```

- The best way to set a prototype is when creating an object – via `__proto__` in an object literal or via:

  ```
  Object.create(proto: Object) : Object
  ```

  If you have to, you can use `Object.setPrototypeOf()` to change the prototype of an existing object. But that may affect performance negatively.

This is how these features are used:

```
const proto1 = {};
const proto2 = {};

const obj = Object.create(proto1);
assert.equal(Object.getPrototypeOf(obj), proto1);

Object.setPrototypeOf(obj, proto2);
assert.equal(Object.getPrototypeOf(obj), proto2);
```

26.1.3.2 Check: is an object a prototype of another one?

So far, "`p` is a prototype of `o`" always meant "`p` is a *direct* prototype of `o`". But it can also be used more loosely and mean that `p` is in the prototype chain of `o`. That looser relationship can be checked via:

```
p.isPrototypeOf(o)
```

For example:

```
const a = {};
const b = {__proto__: a};
const c = {__proto__: b};

assert.equal(a.isPrototypeOf(b), true);
assert.equal(a.isPrototypeOf(c), true);

assert.equal(a.isPrototypeOf(a), false);
assert.equal(c.isPrototypeOf(a), false);
```

26.1.4 Sharing data via prototypes

Consider the following code:

```
const jane = {
  name: 'Jane',
  describe() {
    return 'Person named '+this.name;
  },
};
const tarzan = {
  name: 'Tarzan',
```

```
  describe() {
    return 'Person named '+this.name;
  },
};

assert.equal(jane.describe(), 'Person named Jane');
assert.equal(tarzan.describe(), 'Person named Tarzan');
```

We have two objects that are very similar. Both have two properties whose names are `.name` and `.describe`. Additionally, method `.describe()` is the same. How can we avoid duplicating that method?

We can move it to an object `PersonProto` and make that object a prototype of both `jane` and `tarzan`:

```
const PersonProto = {
  describe() {
    return 'Person named ' + this.name;
  },
};
const jane = {
  __proto__: PersonProto,
  name: 'Jane',
};
const tarzan = {
  __proto__: PersonProto,
  name: 'Tarzan',
};
```

The name of the prototype reflects that both `jane` and `tarzan` are persons.

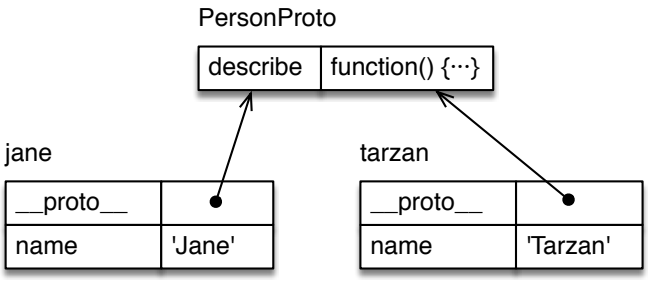

Figure 26.4: Objects `jane` and `tarzan` share method `.describe()`, via their common prototype `PersonProto`.

Fig. 26.4 illustrates how the three objects are connected: The objects at the bottom now contain the properties that are specific to `jane` and `tarzan`. The object at the top contains the properties that are shared between them.

When you make the method call `jane.describe()`, `this` points to the receiver of that method call, `jane` (in the bottom-left corner of the diagram). That's why the method still works. `tarzan.describe()` works similarly.

```
assert.equal(jane.describe(), 'Person named Jane');
assert.equal(tarzan.describe(), 'Person named Tarzan');
```

26.2 Classes

We are now ready to take on classes, which are basically a compact syntax for setting up prototype chains. Under the hood, JavaScript's classes are unconventional. But that is something you rarely see when working with them. They should normally feel familiar to people who have used other object-oriented programming languages.

26.2.1 A class for persons

We have previously worked with `jane` and `tarzan`, single objects representing persons. Let's use a *class declaration* to implement a factory for person objects:

```
class Person {
  constructor(name) {
    this.name = name;
  }
  describe() {
    return 'Person named '+this.name;
  }
}
```

`jane` and `tarzan` can now be created via `new Person()`:

```
const jane = new Person('Jane');
assert.equal(jane.name, 'Jane');
assert.equal(jane.describe(), 'Person named Jane');

const tarzan = new Person('Tarzan');
assert.equal(tarzan.name, 'Tarzan');
assert.equal(tarzan.describe(), 'Person named Tarzan');
```

Class `Person` has two methods:

- The normal method `.describe()`
- The special method `.constructor()` which is called directly after a new instance has been created and initializes that instance. It receives the arguments that are passed to the `new` operator (after the class name). If you don't need any arguments to set up a new instance, you can omit the constructor.

26.2.1.1 Class expressions

There are two kinds of *class definitions* (ways of defining classes):

- *Class declarations*, which we have seen in the previous section.
- *Class expressions*, which we'll see next.

Class expressions can be anonymous and named:

```
// Anonymous class expression
const Person = class { ··· };

// Named class expression
const Person = class MyClass { ··· };
```

The name of a named class expression works similarly to the name of a named function expression.

This was a first look at classes. We'll explore more features soon, but first we need to learn the internals of classes.

26.2.2 Classes under the hood

There is a lot going on under the hood of classes. Let's look at the diagram for `jane` (fig. 26.5).

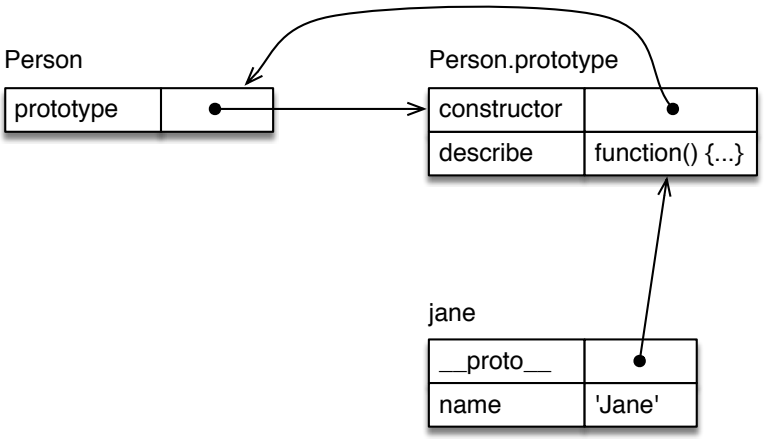

Figure 26.5: The class `Person` has the property `.prototype` that points to an object that is the prototype of all instances of `Person`. `jane` is one such instance.

The main purpose of class `Person` is to set up the prototype chain on the right (`jane`, followed by `Person.prototype`). It is interesting to note that both constructs inside class `Person` (`.constructor` and `.describe()`) created properties for `Person.prototype`, not for `Person`.

The reason for this slightly odd approach is backward compatibility: prior to classes, *constructor functions* (ordinary functions, invoked via the `new` operator) were often used as factories for objects. Classes are mostly better syntax for constructor functions and therefore remain compatible with old code. That explains why classes are functions:

```
> typeof Person
'function'
```

In this book, I use the terms *constructor (function)* and *class* interchangeably.

It is easy to confuse `.__proto__` and `.prototype`. Hopefully, fig. 26.5 makes it clear how they differ:

- `.__proto__` is a pseudo-property for accessing the prototype of an object.
- `.prototype` is a normal property that is only special due to how the `new` operator uses it. The name is not ideal: `Person.prototype` does not point to the prototype of `Person`, it points to the prototype of all instances of `Person`.

26.2.2.1 `Person.prototype.constructor` (advanced)

There is one detail in fig. 26.5 that we haven't looked at, yet: `Person.prototype.constructor` points back to `Person`:

```
> Person.prototype.constructor === Person
true
```

This setup also exists due to backward compatibility. But it has two additional benefits.

First, each instance of a class inherits property `.constructor`. Therefore, given an instance, you can make "similar" objects using it:

```
const jane = new Person('Jane');

const cheeta = new jane.constructor('Cheeta');
// cheeta is also an instance of Person
// (the instanceof operator is explained later)
assert.equal(cheeta instanceof Person, true);
```

Second, you can get the name of the class that created a given instance:

```
const tarzan = new Person('Tarzan');

assert.equal(tarzan.constructor.name, 'Person');
```

26.2.3 Class definitions: prototype properties

All constructs in the body of the following class declaration create properties of `Foo.prototype`.

```
class Foo {
  constructor(prop) {
    this.prop = prop;
  }
  protoMethod() {
    return 'protoMethod';
  }
  get protoGetter() {
    return 'protoGetter';
  }
}
```

Let's examine them in order:

- `.constructor()` is called after creating a new instance of `Foo` to set up that instance.
- `.protoMethod()` is a normal method. It is stored in `Foo.prototype`.
- `.protoGetter` is a getter that is stored in `Foo.prototype`.

The following interaction uses class `Foo`:

```
> const foo = new Foo(123);
> foo.prop
123

> foo.protoMethod()
'protoMethod'
> foo.protoGetter
'protoGetter'
```

26.2.4 Class definitions: static properties

All constructs in the body of the following class declaration create so-called *static* properties – properties of `Bar` itself.

```
class Bar {
  static staticMethod() {
    return 'staticMethod';
  }
  static get staticGetter() {
    return 'staticGetter';
  }
}
```

The static method and the static getter are used as follows:

```
> Bar.staticMethod()
'staticMethod'
> Bar.staticGetter
'staticGetter'
```

26.2.5 The `instanceof` operator

The `instanceof` operator tells you if a value is an instance of a given class:

```
> new Person('Jane') instanceof Person
true
> ({}) instanceof Person
false
> ({}) instanceof Object
true
> [] instanceof Array
true
```

We'll explore the `instanceof` operator in more detail later, after we have looked at subclassing.

26.2.6 Why I recommend classes

I recommend using classes for the following reasons:

- Classes are a common standard for object creation and inheritance that is now widely supported across frameworks (React, Angular, Ember, etc.). This is an improvement to how things were before, when almost every framework had its own inheritance library.
- They help tools such as IDEs and type checkers with their work and enable new features there.
- If you come from another language to JavaScript and are used to classes, then you can get started more quickly.
- JavaScript engines optimize them. That is, code that uses classes is almost always faster than code that uses a custom inheritance library.
- You can subclass built-in constructor functions such as `Error`.

That doesn't mean that classes are perfect:

- There is a risk of overdoing inheritance.
- There is a risk of putting too much functionality in classes (when some of it is often better put in functions).
- How they work superficially and under the hood is quite different. In other words, there is a disconnect between syntax and semantics. Two examples are:
  - A method definition inside a class `C` creates a method in the object `C.prototype`.
  - Classes are functions.

  The motivation for the disconnect is backward compatibility. Thankfully, the disconnect causes few problems in practice; you are usually OK if you go along with what classes pretend to be.

Exercise: Writing a class

`exercises/proto-chains-classes/point_class_test.mjs`

26.3 Private data for classes

This section describes techniques for hiding some of the data of an object from the outside. We discuss them in the context of classes, but they also work for objects created directly, e.g., via object literals.

26.3.1 Private data: naming convention

The first technique makes a property private by prefixing its name with an underscore. This doesn't protect the property in any way; it merely signals to the outside: "You don't need to know about this property."

In the following code, the properties `._counter` and `._action` are private.

```
class Countdown {
  constructor(counter, action) {
    this._counter = counter;
    this._action = action;
  }
  dec() {
    this._counter--;
    if (this._counter === 0) {
      this._action();
    }
  }
}

// The two properties aren't really private:
assert.deepEqual(
  Object.keys(new Countdown()),
  ['_counter', '_action']);
```

With this technique, you don't get any protection and private names can clash. On the plus side, it is easy to use.

26.3.2 Private data: WeakMaps

Another technique is to use WeakMaps. How exactly that works is explained in the chapter on WeakMaps. This is a preview:

```
const _counter = new WeakMap();
const _action = new WeakMap();

class Countdown {
  constructor(counter, action) {
    _counter.set(this, counter);
    _action.set(this, action);
  }
  dec() {
    let counter = _counter.get(this);
    counter--;
    _counter.set(this, counter);
    if (counter === 0) {
      _action.get(this)();
    }
  }
}

// The two pseudo-properties are truly private:
assert.deepEqual(
  Object.keys(new Countdown()),
  []);
```

This technique offers you considerable protection from outside access and there can't be

any name clashes. But it is also more complicated to use.

26.3.3 More techniques for private data

This book explains the most important techniques for private data in classes. There will also probably soon be built-in support for it. Consult the ECMAScript proposal "Class Public Instance Fields & Private Instance Fields" for details.

A few additional techniques are explained in *Exploring ES6*.

26.4 Subclassing

Classes can also subclass ("extend") existing classes. As an example, the following class `Employee` subclasses `Person`:

```
class Person {
  constructor(name) {
    this.name = name;
  }
  describe() {
    return `Person named ${this.name}`;
  }
  static logNames(persons) {
    for (const person of persons) {
      console.log(person.name);
    }
  }
}

class Employee extends Person {
  constructor(name, title) {
    super(name);
    this.title = title;
  }
  describe() {
    return super.describe() +
      ` (${this.title})`;
  }
}

const jane = new Employee('Jane', 'CTO');
assert.equal(
  jane.describe(),
  'Person named Jane (CTO)');
```

Two comments:

- Inside a `.constructor()` method, you must call the super-constructor via `super()` before you can access `this`. That's because `this` doesn't exist before the super-constructor is called (this phenomenon is specific to classes).

- Static methods are also inherited. For example, `Employee` inherits the static method `.logNames()`:

```
> 'logNames' in Employee
true
```

Exercise: Subclassing

`exercises/proto-chains-classes/color_point_class_test.mjs`

26.4.1 Subclasses under the hood (advanced)

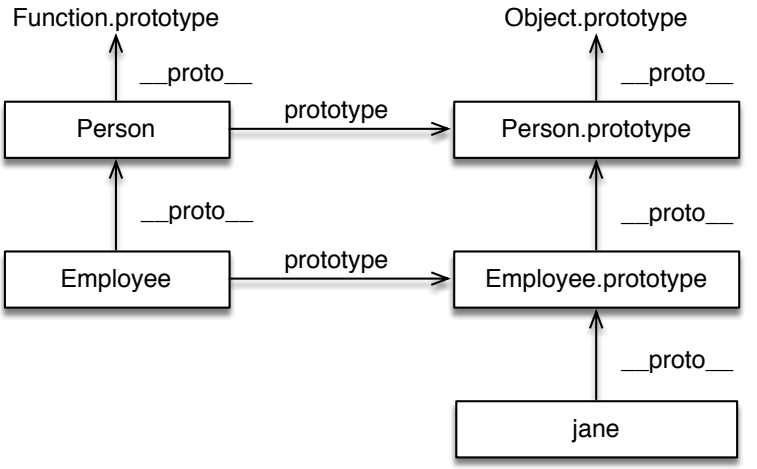

Figure 26.6: These are the objects that make up class `Person` and its subclass, `Employee`. The left column is about classes. The right column is about the `Employee` instance `jane` and its prototype chain.

The classes `Person` and `Employee` from the previous section are made up of several objects (fig. 26.6). One key insight for understanding how these objects are related is that there are two prototype chains:

- The instance prototype chain, on the right.
- The class prototype chain, on the left.

26.4.1.1 The instance prototype chain (right column)

The instance prototype chain starts with `jane` and continues with `Employee.prototype` and `Person.prototype`. In principle, the prototype chain ends at this point, but we get one more object: `Object.prototype`. This prototype provides services to virtually all objects, which is why it is included here, too:

```
> Object.getPrototypeOf(Person.prototype) === Object.prototype
true
```

26.4.1.2 The class prototype chain (left column)

In the class prototype chain, `Employee` comes first, `Person` next. Afterward, the chain continues with `Function.prototype`, which is only there because `Person` is a function and functions need the services of `Function.prototype`.

```
> Object.getPrototypeOf(Person) === Function.prototype
true
```

26.4.2 `instanceof` in more detail (advanced)

We have not yet seen how `instanceof` really works. Given the expression:

```
x instanceof C
```

How does `instanceof` determine if `x` is an instance of `C` (or a subclass of `C`)? It does so by checking if `C.prototype` is in the prototype chain of `x`. That is, the following expression is equivalent:

```
C.prototype.isPrototypeOf(x)
```

If we go back to fig. 26.6, we can confirm that the prototype chain does lead us to the following correct answers:

```
> jane instanceof Employee
true
> jane instanceof Person
true
> jane instanceof Object
true
```

26.4.3 Prototype chains of built-in objects (advanced)

Next, we'll use our knowledge of subclassing to understand the prototype chains of a few built-in objects. The following tool function `p()` helps us with our explorations.

```
const p = Object.getPrototypeOf.bind(Object);
```

We extracted method `.getPrototypeOf()` of `Object` and assigned it to `p`.

26.4.3.1 The prototype chain of `{}`

Let's start by examining plain objects:

```
> p({}) === Object.prototype
true
> p(p({})) === null
true
```

Fig. 26.7 shows a diagram for this prototype chain. We can see that `{}` really is an instance of `Object` – `Object.prototype` is in its prototype chain.

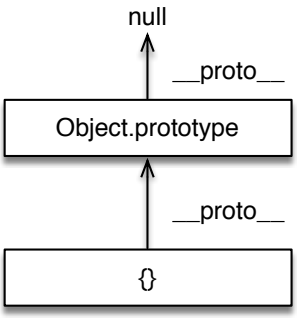

Figure 26.7: The prototype chain of an object created via an object literal starts with that object, continues with `Object.prototype`, and ends with `null`.

26.4.3.2 The prototype chain of `[]`

What does the prototype chain of an Array look like?

```
> p([]) === Array.prototype
true
> p(p([])) === Object.prototype
true
> p(p(p([]))) === null
true
```

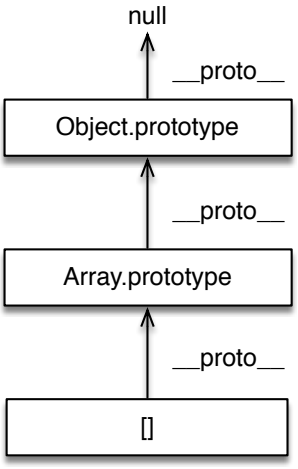

Figure 26.8: The prototype chain of an Array has these members: the Array instance, `Array.prototype`, `Object.prototype`, `null`.

This prototype chain (visualized in fig. 26.8) tells us that an Array object is an instance of `Array`, which is a subclass of `Object`.

26.4.3.3 The prototype chain of `function () {}`

Lastly, the prototype chain of an ordinary function tells us that all functions are objects:

```
> p(function () {}) === Function.prototype
true
> p(p(function () {})) === Object.prototype
true
```

26.4.3.4 Objects that aren't instances of `Object`

An object is only an instance of `Object` if `Object.prototype` is in its prototype chain. Most objects created via various literals are instances of `Object`:

```
> ({}) instanceof Object
true
> (() => {}) instanceof Object
true
> /abc/ug instanceof Object
true
```

Objects that don't have prototypes are not instances of `Object`:

```
> ({ __proto__: null }) instanceof Object
false
```

`Object.prototype` ends most prototype chains. Its prototype is `null`, which means it isn't an instance of `Object` either:

```
> Object.prototype instanceof Object
false
```

26.4.3.5 How exactly does the pseudo-property `.__proto__` work?

The pseudo-property `.__proto__` is implemented by class `Object` via a getter and a setter. It could be implemented like this:

```
class Object {
  get __proto__() {
    return Object.getPrototypeOf(this);
  }
  set __proto__(other) {
    Object.setPrototypeOf(this, other);
  }
  // ···
}
```

That means that you can switch `.__proto__` off by creating an object that doesn't have `Object.prototype` in its prototype chain (see the previous section):

```
> '__proto__' in {}
true
> '__proto__' in { __proto__: null }
false
```

26.4.4 Dispatched vs. direct method calls (advanced)

Let's examine how method calls work with classes. We are revisiting `jane` from earlier:

```
class Person {
  constructor(name) {
    this.name = name;
  }
  describe() {
    return 'Person named '+this.name;
  }
}
const jane = new Person('Jane');
```

Fig. 26.9 has a diagram with `jane`'s prototype chain.

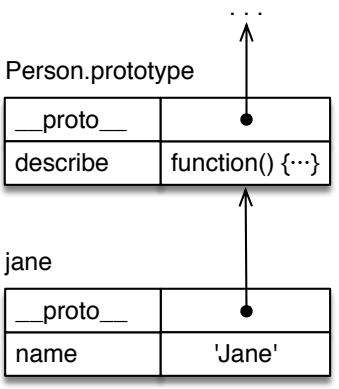

Figure 26.9: The prototype chain of `jane` starts with `jane` and continues with `Person.prototype`.

Normal method calls are *dispatched* – the method call `jane.describe()` happens in two steps:

- Dispatch: In the prototype chain of `jane`, find the first property whose key is `'describe'` and retrieve its value.

  ```
  const func = jane.describe;
  ```

- Call: Call the value, while setting `this` to `jane`.

  ```
  func.call(jane);
  ```

This way of dynamically looking for a method and invoking it is called *dynamic dispatch*.

You can make the same method call *directly*, without dispatching:

```
Person.prototype.describe.call(jane)
```

This time, we directly point to the method via `Person.prototype.describe` and don't search for it in the prototype chain. We also specify `this` differently via `.call()`.

Note that `this` always points to the beginning of a prototype chain. That enables `.describe()` to access `.name`.

26.4.4.1 Borrowing methods

Direct method calls become useful when you are working with methods of `Object.prototype`. For example, `Object.prototype.hasOwnProperty(k)` checks if `this` has a non-inherited property whose key is `k`:

```
> const obj = { foo: 123 };
> obj.hasOwnProperty('foo')
true
> obj.hasOwnProperty('bar')
false
```

However, in the prototype chain of an object, there may be another property with the key `'hasOwnProperty'` that overrides the method in `Object.prototype`. Then a dispatched method call doesn't work:

```
> const obj = { hasOwnProperty: true };
> obj.hasOwnProperty('bar')
TypeError: obj.hasOwnProperty is not a function
```

The workaround is to use a direct method call:

```
> Object.prototype.hasOwnProperty.call(obj, 'bar')
false
> Object.prototype.hasOwnProperty.call(obj, 'hasOwnProperty')
true
```

This kind of direct method call is often abbreviated as follows:

```
> ({}).hasOwnProperty.call(obj, 'bar')
false
> ({}).hasOwnProperty.call(obj, 'hasOwnProperty')
true
```

This pattern may seem inefficient, but most engines optimize this pattern, so performance should not be an issue.

26.4.5 Mixin classes (advanced)

JavaScript's class system only supports *single inheritance*. That is, each class can have at most one superclass. One way around this limitation is via a technique called *mixin classes* (short: *mixins*).

The idea is as follows: Let's say we want a class `C` to inherit from two superclasses `S1` and `S2`. That would be *multiple inheritance*, which JavaScript doesn't support.

Our workaround is to turn `S1` and `S2` into *mixins*, factories for subclasses:

```
const S1 = (Sup) => class extends Sup { /*···*/ };
const S2 = (Sup) => class extends Sup { /*···*/ };
```

Each of these two functions returns a class that extends a given superclass `Sup`. We create class `C` as follows:

```
class C extends S2(S1(Object)) {
  /*···*/
}
```

We now have a class `C` that extends a class `S2` that extends a class `S1` that extends `Object` (which most classes do implicitly).

26.4.5.1 Example: a mixin for brand management

We implement a mixin `Branded` that has helper methods for setting and getting the brand of an object:

```
const Branded = (Sup) => class extends Sup {
  setBrand(brand) {
    this._brand = brand;
    return this;
  }
  getBrand() {
    return this._brand;
  }
};
```

We use this mixin to implement brand management for a class `Car`:

```
class Car extends Branded(Object) {
  constructor(model) {
    super();
    this._model = model;
  }
  toString() {
    return `${this.getBrand()} ${this._model}`;
  }
}
```

The following code confirms that the mixin worked: `Car` has method `.setBrand()` of `Branded`.

```
const modelT = new Car('Model T').setBrand('Ford');
assert.equal(modelT.toString(), 'Ford Model T');
```

26.4.5.2 The benefits of mixins

Mixins free us from the constraints of single inheritance:

- The same class can extend a single superclass and zero or more mixins.
- The same mixin can be used by multiple classes.

26.5 FAQ: objects

26.5.1 Why do objects preserve the insertion order of properties?

In principle, objects are unordered. The main reason for ordering properties is so that operations that list entries, keys, or values are deterministic. That helps, e.g., with testing.

Quiz

See quiz app.

Chapter 27

Where are the remaining chapters?

You are reading a preview version of this book. You can either read all essential chapters online or you can buy the full version.

You can take a look at the full table of contents, which is also linked to from the book's homepage