JavaScript

FOR IMPATIENT PROGRAMMERS

Dr. Axel Rauschmayer

JavaScript for impatient programmers (beta)

Dr. Axel Rauschmayer

2019

First printing: 2019

ISBN 978-1-09-121009-7

exploringjs.com

Contents

Part I

Background

Chapter 1

About this book (ES2019 edition)

Contents

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.

1.2 What is not covered by this book?

- Some advanced language features are not explained, but references to appropriate material are provided. For example, to my other JavaScript books at `ExploringJS.com`, which are free to read online.
- This book deliberately focuses on the language. Browser-only features etc. are not included.

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 Ph.D. in Informatics from the University of Munich.

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

He is located in Munich, Germany.

1.4 Acknowledgements

- Cover by Fran Caye.
- Thanks for reviewing:
  - Johannes Weber (@jowe)

[Generated: 2019-03-31 23:20]

Chapter 2

FAQ: book

Contents

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

2.1 What should I read if I'm *really* impatient?

Do the following:

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

Then this book should be a fairly quick read.

2.2 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.

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

Bonus chapters are only available in the paid versions of this book. They are listed in the full table of contents.

2.4 How do I submit feedback and corrections?

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

2.5 How do I get updates for the downloads at Payhip?

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

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

For example, you may see:

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

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

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

The type notation is explained in detail in a blog post on 2ality.

2.7 What do the notes with icons mean?

Reading

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

Tip

Gives tips related to the current content.

Details

Provides additional details, complementing the current content.

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 frequently:

- Its code name was *Mocha*.
- In the Netscape Navigator 2.0 betas (September 1995), it was called *LiveScript*.

- In Netscape Navigator 2.0 beta 3 (December 1995), it got its final name, *JavaScript*.

3.2 Standardization

There are two standards for JavaScript:

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

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

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

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

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

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

3.3 Timeline of ECMAScript versions

This is a brief timeline of ECMAScript versions:

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

- ECMAScript 2016 (June 2016): First yearly release. That resulted in fewer new features per release – compared to ES6, which was a large upgrade.
- ECMAScript 2017 (June 2017)
- 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.

3.5 The TC39 process

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

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

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

- ECMAScript features are designed independently and go through stages, starting at 0 ("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.

For more information on the TC39 process, consult "Exploring ES2018 and ES2019".

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

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

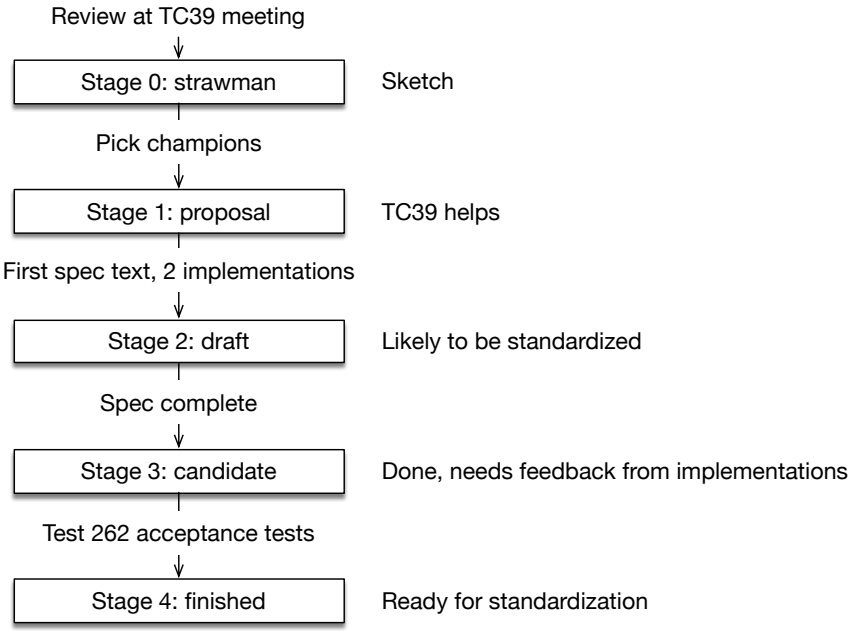

Figure 3.1: Each ECMAScript feature proposal goes through stages that are numbered from 0 to 4. *Champions* are TC39 members that support the authors of a feature. Test 262 is a suite of tests that checks JavaScript engines for compliance with the language specification.

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

3.6 FAQ: TC39 process

3.6.1 How is [my favorite proposed feature] doing?

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

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 are introduced.

3.7 Evolving JavaScript: don't break the web

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

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

- JavaScript engines become bloated: they need to support both the old and the new version. The same is true for tools such as IDEs and build tools.
- Programmers need to know, and be continually conscious of, the differences between the versions.
- You can either migrate all of an existing code base to the new version (which can be a lot of work). Or you can mix versions and refactoring becomes harder, because 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 function.

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

- New versions are always completely backward compatible (but there may occasionally be minor, hardly noticeable clean-ups).
- Old features aren't removed or fixed. Instead, better versions of them are introduced. One example is declaring variables via `let` – which is an improved version of `var`.

- If aspects of the language are changed, it is done so inside new syntactic constructs. That is, you opt in implicitly. For example, `yield` is only a keyword inside generators (which were introduced in ES6). And all code inside modules and classes (both introduced in ES6) is implicitly in strict mode.

For more information on One JavaScript, consult "Exploring ES6".

Quiz

See quiz app.

Chapter 4

FAQ: JavaScript

Contents

4.1 What are good references for JavaScript?

Please consult the section on "JavaScript references".

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

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

- ECMAScript compatibility tables for various engines 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 (which includes 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:

- Sect. "The TC39 process" describes how upcoming features are planned.
- Sect. "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
```

Why is that?

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

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

The chapter on the history and evolution of JavaScript has a section that answers this question.

Part II

First steps

Chapter 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 browsers
- Node.js

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

- You can use it to write server-side software in JavaScript.
- You can also use it to write software for the command line (think Unix shell, Windows PowerShell, etc.). Many JavaScript-related tools are based on (and executed via) Node.js.
- Node's 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

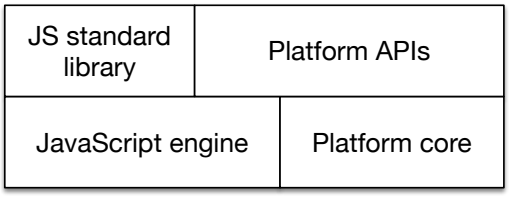

Figure 5.1: The structure of the two JavaScript platforms *web browser* and *Node.js*.

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

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

5.3 Trying out JavaScript code

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

5.3.1 Browser consoles

The consoles of browsers also let you input code. They are JavaScript command lines. How to open the console differs from browser to browser. Fig. 5.2 shows the console of Google Chrome.

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

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

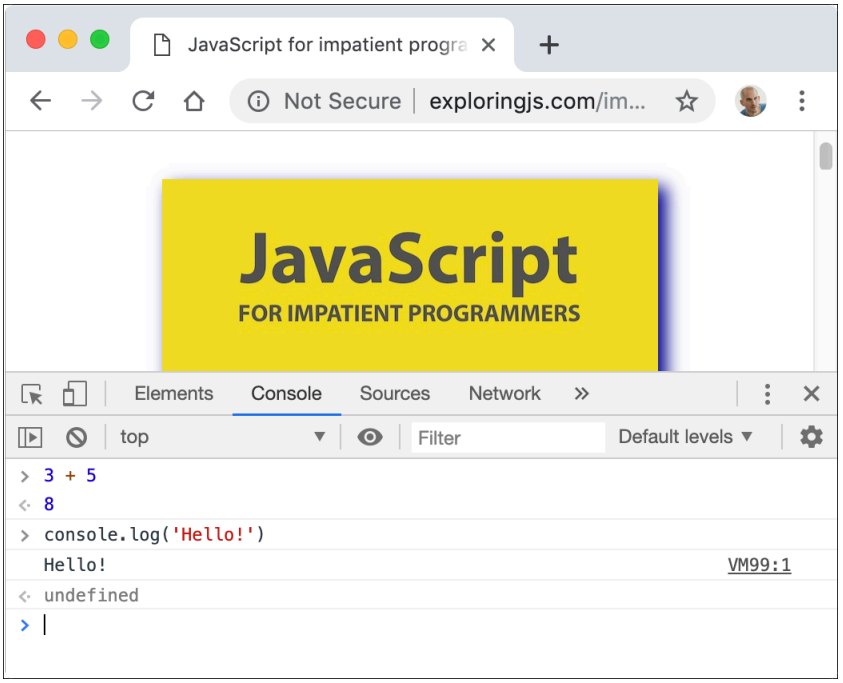

Figure 5.2: The console of the web browser "Google Chrome" is open while visiting a web page.

5.3.2 The Node.js REPL

REPL stands for *read-eval-print loop* and basically means *command line*. To use it, you must first start Node.js from an operating system command line, via the command `node`. Then an interaction with it looks as depicted in fig. 5.3: The text after `>` is input from the user; everything else is output from Node.js.

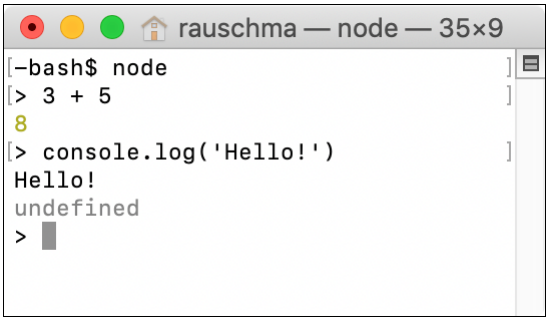

Figure 5.3: Starting and using the Node.js REPL (interactive command line).

Reading: REPL interactions

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

```
> 3 + 5
8
```

5.3.3 Other options

Other options include:

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

5.4 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.
- Exploring JS: contains my JavaScript books.
- Node.js Docs: document the Node.js API.

5.5 Further reading

- The chapter "Next steps" at the end of this book, 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 one type for numbers)
-123
1.141

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

Checking and logging to the console:

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

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

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

Declaring variables:

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

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

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

Control flow statements:

```
// Conditional statement
if (x < 0) { // is x less than zero?
  x = -x;
}
```

Ordinary function declarations:

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

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

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

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

Objects:

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

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

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

Arrays (Arrays are also objects):

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

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

6.1.2 Modules

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

```
file-tools.js
main.js
```

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

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

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

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

6.1.3 Legal variable and property names

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

Identifiers are allowed to have the following characters:

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

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

Reserved words can't be used as variable names:

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

But they are allowed as names of properties:

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

6.1.4 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: `MyConstructor`
- Constants: `MY_CONSTANT`
  - Constants are also often written in camel case: `myConstant`

6.1.6 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. They are:

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

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

> `enum implements package protected interface private public`

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

> `Infinity NaN undefined async`

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

6.4 Statement vs. expression

In this section, we explore how JavaScript distinguishes two kinds of syntactic constructs: *statements* and *expressions*. (For the sake of simplicity, we pretend that there are only statements and expressions in JavaScript.) Afterwards, we'll see that that can cause problems, because the same syntax can mean different things, depending on where it is used.

So what are statements and expressions?

First, statements are constructs that "do something". For example, `if` is a statement:

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

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

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

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

6.4.1 What is allowed where?

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

- The body of a function must be a sequence of statements:

  ```
  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('a' + 'b', max(-1, 4));
  ```

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

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

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

6.5 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')`

#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 = function () {}; // semicolon!
```

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

6.6.2 Semicolons: control statements

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

```
while (condition)
  statement
```

The body can be a single statement:

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

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

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

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

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

Your second option is an empty block:

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

6.7 Automatic semicolon insertion (ASI)

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

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

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

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

6.7.1 ASI triggered unexpectedly

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

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

```
return
{
```

```
  first: 'jane'
};
```

This code is parsed as:

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

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

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

6.7.2 ASI unexpectedly not triggered

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

Example 1: Unintended function call.

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

Parsed as:

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

Example 2: Unintended division.

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

Parsed as:

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

Example 3: Unintended property access.

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

Executed as:

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

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

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 warns about critical cases if you either require or forbid semicolons.

6.9 Strict mode

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

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

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

6.9.1 Switching on strict mode

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

```
'use strict';
```

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

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

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

6.9.2 Example: strict mode in action

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

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

Strict mode does it better:

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

Quiz: advanced

See quiz app.

Chapter 7

Printing information on the console (`console.*`)

Contents

Printing is functionality that is not part of the JavaScript language standard. However, all operations that we examine here, are supported by both browsers and Node.js.

Printing means "displaying something on the console", where "console" is either the browser console or the terminal in which you run Node.js.

The full `console.*` API is documented on MDN and on the Node.js website. We'll just take a quick look at the following two operations:

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

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

There are two variants of this operation:

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

7.1.1 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.

7.1.2 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 Printing error messages: `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.

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

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

This assertion states that the expected result of 3 plus 5 is 8. The import statement uses the recommended `strict` version 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 `mocha`. Checks inside the tests are made via methods of `assert`.

The following is an example of such a test:

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

For more information, consult the chapter on quizzes and exercises.

8.3 Normal comparison vs. deep comparison

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

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

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

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

This method works for Arrays, too:

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

8.4 Quick reference: module assert

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

8.4.1 Normal equality

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

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

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

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

8.4.2 Deep equality

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

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

  // To .equal(), an object is only equal to itself:
  assert.notEqual([], []);
  ```

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

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

8.4.3 Expecting exceptions

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

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

  ```
  assert.throws(
    () => {
      null.prop;
    }
  );
  ```

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

```
assert.throws(
  () => {
    null.prop;
  },
  TypeError
);
```

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

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

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

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

8.4.4 Other tool functions

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

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

Quiz

See quiz app.

Chapter 9

Getting started with quizzes and exercises

Contents

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

9.1 Quizzes

Installation:

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

Running the quiz app:

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

9.2 Exercises

9.2.1 Installing the exercises

To install the exercises:

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

9.2.2 Running exercises

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

9.3 Unit tests in JavaScript

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

9.3.1 A typical test

Typical test code is split into two parts:

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

Take, for example, the following two files:

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

9.3.1.1 Part 1: the code

The code itself resides in `id.js`:

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

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

9.3.1.2 Part 2: the tests

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

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

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

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

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

As for the other lines:

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

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

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

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

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

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

9.3.2 Asynchronous tests in mocha

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

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

9.3.2.1 Calling `done()`

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

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

9.3.2.2 Returning a Promise

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

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

9.3.2.3 The test is an async function

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

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

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

Part III

Variables and values

Chapter 10

Variables and assignment

Contents

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 };

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

However:

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

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

10.2.2 const and loops

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

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

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

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

10.3 Deciding between let and const

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

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

Exercise: const

`exercises/variables-assignment/const_exrc.js`

10.4 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.

```
{ // (A)
  const x = 0;
  // This scope: can access x
  assert.equal(x, 0);
  { // (B)
    const y = 1;
    // This scope: can access x, y
    assert.equal(x, 0);
    assert.equal(y, 1);
```

```
    { // (C)
      const z = 2;
      // This scope: can access x, y, z
      assert.equal(x, 0);
      assert.equal(y, 1);
      assert.equal(z, 2);
    }
  }
}
// Outside: can't access 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.
  - The variables of a scope can be accessed from all of its inner scopes.
- Scope A is an *outer scope* of scope B and scope C.

The scope of a variable declared via `const` or `let` is always the directly surrounding block. That's why these declarations are called *block-scoped*.

10.4.1 Shadowing and blocks

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

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

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

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 Temporal dead zone

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

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

Some possible approaches are:

1. The name is resolved in the scope surrounding the current scope.
2. If you read, you get `undefined`. You can also already write to the variable. (That's how `var` works.)
3. There is an error.

TC39 chose (3) for `const` and `let`, because you likely made a mistake, if you use a variable name that is declared later in the same scope. (2) would not work for `const` (where each variable should always only have the value that it is initialized with). It was therefore also rejected for `let`, 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 unitialized variable, you get a `ReferenceError`.
- Once you reach a variable declaration, the variable is set to either the value of the initializer (specified via the assignment symbol) or `undefined` – if there is no initializer.

The following code illustrates the temporal dead zone:

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

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

  tmp = 123;
  assert.equal(tmp, 123);
}
```

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

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

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

  let myVar = 3; // TDZ ends
```

```
  func(); // OK, called outside TDZ
}
```

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

10.8 Hoisting

Hoisting means that a construct is moved to the beginning of its scope, regardless of where it is located in that scope:

```
assert.equal(func(), 123); // Works!

function func() {
  return 123;
}
```

You can use `func()` before its declaration, because, internally, it is hoisted. That is, the previous code is actually executed like this:

```
function func() {
  return 123;
}

assert.equal(func(), 123);
```

The temporal dead zone can also be viewed as a form of hoisting, because the declaration affects what happens at the beginning of its scope.

More on hoisting functions

For more information on how hoisting affects functions, consult the chapter on callable values.

10.9 Global variables

A variable is global if it is declared in the top-level scope. Every nested scope can access such a variable. In JavaScript, there are multiple layers of global scopes (Fig. 10.1):

- The outermost global scope is special: its variables can be accessed via the properties of an object, the so-called *global object*. The global object is referred to by `window` and `self` in browsers. Variables in this scope are created via:
  - Properties of the global object
  - `var` and `function` at the top level of a script
- Nested in that scope is the global scope of scripts. Variables in this scope are created by `let`, `const` and `class` at the top level of a script.

- Nested in that scope are the scopes of modules. Each module has its own global scope. Variables in that scope are created by declarations at the top level of the module.

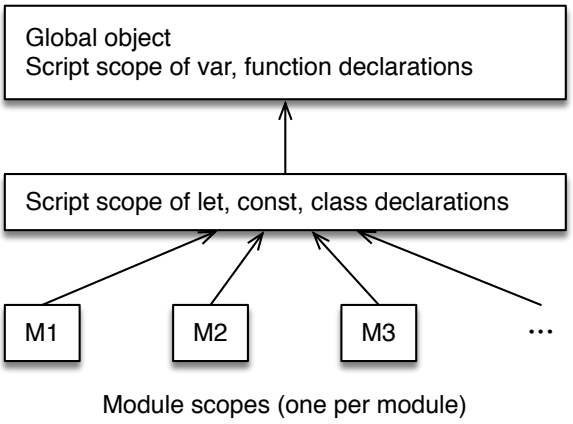

Figure 10.1: JavaScript has multiple global scopes.

10.9.1 The global object

The global object lets you access the outermost global scope via an object. The two are always in sync:

- If you create a variable in the outermost global scope, the global object gets a new property. If you change such a global variable, the property changes.
- If you create or delete a property of the global object, the corresponding global variable is created or deleted. If you change a property of the global object, the corresponding global variable changes.

The global object is available via special variables:

- `window`: is the classic way of referring to the global object. But it only works in normal browser code, not in Web Workers (processes running concurrently to normal browser code) and not in Node.js
- `self`: is available everywhere in browsers, including in Web Workers. But it isn't supported by Node.js.
- `global`: is only available in Node.js.

Let's examine how `self` works:

```
// At the top level of a script
var myGlobalVariable = 123;
assert.equal('myGlobalVariable' in self, true);

delete self.myGlobalVariable;
assert.throws(() => console.log(myGlobalVariable), ReferenceError);
```

```
// Create a global variable anywhere:
if (true) {
  self.anotherGlobalVariable = 'abc';
}
assert.equal(anotherGlobalVariable, 'abc');
```

10.9.2 Avoid the global object!

Brendan Eich called the global object one of his biggest regrets about JavaScript. It is best not to put variables into its scope:

- In general, variables that are global to all scripts on a web page, risk name clashes.
- Via the global object, you can create and delete global variables anywhere. Doing so makes code unpredictable, because it's normally not possible to make this kind of change in nested scopes.

You occasionally see `window.globalVariable` in tutorials on the web, but the prefix "`window.`" is not necessary. I prefer to omit it:

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

10.10 Closures

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

10.10.1 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);
}
```

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

10.10.2 What is a closure?

So what is a closure?

> 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 function () {
    return value;
  };
}

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

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

All functions in JavaScript are closures

Static scoping is supported via closures in JavaScript. Therefore, every function is a closure. Consult the chapter "Variable environments") if you are interested in how closures work under the hood (as described in the ECMAScript specification).

10.10.3 Example: A factory for incrementors

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

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

We can see that the function created in line A keeps its internal number in the free variable `startValue`. This time, we don't just read from the birth 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 function (step) {
    startValue += step;
    index++;
    return [index, startValue];
  };
}
const inc = createInc(5);
assert.deepEqual(inc(2), [0, 7]);
assert.deepEqual(inc(2), [1, 9]);
assert.deepEqual(inc(2), [2, 11]);
```

10.10.4 Use cases for closures

So what are closures good for?

- For starters, they are simply an implementation of static scoping. As such, they provide context data for callbacks.
- They can also be used by functions, to store state that persists across function calls. `createInc()` is an example of that.
- And they can provide private data for objects. For example:

```
function createStringBuilder() {
  let data = ''; // private data
  return {
    add(str) {
      data += str;
    },
    toString() {
      return data;
    },
  };
}
const sb = createStringBuilder();
sb.add('Hello');
sb.add(' world');
sb.add('!');
assert.equal(sb.toString(), 'Hello world!');
```

10.11 Summary: ways of declaring variables

Table 10.1: These are all the ways in which you declare variables in JavaScript.

	Hoisting	Scope	Script scope is global object?
`var`	Declaration only	Function	✔
`let`	Temporal dead zone	Block	✘
`const`	Temporal dead zone	Block	✘
`function`	Everything	Block	✔
`class`	No	Block	✘
`import`	Everything	Module	✘

Tbl. 10.1 lists all ways in which you can declare variables in JavaScript: `var`, `let`, `const`, `function`, `class` and `import`.

Quiz: advanced

See quiz app.

10.12 Further reading

For more information on how variables are handled under the hood, consult the chapter "Variable environments".

Chapter 11

Values

Contents

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

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

11.1 What's a type?

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

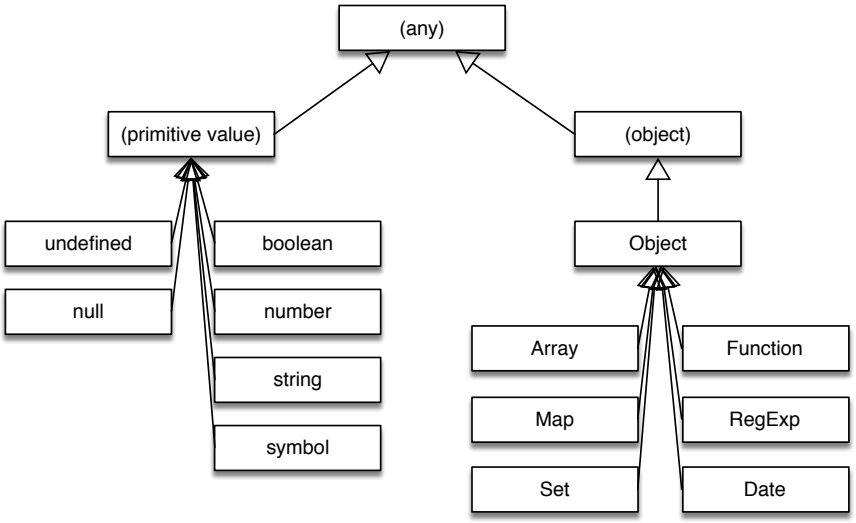

Figure 11.1: A partial hierarchy of JavaScript's types. Missing are the classes for errors, the classes associated with primitive types, and more. The diagram hints at the fact that not all objects are instances of `Object`.

11.2 JavaScript's type hierarchy

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.
- Some objects are not instances of class `Object` (as we'll see in the chapter on prototype chains and classes, an object is only an instance of `Object` if `Objec.prototype` is in its prototype chain). However, you'll rarely encounter those in practice.

11.3 The types of the language specification

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

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

11.4 Primitive values 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, it is:

- 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* (fields) and they 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 later in this book. Here, we mainly focus on how they differ from primitive values.

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

- Object literals:

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

- Array literals:

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

11.4.2.1 Objects are mutable by default

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

```
const obj = {};

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

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

11.4.2.2 Objects are *passed 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. That is also called "passing by sharing".

11.4.2.3 Objects are *compared by identity*

Objects are *compared by identity* (my term): When comparing two objects, we compare their identities.

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

11.5 Classes and constructor functions

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

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

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

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

11.5.1 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);
  ```

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

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

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

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

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

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

11.6 The operators `typeof` and `instanceof`: what's the type of a value?

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

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

So how do they differ?

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

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

11.6.1 `typeof`

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

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

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

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

Exercises: Two exercises on `typeof`

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

11.6.2 `instanceof`

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

```
x instanceof C
```

For example:

```
> (function() {}) instanceof Function
true
> ({}) instanceof Object
true
> [] instanceof Array
true
```

Primitive values are not instances of anything:

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

```
false
> '' instanceof Object
false
```

Exercise: `instanceof`

`exercises/operators/instanceof_exrc.js`

11.7 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:

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

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.js`

Quiz

See quiz app.

Chapter 12

Operators

Contents

12.1 Two important rules for operators

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

12.1.1 Operators coerce their operands to appropriate types

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

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

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

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

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

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

12.1.2 Most operators only work with primitive values

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

> Most operators only work with primitive values.

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

For example:

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

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

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

Next, it concatenates the two strings:

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

12.2 The plus operator (+)

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

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

String mode lets you use + to assemble strings:

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

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

```
> 4 + true
5
```

(`Number(true)` is 1.)

12.3 Assignment operators

12.3.1 The plain assignment operator

- `x = value`

  Assign to a previously declared variable.

- `const x = value`

  Declare and assign at the same time.

- `obj.propKey = value`

  Assign to a property.

- `arr[index] = value`

  Assign to an Array element.

12.3.2 Compound assignment operators

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

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

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

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

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

12.3.3 All compound assignment operators

- Arithmetic operators:

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

+= also works for string concatenation

- Bitwise operators:

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

12.4 Equality: == vs. ===

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

12.4.1 Lenient equality (== and !=)

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

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

Others less so:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

12.4.3 Recommendation: always use strict equality

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

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

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

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

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

I prefer either of the following two alternatives:

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

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

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

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

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

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

I prefer either of the following two alternatives:

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

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

12.5 Ordering operators

Table 12.1: JavaScript's ordering operators.

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

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

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

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

12.6 Various other operators

- Comma operator: `a, b`
- `void` operator: `void 0`

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

Quiz

See quiz app.

Part IV

Primitive values

Chapter 13

The non-values `undefined` and `null`

Contents

13.1 `undefined` vs. `null`

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

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

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

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

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

13.1.1 The history of undefined and null

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

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

Therefore, the original idea was:

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

13.2 Occurrences of undefined and null

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

13.2.1 Occurrences of undefined

Uninitialized variable `myVar`:

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

Parameter `x` is not provided:

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

Property `.unknownProp` is missing:

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

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

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

13.2.2 Occurrences of null

Last member of a prototype chain:

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

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

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

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

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

13.3 Checking for `undefined` or `null`

Checking for either:

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

Does `x` have a value?

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

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

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

Truthy means "is `true` if coerced to boolean". *Falsy* means "is `false` if coerced to boolean". Both concepts are explained properly in the chapter on booleans).

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

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

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

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

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

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

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

Table 14.1: Converting values to booleans.

x	Boolean(x)
undefined	false

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

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

14.1.1 Ways of converting to boolean

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

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

14.2 Falsy and truthy values

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

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

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

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

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

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

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

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

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

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

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

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

14.2.1 Pitfall: truthiness checks are imprecise

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

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

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

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

However, it is also skipped if:

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

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

14.2.2 Checking for truthiness or falsiness

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

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

if (x) {
```

```
  // x is truthy
} else {
  // x is falsy
}

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

The conditional operator that is used in the last line, is explained later in this chapter.

14.2.3 Use case: was a parameter provided?

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

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

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

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

An alternative is to check for `undefined`:

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

14.2.4 Use case: does a property exist?

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

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

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

If you truly want to check if the property exists, you have to use the `in` operator:

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

Exercise: Truthiness

`exercises/booleans/truthiness_exrc.js`

14.3 Conditional operator (? :)

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

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

It is evaluated as follows:

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

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

Examples:

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

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

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

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

14.4 Binary logical operators: And (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, the second operand is not evaluated. The only other operator that delays evaluating its operands is the conditional operator: Usually, all operands are evaluated before performing an operation.

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

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

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

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

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

14.4.1 Logical And (x && y)

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

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

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

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

Examples:

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

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

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

14.4.2 Logical Or (||)

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

- Evaluate `a`.
- Is the result truthy? Return it.

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

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

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

Examples:

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

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

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

14.4.3 Default values via logical Or (||)

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

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

The following code shows a real-world example:

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

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

> **Exercise: Default values via the Or operator (||)**
>
> `exercises/booleans/default_via_or_exrc.js`

14.5 Logical Not (!)

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

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

Examples:

```
> !false
true
> !true
false

> !0
true
> !123
false

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

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 still able to use real integers, with all associated performance and storage size benefits.

15.2 Number literals

Let's examine literals for numbers.

15.2.1 Integer literals

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

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

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

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

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

15.2.2 Floating point literals

Floating point numbers can only be expressed in base 10.

Fractions:

```
> 35.0
35
```

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

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

15.2.3 Syntactic pitfall: properties of integer literals

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

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

There are four ways to work around this pitfall:

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

15.3 Number operators

15.3.1 Binary arithmetic operators

\index{.x%y@x % y}

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

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

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

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

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

15.3.2 Unary plus and negation

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

Both operators coerce their operands to numbers:

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

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

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

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

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

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

Suffix `++` and suffix `--`: return and then change.

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

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

15.3.3.1 Operands: not just variables

You can also apply these operators to property values:

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

And to Array elements:

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

Exercise: Number operators

`exercises/numbers-math/is_odd_test.js`

15.4 Converting to number

Three ways of converting values to numbers:

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

Recommendation: use the descriptive `Number()`.

Examples:

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

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

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

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

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

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

Exercise: Converting to number

`exercises/numbers-math/parse_number_test.js`

15.5 Error values

Two number values are returned when errors happen:

- `NaN`
- `Infinity`

15.6 Error value: `NaN`

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

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

When is `NaN` returned?

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

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

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

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

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

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

15.6.1 Checking for `NaN`

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

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

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

```
const x = NaN;

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

15.6.2 Finding `NaN` in Arrays

Some Array methods can't find `NaN`:

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

Others can:

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

15.7 Error value: `Infinity`

When is the error value `Infinity` returned?

Infinity is returned if a number is too large:

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

Infinity is returned if there is a division by zero:

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

15.7.1 `Infinity` as a default value

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

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

```
    if (n < min) min = n;
  }
  return min;
}

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

15.7.2 Checking for `Infinity`

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

```
const x = Infinity;

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

> **Exercise: Comparing numbers**
>
> `exercises/numbers-math/find_max_test.js`

15.8 The precision of numbers: careful with decimal fractions

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

```
> 0.1 + 0.2
0.30000000000000004
> 1.3 * 3
3.9000000000000004
> 1.4 * 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 this section, we explore how JavaScript represents floating point numbers internally. It uses three numbers to do so (which take up a total of 64 bits of storage):

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

The value of a represented number is computed as follows:

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

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

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

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

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

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

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

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

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

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

15.11 Integers in JavaScript

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

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

15.11.1 Converting to integer

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

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

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

Table 15.1: Functions for converting numbers to integers.

	-2.9	-2.5	-2.1	2.1	2.5	2.9
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

15.11.2 Ranges of integers in JavaScript

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

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

15.11.3 Safe integers

Recall that this is how JavaScript represents floating point numbers:

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

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

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

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

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

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

```
assert.equal(Number.MAX_SAFE_INTEGER, (2 ** 53) - 1);
assert.equal(Number.MIN_SAFE_INTEGER, -Number.MAX_SAFE_INTEGER);
assert.equal(Number.isSafeInteger(123), true);
```

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

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

assert.equal(isSafeInteger(5), true);
assert.equal(isSafeInteger(-5), true);
assert.equal(isSafeInteger(5.1), false);

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

15.11.3.1 Safe computations

Let's look at computations involving unsafe integers.

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

```
> 9007199254740990 + 3
9007199254740992
```

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

```
> 9007199254740995 - 10
9007199254740986
```

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

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

Exercise: Safe integers

`exercises/numbers-math/is_safe_integer_test.js`

15.12 Bitwise operators

JavaScript's bitwise operators work as follows:

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

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

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

15.12.1 Binary bitwise operators

Table 15.2: Binary bitwise operators.

Operation	Name
`num1 & num2`	Bitwise And
`num1 | num2`	Bitwise Or
`num1 ^ num2`	Bitwise Xor

The binary operators (tbl. 15.2) combine the bits of their operands to produce their results:

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

15.12.2 Bitwise Not

Table 15.3: The bitwise Not operator.

Operation	Name
`~num`	Bitwise Not, ones' complement

The bitwise Not operator (tbl. 15.3) inverts each binary digit of its operand:

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

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

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

15.12.3 Bitwise shift operators

Table 15.4: Bitwise shift operators.

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

The shift operators (tbl. 15.4) move binary digits to the left or to the right:

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

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

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

15.13 Quick reference: numbers

15.13.1 Converting to number

Tbl. 15.5 shows what happens if you convert various values to numbers via `Number()`.

Table 15.5: Converting values to numbers.

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

15.13.2 Arithmetic operators

JavaScript has the following arithmetic operators:

- Binary arithmetic operators (tbl. 15.6)
- Prefix and suffix arithmetic operators (tbl. 15.7)

Table 15.6: Binary arithmetic operators.

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

Table 15.7: Prefix and suffix arithmetic operators

Operator	Name		Example
+_	Unary plus	ES1	+(-7) → -7
-_	Unary negation	ES1	-(-7) → 7
_++	Increment	ES1	let x=0; [x++, x] → [0, 1]
++_	Increment	ES1	let x=0; [++x, x] → [1, 1]
_--	Decrement	ES1	let x=1; [x--, x] → [1, 0]
--_	Decrement	ES1	let x=1; [--x, x] → [0, 0]

15.13.3 Bitwise operators

JavaScript has the following bitwise operators:

- Bitwise And, Or, Xor, Not (tbl. 15.8)
- Bitwise shift operators (tbl. 15.9)

Operands and results of bitwise operators:

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

Helper function for displaying binary numbers:

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

Table 15.8: Bitwise And, Or, Xor, Not.

Operator	Name		Example
`_ & _`	Bitwise And	ES1	`(0b1010 & 0b1100).toString(2)` → `'1000'`
`_ | _`	Bitwise Or	ES1	`(0b1010 | 0b1100).toString(2)` → `'1110'`
`_ ^ _`	Bitwise Xor	ES1	`(0b1010 ^ 0b0011).toString(2)` → `'1001'`
`~_`	Bitwise Not	ES1	`~0b11111111111111111111111111111110` → `1`

Table 15.9: Bitwise shift operators.

Operator	Name		Example
`_ << _`	Left shift	ES1	`(0b1 << 1).toString(2)` → `'10'`
`_ >> _`	Signed right shift	ES1	`bin(0b10000000000000000000000000000010 >> 1)` → `'11000000000000000000000000000001'`
`_ >>> _`	Unsigned right shift	ES1	`bin(0b10000000000000000000000000000010 >>> 1)` → `'01000000000000000000000000000001'`

15.13.4 `Number.*` data properties

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

- `Number.MAX_SAFE_INTEGER: number` [ES6]

  The largest integer that JavaScript can represent uniquely. The same as `(2 ** 53) - 1`.

- `Number.MAX_VALUE: number` [ES1]

  The largest positive finite JavaScript number.

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

- `Number.MIN_SAFE_INTEGER: number` [ES6]

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

- `Number.MIN_VALUE: number` [ES1]

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

- `Number.NaN: number` [ES1]

  The same as the global variable `NaN`.

- `Number.NEGATIVE_INFINITY: number` [ES1]

  The same as `-Number.POSITIVE_INFINITY`.

- `Number.POSITIVE_INFINITY: number` [ES1]

  The same as the global variable `Infinity`.

15.13.5 `Number.*` methods

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

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

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

  This method does not coerce its parameter to number (whereas the global function `isFinite()` does):

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

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

  Returns `true` if `num` is a number and does not have a decimal fraction.

```
> Number.isInteger(-17)
true
> Number.isInteger(33)
true
> Number.isInteger(33.1)
false
> Number.isInteger('33')
false
> Number.isInteger(NaN)
false
> Number.isInteger(Infinity)
false
```

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

  Returns `true` if `num` is the value `NaN`. In contrast to the global function `isNaN()`, this method does not coerce its parameter to number:

  ```
  > isNaN('???')
  true
  > Number.isNaN('???')
  false
  ```

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

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

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

  Coerces its parameter to string and parses it as a floating point number. Works the same as the global function `parseFloat()`. For converting strings to numbers, `Number()` (which ignores leading and trailing whitespace) is usually a better choice that `Number.parseFloat()` (which ignores leading whitespace and any illegal trailing characters and can hide problems).

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

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

  Coerces its parameter to string and parses it as an integer, ignoring illegal trailing characters. Same as the global function `parseInt()`.

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

  Do not use this method to convert numbers to integers (use the rounding methods of `Math`, instead): it is inefficient and can produce incorrect results:

```
> Number.parseInt(1e21, 10) // wrong
1
> Math.trunc(1e21) // correct
1e+21
```

15.13.6 `Number.prototype.*`

- `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()
  '1.234e+3'
  > 1234..toExponential(5)
  '1.23400e+3'
  ```

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

  ```
  > 0.003.toString()
  '0.003'

  > 0.003.toExponential()
  '3e-3'
  > 0.003.toExponential(4)
  '3.0000e-3'
  ```

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

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

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

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

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

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

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

Works like `.toString()`, but prunes the mantissa to `precision` digits before returning a result. If `precision` is missing, `.toString()` is used.

```
> 1234..toPrecision(3)  // requires exponential notation
'1.23e+3'

> 1234..toPrecision(4)
'1234'

> 1234..toPrecision(5)
'1234.0'

> 1.234.toPrecision(3)
'1.23'
```

- `toString(radix=10): string` [ES1]

  Returns a string representation of the number.

  Returning a result with base 10:

  ```
  > 123.456.toString()
  '123.456'
  ```

  Returning results with bases other than 10 (specified via `radix`):

  ```
  > 4..toString(2)
  '100'
  > 4.5.toString(2)
  '100.1'

  > 255..toString(16)
  'ff'
  > 255.66796875.toString(16)
  'ff.ab'

  > 1234567890..toString(36)
  'kf12oi'
  ```

  You can use `parseInt()` to convert integer results back to numbers:

  ```
  > parseInt('kf12oi', 36)
  1234567890
  ```

15.13.7 Sources

- Wikipedia
- TypeScript's built-in typings
- 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. Today, it would probably be created as a module, but it has existed since long before 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. This function returns such values whenever the result of `.exp()` is close to 1.

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

  Returns the natural logarithm of `x` (to base e, Euler's number). The inverse of `Math.exp()`.

  ```
  > Math.log(1)
  0
  > Math.log(Math.E)
  1
  > Math.log(Math.E ** 2)
  2
  ```

- `Math.log1p(x: number): number` [ES6]

  Returns `Math.log(1 + x)`. The inverse of `Math.expm1()`. Very small numbers (fractions close to 0) are represented with a higher precision. This function receives such numbers whenever a parameter to `.log()` is close to 1.

- `Math.log10(x: number): number` [ES6]

  Returns the logarithm of `x` to base 10. The inverse of `10 ** x`.

  ```
  > Math.log10(1)
  0
  > Math.log10(10)
  1
  > Math.log10(100)
  2
  ```

- `Math.log2(x: number): number` [ES6]

  Returns the logarithm of `x` to base 2. The inverse of `2 ** x`.

  ```
  > Math.log2(1)
  0
  > Math.log2(2)
  1
  > Math.log2(4)
  2
  ```

- `Math.pow(x: number, y: number): number` [ES1]

  Returns x^y, `x` to the power of `y`. The same as `x ** y`.

  ```
  > Math.pow(2, 3)
  8
  > Math.pow(25, 0.5)
  5
  ```

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

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

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

16.3 Rounding

Rounding means converting an arbitrary number to an integer (a number without a decimal fraction). Tbl. 16.1 lists the available functions and what they return for a few representative inputs.

Table 16.1: Rounding functions of `Math`.

	-2.9	-2.5	-2.1	2.1	2.5	2.9
`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

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

  Returns the smallest (closest to $-\infty$) integer `i` with $x \leq i$.

  ```
  > Math.ceil(1.9)
  2
  > Math.ceil(2.1)
  3
  ```

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

  Returns the greatest (closest to $+\infty$) integer `i` with $i \leq x$.

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

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

  Returns the integer that is closest to `x`. If the decimal fraction of `x` is `.5` then `.round()` rounds up (to the integer closer to positive infinity):

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

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

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

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

16.4 Trigonometric Functions

All angles are specified in radians. Use the following two functions to convert between degrees and radians.

```
function toRadians(degrees) {
  return degrees / 180 * Math.PI;
}
function toDegrees(radians) {
  return radians / Math.PI * 180;
}
```

- `Math.acos(x: number): number` [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 asm.js helpers

WebAssembly is a virtual machine based on JavaScript that is supported by most JavaScript engines.

asm.js is a precursor to WebAssembly. It is a subset of JavaScript that produces fast executables if static languages (such as C++) are compiled to it. In a way, it is also a virtual machine, within the confines of JavaScript.

The following two methods help asm.js and have few use cases, otherwise.

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

  Rounds `x` to a 32-bit floating point value (`float`). Used by asm.js to tell an engine to internally use a `float` value (normal numbers are doubles and take up 64 bits).

- `Math.imul(x: number, y: number): number` [ES6]

  Multiplies the two 32 bit integers `x` and `y` and returns the lower 32 bits of the result. Needed by asm.js: All other basic 32-bit math operations can be simulated by coercing 64-bit results to 32 bits. With multiplication, you may lose bits for results beyond 32 bits.

16.6 Various other functions

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

  Returns the absolute value of `x`.

```
> Math.abs(3)
3
> Math.abs(-3)
3
> Math.abs(0)
0
```

- `Math.clz32(x: number): number` [ES6]

  Counts the leading zero bits in the 32-bit integer `x`. Used in DSP algorithms.

```
> Math.clz32(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
```

16.7 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 1 was published in 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 pieces of data with fixed sizes. One or more code units encode a single code point. The size of code units depends on the encoding format. The most popular format, UTF-8, has 8-bit code units.

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
  - This is the most frequently used plane. Roughly, it comprises the original Unicode.
- Plane 1: Supplementary Multilingual Plane (SMP), 0x10000–0x1FFFF
- Plane 2: Supplementary Ideographic Plane (SIP), 0x20000–0x2FFFF
- Plane 3–13: Unassigned
- Plane 14: Supplementary Special-Purpose Plane (SSP), 0xE0000–0xEFFFF
- Plane 15–16: Supplementary Private Use Area (S PUA A/B), 0x0F0000–0x10FFFF

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

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

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

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

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

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

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:

- BMP (first 16 bits of Unicode): are stored in single code units.

- Astral planes: After subtracting the BMP's count of 0x10000 characters from Unicode's count of 0x110000 characters, 0x100000 characters (20 bits) remain. These are stored in unoccupied "holes" in the BMP:
  - Most significant 10 bits (*leading surrogate*): 0xD800-0xDBFF
  - Least significant 10 bits (*trailing surrogate*): 0xDC00-0xDFFF

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

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

Notes:

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

17.2 Web development: UTF-16 and UTF-8

For web development, two Unicode encoding formats are relevant: 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 UTF-16 code units:

```
> const smiley = '😊';
> smiley.length
2
```

```
> smiley === '\uD83D\uDE42' // code units
true
> smiley === '\u{1F642}' // code point
true
```

For more information on Unicode and strings, consult the section on atoms of text in the chapter on strings.

17.2.3 Source code in files: UTF-8

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

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

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, code points can be said to represent Unicode "characters".

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

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

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

Another example is flag emojis:

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

Reading: More information on grapheme clusters

For more information, consult "Let's Stop Ascribing Meaning to Code Points" 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 with its double quotes.

The next chapter covers *template literals*, which give you:

- String interpolation
- Multiple lines
- Raw string literals (backslash has no special meaning)

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:

```
'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 transported as strings.

```
const str = 'abc';

// Reading a character at a given index
assert.equal(str[1], 'b');

// Counting the characters in a string:
assert.equal(str.length, 3);
```

18.2.2 Accessing Unicode code points via `for-of` and spreading

Iterating over strings via `for-of` or spreading (`...`) visits Unicode code points. Each code point is represented by 1–2 JavaScript characters. For more information, see the section on the atoms of text in this chapter.

This is how you iterate over the code points of a string via `for-of`:

```
for (const ch of 'abc') {
  console.log(ch);
}
// Output:
// 'a'
// 'b'
// 'c'
```

And this is how you convert a string into an Array of code points via spreading:

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

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');
```

As an aside, this way of assembling strings is quite efficient, because most JavaScript engines internally optimize it.

> **Exercise: Concatenating strings**
>
> `exercises/strings/concat_string_array_test.js`

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 can't convert it back via `Boolean()`.

```
> String(false)
'false'
> Boolean('false')
true
```

18.4.1 Stringifying objects

Plain objects have a default representation that is not very useful:

```
> String({a: 1})
'[object Object]'
```

Arrays have a better string representation, but it still hides much information:

```
> String(['a', 'b'])
'a,b'
> String(['a', ['b']])
'a,b'

> String([1, 2])
'1,2'
> String(['1', '2'])
'1,2'

> String([true])
'true'
> String(['true'])
'true'
> String(true)
'true'
```

Stringifying functions returns their source code:

```
> String(function f() {return 4})
'function f() {return 4}'
```

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 stringify data:

```
> JSON.stringify({a: 1})
'{"a":1}'
> JSON.stringify(['a', ['b']])
'["a",["b"]]'
```

The caveat is that JSON only supports `null`, booleans, numbers, strings, Arrays and objects (which it always treats as if they were created by object literals).

Tip: The third parameter lets you switch on multi-line output and specify how much to indent. For example:

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

This statement produces the following output.

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

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: JavaScript characters, code points, grapheme clusters

Quick recap of the chapter on Unicode:

- Code points: Unicode characters, with a range of 21 bits.
- UTF-16 code units: JavaScript characters, with a range of 16 bits. Code points are encoded as 1–2 UTF-16 code units.
- Grapheme clusters: *Graphemes* are written symbols, as displayed on screen or paper. A *grapheme cluster* is a sequence of 1 or more code points that encodes a grapheme.

To represent code points in JavaScript strings, one or two JavaScript characters are used. You can see that when counting characters via `.length`:

```
// 3 Unicode code points, 3 JavaScript characters:
assert.equal('abc'.length, 3);

// 1 Unicode code point, 2 JavaScript characters:
assert.equal('😀'.length, 2);
```

18.6.1 Working with code points

Let's explore JavaScript's tools for working with code points.

Code point escapes let you specify code points hexadecimally. They expand to one or two JavaScript characters.

```
> '\u{1F642}'
'😂'
```

Converting from code points:

```
> String.fromCodePoint(0x1F642)
'😂'
```

Converting to code points:

```
> '😀'.codePointAt(0).toString(16)
'1f642'
```

Iteration honors code points. For example, the iteration-based `for-of` loop:

```
const str = '☺a';
assert.equal(str.length, 3);

for (const codePoint of str) {
  console.log(codePoint);
}

// Output:
// '☺'
// 'a'
```

Or iteration-based *spreading* (`...`):

```
> [...'☺a']
[ '☺', 'a' ]
```

Spreading is therefore a good tool for counting code points:

```
> [...'☺a'].length
2
> '☺a'.length
3
```

18.6.2 Working with code units

Indices and lengths of strings are based on JavaScript characters (i.e., code units).

To specify code units numerically, you can use *code unit escapes*:

```
> '\uD83D\uDE42'
'☺'
```

And you can use so-called *char codes*:

```
> String.fromCharCode(0xD83D) + String.fromCharCode(0xDE42)
'☺'
```

To get the char code of a character, use `.charCodeAt()`:

```
> '☺'.charCodeAt(0).toString(16)
'd83d'
```

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 code units.

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. 18.1 describes how various values are converted to strings.

Table 18.1: Converting values to strings.

`x`	`String(x)`
`undefined`	`'undefined'`
`null`	`'null'`
Boolean value	`false` → `'false'`, `true` → `'true'`
Number value	Example: `123` → `'123'`
String value	`x` (input, unchanged)
An object	Configurable via, e.g., `toString()`

18.7.2 Numeric values of characters and code points

- **Char codes:** Unicode UTF-16 code units as numbers
  - `String.fromCharCode()`, `String.prototype.charCodeAt()`
  - Precision: 16 bits, unsigned
- **Code points:** Unicode code points as numbers
  - `String.fromCodePoint()`, `String.prototype.codePointAt()`
  - Precision: 21 bits, unsigned (17 planes, 16 bits each)

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

- `.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 be `minIndex` or higher.

  ```
  > 'abab'.indexOf('a')
  0
  > 'abab'.indexOf('a', 1)
  2
  > 'abab'.indexOf('c')
  -1
  ```

- `.lastIndexOf(searchString: string, maxIndex=Infinity): number` [ES1]

  Returns the highest index at which `searchString` appears within the string, or `-1`, otherwise. Any returned index will be `maxIndex` or lower.

  ```
  > 'abab'.lastIndexOf('ab', 2)
  2
  > 'abab'.lastIndexOf('ab', 1)
  0
  > 'abab'.lastIndexOf('ab')
  2
  ```

- `.match(regExp: string | RegExp): RegExpMatchArray | null` [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 before performing the previous steps.

  The result has the following type:

```
interface RegExpMatchArray extends Array<string> {
  index: number;
  input: string;
  groups: undefined | {
    [key: string]: string
  };
}
```

Numbered capture groups become Array indices. Named capture groups (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
```

- `.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.

  ```
  > '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`. You can use negative indices where `-1` means `this.length-1` (etc.).

  ```
  > 'abc'.slice(1, 3)
  'bc'
  > 'abc'.slice(1)
  'bc'
  > 'abc'.slice(-2)
  'bc'
  ```

- `.split(separator: string | RegExp, limit?: number): string[]` [ES3]

  Splits the string into an Array of substrings – the strings that occur between the separators. The separator can either be a string or a regular expression. Captures made by groups in the regular expression are included in the result.

  ```
  > 'abc'.split('')
  [ 'a', 'b', 'c' ]
  > 'a | b | c'.split('|')
  [ 'a ', ' b ', ' c' ]

  > 'a : b : c'.split(/ *: */)
  [ 'a', 'b', 'c' ]
  > 'a : b : c'.split(/( *):( *)/)
  [ 'a', ' ', ' ', 'b', ' ', ' ', 'c' ]
  ```

- `.substring(start: number, end=this.length): string` [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'+'b'` is equivalent to `'a'.concat('b')` and more concise.

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

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

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

  ```
  > '#'.padEnd(2)
  '# '
  > 'abc'.padEnd(2)
  ```

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

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

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

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

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

  Returns a string that is the string, repeated `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.

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

  Replace matches of `searchValue` with `replaceValue`. If `searchValue` is a string, only the first verbatim occurrence is replaced. If `searchValue` is a regular expression without flag `/g`, only the first match is replaced. If `searchValue` is a regular expression with `/g` then all matches are replaced.

```
> 'x.x.'.replace('.', '#')
'x#x.'
> 'x.x.'.replace(/./, '#')
'#.x.'
> 'x.x.'.replace(/./g, '#')
'####'
```

  Special characters in `replaceValue` are:

  - `$$`: becomes `$`
  - `$n`: becomes the capture of numbered group `n` (alas, `$0` does not work)
  - `$&`: becomes the complete match
  - `` $` ``: becomes everything before the match
  - `$'`: becomes everything after the match

  Examples:

```
> 'a 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:

```
> 'a 2020-04 b'.replace(/(?<year>[0-9]{4})-(?<month>[0-9]{2})/, '|$<month>|')
'a |04| b'
```

- `.replace(searchValue: string | RegExp, replacer: (...args: any[]) => string): string` [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, a last parameter contains an object whose properties contain the captures:

  ```
  const regexp = /(?<year>[0-9]{4})-(?<month>[0-9]{2})/;
  const replacer = (...args) => {
    const groups=args.pop();
    return '|' + groups.month + '|';
  };
  assert.equal(
    'a 2020-04 b'.replace(regexp, replacer),
    'a |04| b');
  ```

- `.toUpperCase(): string` [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()
'#'
```

- `.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 `String.prototype`: chars, char codes, code points

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

  Returns the character at index `pos`, as a string (JavaScript does not have a datatype for characters). `str[i]` is equivalent to `str.charAt(i)` and more concise (caveat: may not work on old engines).

```
> 'abc'.charAt(1)
'b'
```

- `.charCodeAt(pos: number): number` [ES1]

  Returns the 16-bit number (0–65535) of the UTF-16 code unit (character) at index `pos`.

```
> 'abc'.charCodeAt(1)
98
```

- `.codePointAt(pos: number): number | undefined` [ES6]

  Returns the 21-bit number of the Unicode code point of the 1–2 characters at index `pos`. If there is no such index, it returns `undefined`.

18.7.9 Sources

- TypeScript's built-in typings
- MDN web docs for JavaScript
- ECMAScript language specification

Exercise: Using string methods

`exercises/strings/remove_extension_test.js`

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 *web template* is a function from data to text. It is frequently used in web development and often defined via text files. For example, the following text defines a template for the library Handlebars:

```
<div class="entry">
  <h1>{{title}}</h1>
  <div class="body">
    {{body}}
  </div>
</div>
```

- A *template literal* is a string literal with more features. For example, interpolation. It is delimited by backticks:

```
const num = 5;
assert.equal(`Count: ${num}!`, 'Count: 5!');
```

- A *tagged template* is a function followed by a template literal. It results in that function being called and the contents of the template literal being fed into it as parameters.

```
const getArgs = (...args) => args;
assert.deepEqual(
  getArgs`Count: ${5}!`,
  [['Count: ', '!'], 5] );
```

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

19.2 Template literals

Template literals have two main benefits, compared to normal string literals.

First, they support *string interpolation*: you can insert expressions if you put them inside `${` and `}`:

```
const MAX = 100;
function doSomeWork(x) {
  if (x > MAX) {
    throw new Error(`At most ${MAX} allowed: ${x}!`);
  }
  // ···
}
assert.throws(
  () => doSomeWork(101),
  {message: 'At most 100 allowed: 101!'});
```

Second, template literals can span multiple lines:

```
const str = `this is
a text with
```

```
multiple lines`;
```

Template literals always produce strings.

19.3 Tagged templates

The expression in line A is a *tagged template*:

```
const first = 'Lisa';
const last = 'Simpson';

const result = tagFunction`Hello ${first} ${last}!`; // A
```

The last line is equivalent to:

```
const result = tagFunction(['Hello ', ' ', '!'], first, last);
```

The parameters of `tagFunction` are:

- Template strings (first parameter): an Array with the text fragments surrounding the interpolations (`${···}`).
  - In the example: `['Hello ', ' ', '!']`
- Substitutions (remaining parameters): the interpolated values.
  - In the example: `'Lisa'` and `'Simpson'`

The static (fixed) parts of the literal (the template strings) are separated from the dynamic parts (the substitutions).

`tagFunction` can return arbitrary values and gets two views of the template strings as input (only the cooked view is shown in the previous example):

- A *cooked view* where, e.g.:
  - `\t` becomes a tab
  - `\\` becomes a single backslash
- A *raw view* where, e.g.:
  - `\t` becomes a slash followed by a `t`
  - `\\` becomes two backslashes

The raw view enables raw string literals via `String.raw` (described later) and similar applications.

Tagged templates are great for supporting small embedded languages (so-called *domain-specific languages*). We'll continue with a few examples.

19.3.1 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.

19.3.2 Tag function library: re-template-tag

re-template-tag is a simple library for composing regular expressions. Templates tagged with `re` produce regular expressions. The main benefit is that you can interpolate regular expressions and plain text via `${}` (see `RE_DATE`):

```
import {re} from 're-template-tag';

const RE_YEAR = re`(?<year>[0-9]{4})`;
const RE_MONTH = re`(?<month>[0-9]{2})`;
const RE_DAY = re`(?<day>[0-9]{2})`;
const RE_DATE = re`/${RE_YEAR}-${RE_MONTH}-${RE_DAY}/u`;

const match = RE_DATE.exec('2017-01-27');
assert.equal(match.groups.year, '2017');
```

19.3.3 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 a string literal where backslashes don't do anything special (such as escaping characters etc.):

```
assert.equal(String.raw`\back`, '\\back');
```

One example where that helps is strings with regular expressions:

```
const regex1 = /^\./;
const regex2 = new RegExp('^\\.');
const regex3 = new RegExp(String.raw`^\.`);
```

All three regular expressions are equivalent. You can see that with a string literal, you have to write the backslash twice to escape it for that literal. With a raw string literal, you don't have to do that.

Another example where raw string literal are useful is Windows paths:

```
const WIN_PATH = String.raw`C:\foo\bar`;
assert.equal(WIN_PATH, 'C:\\foo\\bar');
```

19.5 (Advanced)

All remaining sections are advanced

19.6 Multi-line template literals and indentation

If you put multi-line text in template literals, two goals are in conflict: On one hand, the text should be indented to fit inside the source code. On the other hand, its lines should start in the leftmost column.

For example:

```
function div(text) {
  return `
    <div>
      ${text}
    </div>
  `;
}
console.log('Output:');
console.log(div('Hello!')
  // Replace spaces with mid-dots:
  .replace(/ /g, '·')
  // Replace \n with #\n:
  .replace(/\n/g, '#\n'));
```

Due to the indentation, the template literal fits well into the source code. Alas, the output is also indented. And we don't want the return at the beginning and the return plus two spaces at the end.

```
Output:
#
····<div>#
··········Hello!#
····</div>#
··
```

There are two ways to fix this: via a tagged template or by trimming the result of the template literal.

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 cuts off the indents everywhere. It also removes the line break at the very beginning and the indentation at the very end. One such template tag is `dedent` by Desmond Brand:

```
import dedent from 'dedent';
function divDedented(text) {
  return dedent`
    <div>
      ${text}
    </div>
  `;
}
console.log('Output:');
console.log(divDedented('Hello!'));
```

This time, the output is not indented:

```
Output:
<div>
  Hello!
</div>
```

19.6.2 Fix: `.trim()`

The second fix is quicker, but also dirtier:

```
function divDedented(text) {
  return `
<div>
  ${text}
</div>
  `.trim();
```

```
}
console.log('Output:');
console.log(divDedented('Hello!'));
```

The string method `.trim()` removes the superfluous whitespace at the beginning and at the end, but the content itself must start in the leftmost column. The advantage of this solution is not needing a custom tag function. The downside is that it looks ugly.

The output looks like it did with `dedent` (however, there is no line break at the end):

```
Output:
<div>
  Hello!
</div>
```

19.7 Simple templating via template literals

While template literals look like web templates, it is not immediately obvious how to use them for (web) templating: A web template gets its data from an object, while a template literal gets its data from variables. The solution is to use a template literal in the body of a function whose parameter receives the templating data. For example:

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

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();
```

`tmpl()` takes the following steps:

- The text inside the `<table>` is produced via a nested templating function for single addresses (line 4). Note how it uses the string method `.trim()` at the end, to remove unnecessary whitespace.
- The nested templating function is applied to each element of the Array `addrs` via the Array method `.map()` (line 3).
- The resulting Array (of strings) is converted into a string via the Array method `.join()` (line 10).
- The helper function `escapeHtml()` is used to escape special HTML characters (line 6 and line 7). Its implementation is shown in the next section.

This is how to call `tmpl()` with the addresses and log the result:

```
console.log(tmpl(addresses));
```

The output is:

```
<table>
  <tr>
        <td>&lt;Jane&gt;</td>
        <td>Bond</td>
      </tr><tr>
        <td>Lars</td>
        <td>&lt;Croft&gt;</td>
      </tr>
</table>
```

19.7.2 Simple HTML-escaping

```
function escapeHtml(str) {
  return str
    .replace(/&/g, '&amp;') // first!
    .replace(/>/g, '&gt;')
    .replace(/</g, '&lt;')
    .replace(/"/g, '&quot;')
    .replace(/'/g, '&#39;')
    .replace(/`/g, '&#96;')
    ;
}
```

Exercise: HTML templating

Exercise with bonus challenge: `exercises/template-literals/templating_test.js`

19.8 Further reading

- How to implement your own tag functions is described in "Exploring ES6".

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` and they can be property keys in objects:

```
const sym = Symbol();
assert.equal(typeof sym, 'symbol');

const obj = {
  [sym]: 123,
};
```

20.1 Use cases for symbols

The main use cases for symbols are:

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

20.1.1 Symbols: enum-style values

Let's assume you want to create constants representing the colors red, orange, yellow, green, blue and violet. One simple way of doing so would be to use strings:

```
const COLOR_BLUE = 'Blue';
```

On the plus side, logging that constant produces helpful output. On the minus side, there is a risk of mistaking an unrelated value for a color, because two strings with the same content are considered equal:

```
const MOOD_BLUE = 'Blue';
assert.equal(COLOR_BLUE, MOOD_BLUE);
```

We can fix that problem via a symbol:

```
const COLOR_BLUE = Symbol('Blue');

const MOOD_BLUE = 'Blue';
assert.notEqual(COLOR_BLUE, MOOD_BLUE);
```

Let's use symbol-valued constants to implement a function:

```
const COLOR_RED    = Symbol('Red');
const COLOR_ORANGE = Symbol('Orange');
const COLOR_YELLOW = Symbol('Yellow');
const COLOR_GREEN  = Symbol('Green');
const COLOR_BLUE   = Symbol('Blue');
const COLOR_VIOLET = Symbol('Violet');

function getComplement(color) {
  switch (color) {
    case COLOR_RED:
      return COLOR_GREEN;
    case COLOR_ORANGE:
      return COLOR_BLUE;
    case COLOR_YELLOW:
      return COLOR_VIOLET;
    case COLOR_GREEN:
      return COLOR_RED;
    case COLOR_BLUE:
      return COLOR_ORANGE;
    case COLOR_VIOLET:
      return COLOR_YELLOW;
```

```
    default:
      throw new Exception('Unknown color: '+color);
  }
}
assert.equal(getComplement(COLOR_YELLOW), COLOR_VIOLET);
```

Notably, this function throws an exception if you call it with `'Blue'`:

```
assert.throws(() => getComplement('Blue'));
```

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. Its keys reflect the problem domain.
- Libraries and ECMAScript operate at a meta-level. For example, `.toString` is a meta-level property key:

The following code demonstrates the difference:

```
const point = {
  x: 7,
  y: 4,
  toString() {
    return `(${this.x}, ${this.y})`;
  },
};
assert.equal(String(point), '(7, 4)');
```

Properties `.x` and `.y` exist at the base level. They are the coordinates of the point encoded by `point` and reflect the problem domain. Method `.toString()` is a meta-level property. It tells JavaScript how to stringify this object.

The meta-level and the base level must never clash, which becomes harder when introducing new mechanisms later in the life of a programming language.

Symbols can be used as property keys and solve this problem: Each symbol is unique and never clashes with any string or any other symbol.

As an example, let's assume we are writing a library that treats objects differently if they implement a special method. This is what defining a property key for such a method and implementing it for an object would look like:

```
const specialMethod = Symbol('specialMethod');
const obj = {
  [specialMethod](x) {
    return x + x;
  }
};
assert.equal(obj[specialMethod]('abc'), 'abcabc');
```

This syntax is explained in more detail in the chapter on objects.

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. Iteration is explained in its own chapter.
- `Symbol.hasInstance`: customizes how `instanceof` works. It's the key of a method to be implemented by the right-hand side of that operator. For example:

```
class PrimitiveNull {
  static [Symbol.hasInstance](x) {
    return x === null;
  }
}
assert.equal(null instanceof PrimitiveNull, true);
```

- `Symbol.toStringTag`: influences the default `.toString()` method.

```
> String({})
'[object Object]'
> String({ [Symbol.toStringTag]: 'is no money' })
'[object is no money]'
```

  Note: It's usually better to override `.toString()`.

Exercises: Publicly known symbols

- `Symbol.toStringTag`: `exercises/symbols/to_string_tag_test.js`
- `Symbol.hasInstance`: `exercises/symbols/has_instance_test.js`

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 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)'
```

20.4 Further reading

- For in-depth information on symbols (cross-realm symbols, all publicly known symbols, etc.), see "Exploring ES6"

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` statements (ES1)
- `switch` statements (ES3)
- `while` loops (ES1)
- `do-while` loops (ES3)

- `for` loops (ES1)
- `for-of` loops (ES6)
- `for-await-of` loops (ES2018)
- `for-in` loops (ES1)

Before we get to the actual control flow statements, let's take a look at two operators for controlling loops.

21.1 Controlling loops: `break` and `continue`

The two operators `break` and `continue` can be used to control loops and other statements while you are inside them.

21.1.1 `break`

There are two versions of `break`: one with an operand and one without an operand. The latter version works inside the following statements: `while`, `do-while`, `for`, `for-of`, `for-await-of`, `for-in` and `switch`. It immediately leaves the current statement:

```
for (const x of ['a', 'b', 'c']) {
  console.log(x);
  if (x === 'b') break;
  console.log('---')
}

// Output:
// 'a'
// '---'
// 'b'
```

21.1.2 Additional use case for `break`: leaving blocks

`break` with an operand works everywhere. Its operand is a *label*. Labels can be put in front of any statement, including blocks. `break foo` leaves the statement whose label is `foo`:

```
foo: { // label
  if (condition) break foo; // labeled break
  // ···
}
```

Breaking from blocks is occasionally handy if you are using a loop and want to distinguish between finding what you were looking for and finishing the loop without success:

```
function search(stringArray, suffix) {
  let result;
  search_block: {
```

```
    for (const str of stringArray) {
      if (str.endsWith(suffix)) {
        // Success
        result = str;
        break search_block;
      }
    } // for
    // Failure
    result = '(Untitled)';
  } // search_block

  return { suffix, result };
    // same as: {suffix: suffix, result: result}
}
assert.deepEqual(
  search(['foo.txt', 'bar.html'], '.html'),
  { suffix: '.html', result: 'bar.html' }
);
assert.deepEqual(
  search(['foo.txt', 'bar.html'], '.js'),
  { suffix: '.js', result: '(Untitled)' }
);
```

21.1.3 `continue`

`continue` only works inside `while`, `do-while`, `for`, `for-of`, `for-await-of` and `for-in`. It immediately leaves the current loop iteration and continues with the next one. For example:

```
const lines = [
  'Normal line',
  '# Comment',
  'Another normal line',
];
for (const line of lines) {
  if (line.startsWith('#')) continue;
  console.log(line);
}
// Output:
// 'Normal line'
// 'Another normal line'
```

21.2 `if` statements

These are two simple `if` statements: One with just a "then" branch and one with both a "then" branch and an "else" branch:

```
if (cond) {
  // then branch
}

if (cond) {
  // then branch
} else {
  // else branch
}
```

Instead of the block, `else` can also be followed by another `if` statement:

```
if (cond1) {
  // ···
} else if (cond2) {
  // ···
}

if (cond1) {
  // ···
} else if (cond2) {
  // ···
} else {
  // ···
}
```

You can continue this chain with more `else if`s.

21.2.1 The syntax of `if` statements

The general syntax of `if` statements is:

```
if (cond) «then_statement»
else «else_statement»
```

So far, the `then_statement` has always been a block, but you can also use a statement. That statement must be terminated with a semicolon:

```
if (true) console.log('Yes'); else console.log('No');
```

That means that `else if` is not its own construct, it's simply an `if` statement whose `else_statement` is another `if` statement.

21.3 `switch` statements

The head of a `switch` statement looks as follows:

```
switch («switch_expression») {
  «switch_body»
}
```

Inside the body of `switch`, there are zero or more case clauses:

```
case «case_expression»:
  «statements»
```

And, optionally, a default clause:

```
default:
  «statements»
```

A `switch` is executed as follows:

- Evaluate the switch expression.
- Jump to the first case clause whose expression has the same result as the switch expression.
- If there is no such case clause, jump to the default clause.
- If there is no default clause, nothing happens.

21.3.1 A first example

Let's look at an example: The following function converts a number from 1–7 to the name of a weekday.

```
function dayOfTheWeek(num) {
  switch (num) {
    case 1:
      return 'Monday';
    case 2:
      return 'Tuesday';
    case 3:
      return 'Wednesday';
    case 4:
      return 'Thursday';
    case 5:
      return 'Friday';
    case 6:
      return 'Saturday';
    case 7:
      return 'Sunday';
  }
}
assert.equal(dayOfTheWeek(5), 'Friday');
```

21.3.2 Don't forget to `return` or `break`!

At the end of a case clause, execution continues with the next case clause (unless you `return` or `break`). For example:

```
function dayOfTheWeek(num) {
  let name;
```

```
  switch (num) {
    case 1:
      name = 'Monday';
    case 2:
      name = 'Tuesday';
    case 3:
      name = 'Wednesday';
    case 4:
      name = 'Thursday';
    case 5:
      name = 'Friday';
    case 6:
      name = 'Saturday';
    case 7:
      name = 'Sunday';
  }
  return name;
}
assert.equal(dayOfTheWeek(5), 'Sunday'); // not 'Friday'!
```

That is, the previous implementation of `dayOfTheWeek()` only worked, because we used `return`. We can fix this implementation by using `break`:

```
function dayOfTheWeek(num) {
  let name;
  switch (num) {
    case 1:
      name = 'Monday';
      break;
    case 2:
      name = 'Tuesday';
      break;
    case 3:
      name = 'Wednesday';
      break;
    case 4:
      name = 'Thursday';
      break;
    case 5:
      name = 'Friday';
      break;
    case 6:
      name = 'Saturday';
      break;
    case 7:
      name = 'Sunday';
      break;
  }
  return name;
```

```
}
assert.equal(dayOfTheWeek(5), 'Friday');
```

21.3.3 Empty cases clauses

The statements of a case clause can be omitted, which effectively gives us multiple case expressions per case clause:

```
function isWeekDay(name) {
  switch (name) {
    case 'Monday':
    case 'Tuesday':
    case 'Wednesday':
    case 'Thursday':
    case 'Friday':
      return true;
    case 'Saturday':
    case 'Sunday':
      return false;
  }
}
assert.equal(isWeekDay('Wednesday'), true);
assert.equal(isWeekDay('Sunday'), false);
```

21.3.4 Checking for illegal values via a `default` clause

A `default` clause is jumped to if the `switch` expression has no other match. That makes it useful for error checking:

```
function isWeekDay(name) {
  switch (name) {
    case 'Monday':
    case 'Tuesday':
    case 'Wednesday':
    case 'Thursday':
    case 'Friday':
      return true;
    case 'Saturday':
    case 'Sunday':
      return false;
    default:
      throw new Error('Illegal value: '+name);
  }
}
assert.throws(
  () => isWeekDay('January'),
  {message: 'Illegal value: January'});
```

Exercises: `switch`

- `exercises/control-flow/number_to_month_test.js`
- Bonus: `exercises/control-flow/is_object_via_switch_test.js`

21.4 `while` loops

A `while` loop has the following syntax:

```
while («condition») {
  «statements»
}
```

Before each loop iteration, `while` evaluates `condition`:

- If the result is falsy, the loop is finished.
- If the result is truthy, the `while` body is executed one more time.

21.4.1 Examples

The following code uses a `while` loop. In each loop iteration, it removes the first element of `arr` via `.shift()` and logs it.

```
const arr = ['a', 'b', 'c'];
while (arr.length > 0) {
  const elem = arr.shift(); // remove first element
  console.log(elem);
}
// Output:
// 'a'
// 'b'
// 'c'
```

If the condition is `true` then `while` is an infinite loop:

```
while (true) {
  if (Math.random() === 0) break;
}
```

21.5 `do-while` loops

The `do-while` loop works much like `while`, but it checks its condition *after* each loop iteration (not before).

```
let input;
do {
```

```
  input = prompt('Enter text:');
} while (input !== ':q');
```

21.6 `for` loops

With a `for` loop, you use the head to control how its body is executed. The head has three parts and each of them is optional:

```
for («initialization»; «condition»; «post_iteration») {
  «statements»
}
```

- `initialization`: sets up variables etc. for the loop. Variables declared here via `let` or `const` only exist inside the loop.
- `condition`: This condition is checked before each loop iteration. If it is falsy, the loop stops.
- `post_iteration`: This code is executed after each loop iteration.

A `for` loop is therefore roughly equivalent to the following `while` loop:

```
«initialization»
while («condition») {
  «statements»
  «post_iteration»
}
```

21.6.1 Examples

As an example, this is how to count from zero to two via a `for` loop:

```
for (let i=0; i<3; i++) {
  console.log(i);
}

// Output:
// 0
// 1
// 2
```

This is how to log the contents of an Array via a `for` loop:

```
const arr = ['a', 'b', 'c'];
for (let i=0; i<3; i++) {
  console.log(arr[i]);
}

// Output:
// 'a'
```

```
// 'b'
// 'c'
```

If you omit all three parts of the head, you get an infinite loop:

```
for (;;) {
  if (Math.random() === 0) break;
}
```

21.7 `for-of` loops

A `for-of` loop iterates over an *iterable* – a data container that supports the iteration protocol. Each iterated value is stored in a variable, as specified in the head:

```
for («iteration_variable» of «iterable») {
  «statements»
}
```

The iteration variable is usually created via a variable declaration:

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

But you can also use a (mutable) variable that already exists:

```
const iterable = ['hello', 'world'];
let elem;
for (elem of iterable) {
  console.log(elem);
}
```

21.7.1 `const`: `for-of` vs. `for`

Note that, in `for-of` loops, you can use `const`. The iteration variable can still be different for each iteration (it just can't change during the iteration). Think of it as a new `const` declaration being executed each time, in a fresh scope.

In contrast, in `for` loops, you must declare variables via `let` or `var` if their values change.

21.7.2 Iterating over iterables

As mentioned before, `for-of` works with any iterable object, not just with Arrays. For example, with Sets:

```
const set = new Set(['hello', 'world']);
for (const elem of set) {
  console.log(elem);
}
```

21.7.3 Iterating over [index, element] pairs of Arrays

Lastly, you can also use `for-of` to iterate over the [index, element] entries of Arrays:

```
const arr = ['a', 'b', 'c'];
for (const [index, elem] of arr.entries()) {
  console.log(`${index} -> ${elem}`);
}
// Output:
// '0 -> a'
// '1 -> b'
// '2 -> c'
```

> **Exercise: `for-of`**
>
> `exercises/control-flow/array_to_string_test.js`

21.8 `for-await-of` loops

`for-await-of` is like `for-of`, but it works with asynchronous iterables instead of synchronous ones. And it can only be used inside async functions and async generators.

```
for await (const item of asyncIterable) {
  // ···
}
```

`for-await-of` is described in detail in a later chapter.

21.9 `for-in` loops (avoid)

`for-in` has several pitfalls. Therefore, it is usually best to avoid it.

This is an example of using `for-in`:

```
function getOwnPropertyNames(obj) {
  const result = [];
  for (const key in obj) {
    if ({}.hasOwnProperty.call(obj, key)) {
      result.push(key);
    }
  }
```

```
  return result;
}
assert.deepEqual(
  getOwnPropertyNames({ a: 1, b:2 }),
  ['a', 'b']);
assert.deepEqual(
  getOwnPropertyNames(['a', 'b']),
  ['0', '1']); // strings!
```

This is a better alternative:

```
function getOwnPropertyNames(obj) {
  const result = [];
  for (const key of Object.keys(obj)) {
    result.push(key);
  }
  return result;
}
```

For more information on `for-in`, consult "Speaking JavaScript".

Quiz

See quiz app.

Chapter 22

Exception handling

Contents

This chapter covers how JavaScript handles exceptions.

As an aside: JavaScript didn't support exceptions until ES3. That explains why they are used sparingly by the language and its standard library.

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
```

`throw` walks up this chain of constructs, until it finds a `try` statement. Execution continues in the `catch` clause of that `try` statement.

22.2 `throw`

```
throw «value»;
```

Any value can be thrown, but it's best to throw instances of `Error`:

```
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;
}
assert.throws(
  () => func(),
  MyError);
```

22.3 try-catch-finally

The maximal version of the `try` statement looks as follows:

```
try {
  // try_statements
} catch (error) {
  // catch_statements
} finally {
  // finally_statements
}
```

The `try` clause is mandatory, but you can omit either `catch` or `finally` (but not both). Since ECMAScript 2019, you can also omit `(error)`, if you are not interested in the value that was thrown.

22.3.1 The catch clause

If an exception is thrown in the `try` block (and not caught earlier) then it is assigned to the parameter of the `catch` clause and the code in that clause is executed. Unless it is directed elsewhere (via `return` or similar), execution continues after the `catch` clause: with the `finally` clause – if it exists – or after the `try` statement.

The following code demonstrates that the value that is thrown in line A is indeed caught in line B.

```
const errorObject = new Error();
function func() {
  throw errorObject; // (A)
}

try {
```

```
  func();
} catch (err) { // (B)
  assert.equal(err, errorObject);
}
```

22.3.2 The `finally` clause

Let's look at a common use case for `finally`: You have created a resource and want to always destroy it when your are done with it – no matter what happens while working with it. You'd implement that as follows:

```
const resource = createResource();
try {
  // Work with `resource`: errors may be thrown.
} finally {
  resource.destroy();
}
```

The `finally` is always executed – even if an error is thrown (line A):

```
let finallyWasExecuted = false;
assert.throws(
  () => {
    try {
      throw new Error(); // (A)
    } finally {
      finallyWasExecuted = true;
    }
  },
  Error
);
assert.equal(finallyWasExecuted, true);
```

The `finally` is always executed – even if there is a `return` statement (line A):

```
let finallyWasExecuted = false;
function func() {
  try {
    return; // (A)
  } finally {
    finallyWasExecuted = true;
  }
}
func();
assert.equal(finallyWasExecuted, true);
```

22.4 Error classes and their properties

Quoting the ECMAScript specification:

- `Error` [root class]
  - `RangeError`: Indicates a value that is not in the set or range of allowable values.
  - `ReferenceError`: Indicate that an invalid reference value has been detected.
  - `SyntaxError`: Indicates that a parsing error has occurred.
  - `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 classes

Consider `err`, an instance of `Error`:

```
const err = new Error('Hello!');
assert.equal(String(err), 'Error: Hello!');
```

Two properties of `err` are especially useful:

- `.message`: contains just the error message.

  ```
  assert.equal(err.message, 'Hello!');
  ```

- `.stack`: contains a stack trace. It is supported by all mainstream browsers.

  ```
  assert.equal(
  err.stack,
  `
  Error: Hello!
      at Context.<anonymous> (ch_exception-handling.js:1:13)
  `.trim());
  ```

Exercise: Exception handling

`exercises/exception-handling/call_function_test.js`

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 (in other languages, you'd simply use the term "function"; in JavaScript, we need to distinguish between the role "real function" and the entity "ordinary function" that can play that role)
  - Method
  - Constructor function
- A *specialized function* can only play one of those roles. For example:
  - An *arrow function* can only be a real function.
  - A *method* can only be a method.
  - A *class* can only be a constructor function.

The next sections explain what all of those things mean.

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) {
  // ···
}

// Anonymous function expression
const ordinary2 = function (a, b, c) {
  // ···
};

// Named function expression
const ordinary3 = function myName(a, b, c) {
  // `myName` is only accessible in here
};
```

As we have seen in the chapter on variables, function declarations are hoisted, while variable declarations (e.g. via `const`) are not. We'll explore the consequences of that later in this chapter.

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 the chapter on 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` operator explicitly returns a value from the function.

23.2.2 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 inside the current scope
assert.equal(funcDecl(), funcDecl);
```

23.2.3 Roles played by ordinary functions

Consider the following function declaration from the previous section:

```
function add(x, y) {
  return x + y;
}
```

This function declaration creates an ordinary function whose name is `add`. As an ordinary function, `add()` can play three roles:

- Real function: invoked via a function call. It's what most programming languages consider to be simply *a function*.

  ```
  assert.equal(add(2, 1), 3);
  ```

- Method: stored in property, invoked via a method call.

  ```
  const obj = { addAsMethod: add };
  assert.equal(obj.addAsMethod(2, 4), 6);
  ```

- Constructor function/class: invoked via `new`.

  ```
  const inst = new add();
  assert.equal(inst instanceof add, true);
  ```

  (As an aside, the names of classes normally start with capital letters.)

23.3 Specialized functions

Specialized functions are specialized versions of ordinary functions. Each one of them only plays a single role:

- An *arrow function* can only be a real function:

```
const arrow = () => { return 123 };
assert.equal(arrow(), 123);
```

- A *method* can only be a method:

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

- A *class* can only be a constructor function:

```
class MyClass { /* ··· */ }
const inst = new MyClass();
```

Apart from nicer syntax, each kind of specialized function also supports new features, making them better at their job than ordinary functions.

- Arrow functions are explained later in this chapter.
- Methods are explained in the chapter on single objects.
- Classes are explained in the chapter on prototype chains and classes.

Tbl. 23.1 lists the capabilities of ordinary and specialized functions.

Table 23.1: Capabilities of four kinds of functions.

	Ordinary function	Arrow function	Method	Class
Function call	✔	✔	✔	✘
Method call	✔	lexical `this`	✔	✘
Constructor call	✔	✘	✘	✔

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:

- Arrow functions don't have `this` as an implicit parameter. That is almost always what you want if you use a real function, because it avoids an important `this`-related pitfall (for details, consult the chapter on single objects).
- However, I like the function declaration (which produces an ordinary function) syntactically. 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 (details soon).

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 last example demonstrates the first benefit of arrow functions – conciseness. In contrast, this is the same method call, but with a function expression:

```
[1,2,3].map(function (x) { return x+1 });
```

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`.
- That own `this` makes it impossible to access the `this` of the surrounding scope from inside an ordinary function. And that is inconvenient for real functions.

The following code demonstrates a common work-around:

```
const prefixer = {
  prefix: '==> ',
  prefixStringArray(stringArray) {
    const that = this; // (A)
    return stringArray.map(
      function (x) {
        return that.prefix + x; // (B)
      });
  },
};
assert.deepEqual(
  prefixer.prefixStringArray(['a', 'b']),
  ['==> a', '==> b']);
```

In line B, we want to access the `this` of `.prefixStringArray()`. But we can't, since the surrounding ordinary function has its own `this` that *shadows* (blocks access to) the `this` of the method. Therefore, we save the method's `this` in the extra variable `that` (line A) and use that variable in line B.

An arrow function doesn't have `this` as an implicit parameter, it picks up its value from the surroundings. That is, `this` behaves just like any other variable.

```
const prefixer = {
  prefix: '==> ',
  prefixStringArray(stringArray) {
    return stringArray.map(
      x => this.prefix + x);
  },
};
```

To summarize:

- In ordinary functions, `this` is an implicit (*dynamic*) parameter (details in the chapter on single objects).
- Arrow functions get `this` from their surrounding scopes (*lexically*).

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`.

23.4 Hoisting functions

Function declarations are *hoisted* (internally moved to the top):

```
assert.equal(foo(), 123); // OK

function foo() { return 123; }
```

Hoisting lets you call `foo()` before it is declared.

Variable declarations are not hoisted: 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
```

Class declarations are not hoisted, either:

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

class MyClass {}

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

23.4.1 Calling ahead without hoisting

Note that a function `f()` can still call a non-hoisted function `g()` before its declaration – if `f()` is 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 the complete body of a module was executed. Therefore, you rarely need to worry about the order of functions in a module.

23.4.2 A pitfall of hoisting

If you rely on hoisting to call a function before its declaration then you need to be careful that it doesn't access non-hoisted data.

```
hoistedFunc();

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

As before, the problem goes away if you make the function call `hoistedFunc()` at the end.

23.5 Returning values from functions

You use the `return` operator to return values from a function:

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

Another example:

```
function boolToYesNo(bool) {
  if (bool) {
    return 'Yes';
  } else {
    return 'No';
  }
}
```

```
assert.equal(boolToYesNo(true), 'Yes');
assert.equal(boolToYesNo(false), 'No');
```

If, at the end of a function, you haven't returned anything explicitly, JavaScript returns `undefined` for you:

```
function noReturn() {
  // No explicit return
}
assert.equal(noReturn(), undefined);
```

23.6 Parameter handling

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 passed as an argument to another function or a method. This term is used often and broadly in the JavaScript community.

The following is an example of a callback:

```
const myArray = ['a', 'b'];
const callback = (x) => console.log(x);
myArray.forEach(callback);

// Output:
// 'a'
// 'b'
```

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 bar(a, b) {
  // ···
}
```

This is how we force callers to always provide two arguments:

```
function bar(...args) {
  if (args.length !== 2) {
    throw new Error('Please provide exactly 2 arguments!');
  }
  const [a, b] = args;
  // ···
}
```

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.
- Order of parameters doesn't matter (as long as the names are correct).
- Handling more than one optional parameter is more convenient: Callers can easily provide any subset of all optional parameters and don't have to be aware of the ones they omitted (with positional parameters, you have to fill in preceding optional parameters, with `undefined`).

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

The prefix `(...)` of a spread argument is the same as the prefix of a rest parameter. The former is used when calling functions or methods. Its operand must be an iterable object. The iterated values are turned into positional arguments. For example:

```
function func(x, y) {
  console.log(x);
  console.log(y);
}
const someIterable = ['a', 'b'];
func(...someIterable);

// Output:
// 'a'
// 'b'
```

Therefore, spread arguments and rest parameters serve opposite purposes:

- Rest parameters are used when defining functions or methods. They collect arguments in Arrays.
- Spread arguments are used when calling functions or methods. They turn iterable objects into arguments.

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, but once again we are saved by spreading:

```
const arr1 = ['a', 'b'];
const arr2 = ['c', 'd'];

arr1.push(...arr2);
assert.deepEqual(arr1, ['a', 'b', 'c', 'd']);
```

Exercises: Parameter handling

- Positional parameters: `exercises/callables/positional_parameters_test.js`
- Named parameters: `exercises/callables/named_parameters_test.js`

Quiz

See quiz app.

Part VI

Modularity

Chapter 24

Modules

Contents

The current landscape of JavaScript modules is quite diverse: ES6 brought built-in modules, but the module systems that came before them, are still around, too. Understanding the latter helps understand the former, so let's investigate.

24.1 Before modules: scripts

Initially, browsers only had *scripts* – pieces of code that were executed in global scope. As an example, consider an HTML file that loads a *script file* via the following HTML element:

```
<script src="my-library.js"></script>
```

In the script file, we simulate a module:

```
var myModule = function () { // Open IIFE
  // Imports (via global variables)
  var importedFunc1 = otherLibrary1.importedFunc1;
  var importedFunc2 = otherLibrary2.importedFunc2;

  // Body
  function internalFunc() {
    // ···
  }
  function exportedFunc() {
    importedFunc1();
    importedFunc2();
    internalFunc();
  }

  // Exports (assigned to global variable `myModule`)
  return {
    exportedFunc: exportedFunc,
  };
}(); // Close IIFE
```

Before we get to real modules (which were introduced with ES6), all code is written in ES5 (which didn't have `const` and `let`, only `var`).

`myModule` is a global variable. The code that defines the module is wrapped in an *immediately invoked function expression* (IIFE). Creating a function and calling it right away, only has one benefit compared to executing the code directly (without wrapping it): All variables defined inside the IIFE, remain local to its scope and don't become global. At the end, we pick what we want to export and return it via an object literal. This pattern is called the *revealing module pattern* (coined by Christian Heilmann).

This way of simulating modules has several problems:

- Libraries in script files export and import functionality via global variables, which risks name clashes.
- Dependencies are not stated explicitly and there is no built-in way for a script to load the scripts it depends on. Therefore, the web page has to load not just the scripts that are needed by the page, but also the dependencies of those scripts, the dependencies' dependencies, etc. And it has to do so in the right order!

24.2 Module systems created prior to ES6

Prior to ECMAScript 6, JavaScript did not have built-in modules. Therefore, the flexible syntax of the language was used to implement custom module systems *within* the lan-

guage. Two popular ones are CommonJS (targeted at the server side) and AMD (Asynchronous Module Definition, targeted at the client side).

24.2.1 Server side: CommonJS modules

The original CommonJS standard for modules was mainly created for server and desktop platforms. It was the foundation of the module system of Node.js where it achieved incredible popularity. Contributing to that popularity were Node's package manager, npm, and tools that enabled using Node modules on the client side (browserify and webpack).

From now on, I use the terms *CommonJS module* and *Node.js module* interchangeably, even though Node.js has a few additional features. The following is an example of a Node.js module.

```
// Imports
var importedFunc1 = require('other-module1').importedFunc1;
var importedFunc2 = require('other-module2').importedFunc2;

// 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.
- Compact syntax.

24.2.2 Client side: AMD (Asynchronous Module Definition) modules

The AMD module format was created to be easier to use in browsers than the CommonJS format. Its most popular implementation is RequireJS. The following is an example of a RequireJS module.

```
define(['other-module1', 'other-module2'],
  function (otherModule1, otherModule2) {
    var importedFunc1 = otherModule1.importedFunc1;
```

```
    var importedFunc2 = otherModule2.importedFunc2;

    function internalFunc() {
      // ···
    }
    function exportedFunc() {
      importedFunc1();
      importedFunc2();
      internalFunc();
    }

    return {
      exportedFunc: exportedFunc,
    };
});
```

AMD can be characterized as follows:

- Designed for browsers.
- Modules are meant to be loaded asynchronously. That's a crucial requirement for browsers, where code can't wait until a module has finished downloading. It has to be notified once the module is available.
- The syntax is slightly more complicated. On the plus side, AMD modules can be executed directly, without customized creation and execution of source code (think `eval()`). That is not always permitted on the web.

24.2.3 Characteristics of JavaScript modules

Looking at CommonJS and AMD, similarities between JavaScript module systems emerge:

- There is one module per file (AMD also supports more than one module per file).
- Such a file is basically a piece of code that is executed:
  - Exports: That code contains declarations (variables, functions, etc.). By default, those declarations remain local to the module, but you can mark some of them as exports.
  - Imports: The module can import entities from other modules. Those other modules are identified via *module specifiers* (usually paths, occasionally URLs).
- Modules are *singletons*: Even if a module is imported multiple times, only a single instance of it exists.
- No global variables are used. Instead, module specifiers serve as global IDs.

24.3 ECMAScript modules

ECMAScript modules were introduced with ES6: They stand firmly in the tradition of JavaScript modules and share many of the characteristics of existing module systems:

- With CommonJS, ES modules share the compact syntax, better syntax for single exports than for *named exports* (so far, we have only seen named exports) and support for cyclic dependencies.
- With AMD, ES modules share a design for asynchronous loading and configurable module loading (e.g. how specifiers are resolved).

ES modules also have new benefits:

- Their syntax is even more compact than CommonJS's.
- Their modules have a static structure (that can't be changed at runtime). That enables static checking, optimized access of imports, better bundling (delivery of less code) and more.
- Their support for cyclic imports is completely transparent.

This is an example of ES module syntax:

```
import {importedFunc1} from 'other-module1';
import {importedFunc2} from 'other-module2';

function internalFunc() {
  ...
}

export function exportedFunc() {
  importedFunc1();
  importedFunc2();
  internalFunc();
}
```

From now on, "module" means "ECMAScript module".

24.3.1 ECMAScript modules: three parts

ECMAScript modules comprise three parts:

1. Declarative module syntax: What is a module? How are imports and exports declared?
2. The semantics of the syntax: How are the variable bindings handled that are created by imports? How are exported variable bindings handled?
3. A programmatic loader API for configuring module loading.

Parts 1 and 2 were introduced with ES6. Work on Part 3 is ongoing.

24.4 Named exports

Each module can have zero or more named exports.

As an example, consider the following three files:

```
lib/my-math.js
main1.js
main2.js
```

Module `my-math.js` has two named exports: `square` and `MY_CONSTANT`.

```
let notExported = 'abc';
export function square(x) {
  return x * x;
}
export const MY_CONSTANT = 123;
```

Module `main1.js` has a single named import, `square`:

```
import {square} from './lib/my-math.js';
assert.equal(square(3), 9);
```

Module `main2.js` has a so-called *namespace import* – all named exports of `my-math.js` can be accessed as properties of the object `myMath`:

```
import * as myMath from './lib/my-math.js';
assert.equal(myMath.square(3), 9);
```

Exercise: Named exports

`exercises/modules/export_named_test.js`

24.5 Default exports

Each module can have at most one default export. The idea is that the module *is* the default-exported value. A module can have both named exports and a default export, but it's usually better to stick to one export style per module.

As an example for default exports, consider the following two files:

```
my-func.js
main.js
```

Module `my-func.js` has a default export:

```
export default function () {
  return 'Hello!';
}
```

Module `main.js` default-imports the exported function:

```
import myFunc from './my-func.js';
assert.equal(myFunc(), 'Hello!');
```

Note the syntactic difference: The curly braces around named imports indicate that we are reaching *into* the module, while a default import *is* the module.

The most common use case for a default export is a module that contains a single function or a single class.

24.5.1 The two styles of default-exporting

There are two styles of doing default exports.

First, you can label existing declarations with `export default`:

```
export default function foo() {} // no semicolon!
export default class Bar {} // no semicolon!
```

Second, you can directly default-export values. In that style, `export default` is itself much like a declaration.

```
export default 'abc';
export default foo();
export default /^xyz$/;
export default 5 * 7;
export default { no: false, yes: true };
```

Why are there two default export styles? The reason is that `export default` can't be used to label `const`: `const` may define multiple values, but `export default` needs exactly one value.

```
// Not legal JavaScript!
export default const foo = 1, bar = 2, baz = 3;
```

With this hypothetical code, you don't know which one of the three values is the default export.

Exercise: Default exports

`exercises/modules/export_default_test.js`

24.6 Naming modules

There are no established best practices for naming module files and the variables they are imported into.

In this chapter, I've used the following naming style:

- The names of module files are dash-cased and start with lowercase letters:

  ```
  ./my-module.js
  ./some-func.js
  ```

- The names of namespace imports are lowercased and camel-cased:

  ```
  import * as myModule from './my-module.js';
  ```

- The names of default imports are lowercased and camel-cased:

```
import someFunc from './some-func.js';
```

What are the rationales behind this style?

- npm doesn't allow uppercase letters in package names (source). Thus, we avoid camel case, so that "local" files have names that are consistent with those of npm packages.
- There are clear rules for translating 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.js';
```

But that style does not work for default imports: I like underscore-casing for namespace objects, but it is not a good choice for functions etc.

24.7 Imports are read-only views on exports

So far, we have used imports and exports intuitively and everything seems to have worked as expected. But now it is time to take a closer look at how imports and exports are really related.

Consider the following two modules:

```
counter.js
main.js
```

`counter.js` exports a (mutable!) variable and a function:

```
export let counter = 3;
export function incCounter() {
  counter++;
}
```

`main.js` name-imports both exports. When we use `incCounter()`, we discover that the connection to `counter` is live – we can always access the live state of that variable:

```
import { counter, incCounter } from './counter.js';

// The imported value `counter` is live
assert.equal(counter, 3);
incCounter();
assert.equal(counter, 4);
```

Note that, while the connection is live and we can read `counter`, we cannot change this variable (e.g. via `counter++`).

Why do ES modules behave this way?

First, it is easier to split modules, because previously shared variables can become exports.

Second, this behavior is crucial for cyclic imports. The exports of a module are known before executing it. Therefore, if a module L and a module M import each other, cyclically, the following steps happen:

- The execution of L starts.
  - L imports M. L's imports point to uninitialized slots inside M.
  - L's body is not executed, yet.
- The execution of M starts (triggered by the import).
  - M imports L.
  - The body of M is executed. Now L's imports have values (due to the live connection).
- The body of L is executed. Now M's imports have values.

Cyclic imports are something that you should avoid as much as possible, but they can arise in complex systems or when refactoring systems. It is important that things don't break when that happens.

24.8 Module specifiers

One key rule is:

> All ES module specifiers must be valid URLs and point to real files.

Beyond that, everything is still somewhat in flux.

24.8.1 Categories of module specifiers

Before we get into further details, we need to establish the following categories of module specifiers (which originated with CommonJS):

- Relative paths: start with a dot. Examples:

  ```
  './some/other/module.js'
  '../../lib/counter.js'
  ```

- Absolute paths: start with slashes. Example:

  ```
  '/home/jane/file-tools.js'
  ```

- Full URLs: include protocols (technically, paths are URLs, too). Example:

  ```
  'https://example.com/some-module.js'
  ```

- Bare paths: do not start with dots, slashes or protocols. In CommonJS modules, bare paths rarely have file name extensions.

  ```
  'lodash'
  'mylib/string-tools'
  'foo/dist/bar.js'
  ```

24.8.2 ES module specifiers in Node.js

Support for ES modules in Node.js is work in progress. The current plan (as of 2018-12-20) is to handle module specifiers as follows:

- Relative paths, absolute paths and full URLs work as expected. They all must point to real files.
- Bare paths:
  - Built-in modules (`path`, `fs`, etc.) can be imported via bare paths.
  - All other bare paths must point to files: `'foo/dist/bar.js'`
- The default file name extension for ES modules is `.mjs` (there will probably be a way to switch to a different extension, per package).

24.8.3 ES module specifiers in browsers

Browsers handle module specifiers as follows:

- Relative paths, absolute paths and full URLs work as expected. They all must point to real files.
- How bare paths will end up being handled is not yet clear. You may eventually be able to map them to other specifiers via lookup tables.
- The file name extensions of modules don't matter, as long as they are served with the content type `text/javascript`.

Note that bundling tools such as browserify and webpack that compile multiple modules into single files are less restrictive with module specifiers than browsers, because they operate at compile time, not at runtime.

24.9 Syntactic pitfall: importing is not destructuring

Both importing and destructuring look similar:

```
import {foo} from './bar.js'; // import
const {foo} = require('./bar.js'); // destructuring
```

But they are quite different:

- 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.js'; // importing
  const {foo: f} = require('./bar.js'); // destructuring
  ```

  Rationale: Destructuring is reminiscent of an object literal (incl. nesting), while importing evokes the idea of renaming.

24.10 Preview: loading modules dynamically

So far, the only way to import a module has been via an `import` statement. Limitations of those statements:

- You must use them at the top level of a module. That is, you can't, e.g., 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, you can't retrieve or assemble a specifier dynamically.

An upcoming JavaScript feature changes that: The `import()` operator, which is used as if it were an asynchronous function (it is only an operator, because it needs implicit access to the URL of the current module).

Consider the following files:

```
lib/my-math.js
main1.js
main2.js
```

We have already seen module `my-math.js`:

```
let notExported = 'abc';
export function square(x) {
  return x * x;
}
export const MY_CONSTANT = 123;
```

This is what using `import()` looks like in `main1.js`:

```
const dir = './lib/';
const moduleSpecifier = dir + 'my-math.js';

function loadConstant() {
  return import(moduleSpecifier)
  .then(myMath => {
    const result = myMath.MY_CONSTANT;
    assert.equal(result, 123);
    return result;
  });
}
```

Method `.then()` is part of *Promises*, a mechanism for handling asynchronous results, which is covered later in this book.

Two things in this code weren't possible before:

- We are importing inside a function (not at the top level).
- The module specifier comes from a variable.

Next, we'll implement the exact same functionality in `main2.js`, but via a so-called *async function*, which provides nicer syntax for Promises.

```
const dir = './lib/';
const moduleSpecifier = dir + 'my-math.js';

async function loadConstant() {
  const myMath = await import(moduleSpecifier);
  const result = myMath.MY_CONSTANT;
  assert.equal(result, 123);
  return result;
}
```

Alas, `import()` isn't a standard part of JavaScript yet, but probably will be, relatively soon. That means that support is mixed and may be inconsistent.

24.11 Further reading

- More on `import()`: "ES proposal: `import()` – dynamically importing ES modules" on 2ality.
- For an in-depth look at ECMAScript modules, consult "Exploring ES6".

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:** How do *objects*, JavaScript's basic OOP building blocks, work in isolation?
2. Prototype chains: Each object has a chain of zero or more *prototype objects*. Prototypes are JavaScript's core inheritance mechanism.
3. Classes: JavaScript's *classes* are factories for objects. The relationship between a class and its instances is based on prototypal inheritance.
4. Subclassing: The relationship between a *subclass* and its *superclass* is also based on prototypal inheritance.

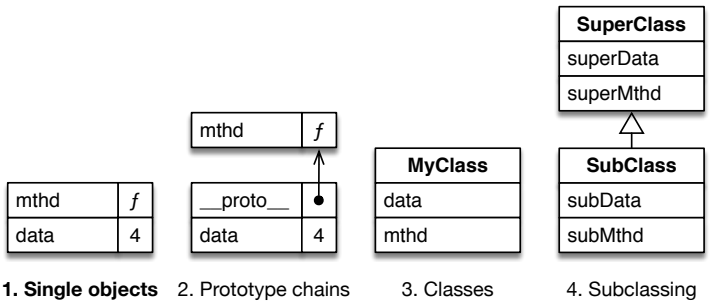

Figure 25.1: This book introduces object-oriented programming in JavaScript in four steps.

25.1 The two roles of objects in JavaScript

In JavaScript, an object is a set of key-value entries that are called *properties*.

Objects play two roles in JavaScript:

- Records: Objects-as-records have a fixed number of properties, whose keys are known at development time. Their values can have different types. This way of using objects is covered first in this chapter.

- Dictionaries: Objects-as-dictionaries have a variable number of properties, whose keys are not known at development time. All of their values have the same type. It is usually better to use Maps as dictionaries than objects (which is covered later in this chapter).

25.2 Objects as records

25.2.1 Object literals: properties

Objects as records are created via so-called *object literals*. Object literals are a stand-out feature of JavaScript: they allow you to directly create objects. No need for classes! This is an example:

```
const jane = {
  first: 'Jane',
  last: 'Doe', // optional trailing comma
};
```

In the example, we created an object via an object literal, which starts with a curly brace and ends with a curly brace: `{ ··· }`. 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'`.

If they are written this way, property keys must follow the rules of JavaScript variable names, with the exception that reserved words are allowed.

The properties are accessed as follows:

```
assert.equal(jane.first, 'Jane'); // get property .first
jane.first = 'John'; // set property .first
assert.equal(jane.first, 'John');
```

25.2.2 Object literals: property value shorthands

Whenever the value of a property is defined via a variable name and that name is the same as the key, you can omit the key.

```
const x = 4;
const y = 1;

assert.deepEqual(
  { x, y },
  { x: x, y: y }
);
```

25.2.3 Terminology: property keys, property names, property symbols

Given that property keys can be strings and symbols, the following distinction is made:

- If a property key is a string, it is also called a *property name*.
- If a property key is a symbol, it is also called a *property symbol*.

This terminology is used in the JavaScript standard library ("own" means "not inherited" and is explained in the next chapter):

- `Object.keys(obj)`: returns all property keys of `obj`
- `Object.getOwnPropertyNames(obj)`
- `Object.getOwnPropertySymbols(obj)`

25.2.4 Getting properties

This is how you *get* (read) a property:

```
obj.propKey
```

If `obj` does not have a property whose key is `propKey`, this expression evaluates to `undefined`:

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

25.2.5 Setting properties

This is how you *set* (write to) a property:

```
obj.propKey = value;
```

If `obj` already has a property whose key is `propKey`, this statement changes that property. Otherwise, it creates a new property:

```
const obj = {};
assert.deepEqual(
  Object.keys(obj), []);

obj.propKey = 123;
assert.deepEqual(
  Object.keys(obj), ['propKey']);
```

25.2.6 Object literals: methods

The following code shows how to create the method `.describe()` via an object literal:

```
const jane = {
  first: 'Jane', // data property
  says(text) {   // method
```

```
    return `${this.first} says “${text}”`; // (A)
  }, // comma as separator (optional at end)
};
assert.equal(jane.says('hello'), 'Jane says “hello”');
```

During the method call `jane.says('hello')`, `jane` is called the *receiver* of the method call and assigned to the special variable `this`. That enables method `.says()` to access the sibling property `.first` in line A.

25.2.7 Object literals: accessors

There are two kinds of accessors in JavaScript:

- A *getter* is a method that is invoked by *getting* (reading) a property.
- A *setter* is a method that is invoked by *setting* (writing) a property.

25.2.7.1 Getters

A getter is created by prefixing a method definition with the keyword `get`:

```
const jane = {
  first: 'Jane',
  last: 'Doe',
  get full() {
    return `${this.first} ${this.last}`;
  },
};

assert.equal(jane.full, 'Jane Doe');
jane.first = 'John';
assert.equal(jane.full, 'John Doe');
```

25.2.7.2 Setters

A setter is created by prefixing a method definition with the keyword `set`:

```
const jane = {
  first: 'Jane',
  last: 'Doe',
  set full(fullName) {
    const parts = fullName.split(' ');
    this.first = parts[0];
    this.last = parts[1];
  },
};

jane.full = 'Richard Roe';
```

```
assert.equal(jane.first, 'Richard');
assert.equal(jane.last, 'Roe');
```

Exercise: Creating an object via an object literal

`exercises/single-objects/color_point_object_test.js`

25.3 Spreading into object literals (...)

We have already seen spreading (`...`) being used in function calls, where it turns the contents of an iterable into arguments.

Inside an object literal, a *spread property* adds the properties of another object to the current one:

```
> const obj = {foo: 1, bar: 2};
> {...obj, baz: 3}
{ foo: 1, bar: 2, baz: 3 }
```

If property keys clash, the property that is mentioned last "wins":

```
> const obj = {foo: 1, bar: 2, baz: 3};
> {...obj, foo: true}
{ foo: true, bar: 2, baz: 3 }
> {foo: true, ...obj}
{ foo: 1, bar: 2, baz: 3 }
```

25.3.1 Use case for spreading: copying objects

You can use spread to create a copy of an object:

```
const copy = {...obj};
```

Caveat – the copy is shallow:

```
const original = { a: 1, b: {foo: true} };
const copy = {...original};

// The first level is a true copy:
assert.deepEqual(
  copy, { a: 1, b: {foo: true} });
original.a = 2;
assert.deepEqual(
  copy, { a: 1, b: {foo: true} }); // no change

// Deeper levels are not copied:
original.b.foo = false;
// The value of property `b` is shared
```

```
// between original and copy.
assert.deepEqual(
  copy, { a: 1, b: {foo: false} });
```

25.3.2 Use case for spreading: default values for missing properties

If one of the inputs of your code is an object with data, you can make properties optional if you specify default values for them. One technique for doing so is via an object whose properties contain the default values. In the following example, that object is `DEFAULTS`:

```
const DEFAULTS = {foo: 'a', bar: 'b'};
const providedData = {foo: 1};

const allData = {...DEFAULTS, ...providedData};
assert.deepEqual(allData, {foo: 1, bar: 'b'});
```

The result, the object `allData`, is created by creating a copy of `DEFAULTS` and overriding its properties with those of `providedData`.

But you don't need an object to specify the default values, you can also specify them inside the object literal, individually:

```
const providedData = {foo: 1};

const allData = {foo: 'a', bar: 'b', ...providedData};
assert.deepEqual(allData, {foo: 1, bar: 'b'});
```

25.3.3 Use case for spreading: non-destructively changing properties

So far, we have encountered one way of changing a property of an object: We *set* it and mutate the object. That is, this way of changing a property is *destructive*

With spreading, you can change a property *non-destructively*: You make a copy of the object where the property has a different value.

For example, this code non-destructively updates property `.foo`:

```
const obj = {foo: 'a', bar: 'b'};
const updatedObj = {...obj, foo: 1};
assert.deepEqual(updatedObj, {foo: 1, bar: 'b'});
```

Exercise: Non-destructively updating properties via spreading (fixed key)

`exercises/single-objects/update_name_test.js`

25.4 Methods

25.4.1 Methods are properties whose values are functions

Let's revisit the example that was used to introduce methods:

```
const jane = {
  first: 'Jane',
  says(text) {
    return `${this.first} says “${text}”`;
  },
};
```

Somewhat surprisingly, methods are functions:

```
assert.equal(typeof jane.says, 'function');
```

Why is that? Remember that, in the chapter on callable entities, we learned that ordinary functions play several roles. *Method* is one of those roles. Therefore, under the hood, `jane` roughly looks as follows.

```
const jane = {
  first: 'Jane',
  says: function (text) {
    return `${this.first} says “${text}”`;
  },
};
```

25.4.2 `.call()`: explicit parameter `this`

Remember that each function `someFunc` is also an object and therefore has methods. One such method is `.call()` – it lets you call functions while specifying `this` explicitly:

```
someFunc.call(thisValue, arg1, arg2, arg3);
```

25.4.2.1 Methods and `.call()`

If you make a method call, `this` is always an implicit parameter:

```
const obj = {
  method(x) {
    assert.equal(this, obj); // implicit parameter
    assert.equal(x, 'a');
  },
};

obj.method('a');
// Equivalent:
obj.method.call(obj, 'a');
```

As an aside, that means that there are actually two different dot operators:

1. One for accessing properties: `obj.prop`
2. One for making method calls: `obj.prop()`

They are different in that (2) is not just (1), followed by the function call operator `()`. Instead, (2) additionally specifies a value for `this` (as shown in the previous example).

25.4.2.2 Functions and `.call()`

However, `this` is also an implicit parameter if you function-call an ordinary function:

```
function func(x) {
  assert.equal(this, undefined); // implicit parameter
  assert.equal(x, 'a');
}

func('a');
// Equivalent:
func.call(undefined, 'a');
```

That is, during a function call, ordinary functions have a `this`, but it is set to `undefined`, which indicates that it doesn't really have a purpose here.

Next, we'll examine the pitfalls of using `this`. Before we can do that, we need one more tool: the method `.bind()` of functions.

25.4.3 `.bind()`: pre-filling `this` and parameters of functions

`.bind()` is another method of function objects. This method is invoked as follows.

```
const boundFunc = someFunc.bind(thisValue, arg1, arg2, arg3);
```

`.bind()` returns a new function `boundFunc()`. Calling that function invokes `someFunc()` with `this` set to `thisValue` and these parameters: `arg1`, `arg2`, `arg3`, followed by the parameters of `boundFunc()`.

That is, the following two function calls are equivalent:

```
boundFunc('a', 'b')
someFunc.call(thisValue, arg1, arg2, arg3, 'a', 'b')
```

Another way of pre-filling `this` and parameters, is via an arrow function:

```
const boundFunc2 = (...args) =>
  someFunc.call(thisValue, arg1, arg2, arg3, ...args);
```

Therefore, `.bind()` can be implemented as a real function as follows:

```
function bind(func, thisValue, ...boundArgs) {
  return (...args) =>
    func.call(thisValue, ...boundArgs, ...args);
}
```

25.4.3.1 Example: binding a real function

Using `.bind()` for real functions is somewhat unintuitive, because you have to provide a value for `this`. That value is usually `undefined`, mirroring what happens during function calls.

In the following example, we create `add8()`, a function that has one parameter, by binding the first parameter of `add()` to 8.

```
function add(x, y) {
  return x + y;
}

const add8 = add.bind(undefined, 8);
assert.equal(add8(1), 9);
```

25.4.3.2 Example: binding a method

In the following code, we turn method `.says()` into the stand-alone function `func()`:

```
const jane = {
  first: 'Jane',
  says(text) {
    return `${this.first} says “${text}”`; // (A)
  },
};

const func = jane.says.bind(jane, 'hello');
assert.equal(func(), 'Jane says “hello”');
```

Setting `this` to `jane` via `.bind()` is crucial here. Otherwise, `func()` wouldn't work properly, because `this` is used in line A.

25.4.4 `this` pitfall: extracting methods

We now know quite a bit about functions and methods and are ready to take a look at the biggest pitfall involving methods and `this`: function-calling a method extracted from an object can fail if you are not careful.

In the following example, we fail when we extract method `jane.says()`, store it in the variable `func` and function-call `func()`.

```
const jane = {
  first: 'Jane',
  says(text) {
    return `${this.first} says “${text}”`;
  },
};
const func = jane.says; // extract the method
```

```
assert.throws(
  () => func('hello'), // (A)
  {
    name: 'TypeError',
    message: "Cannot read property 'first' of undefined",
  });
```

The function call in line A is equivalent to:

```
assert.throws(
  () => jane.says.call(undefined, 'hello'), // `this` is undefined!
  {
    name: 'TypeError',
    message: "Cannot read property 'first' of undefined",
  });
```

So how do we fix this? We need to use `.bind()` to extract method `.says()`:

```
const func2 = jane.says.bind(jane);
assert.equal(func2('hello'), 'Jane says “hello”');
```

The `.bind()` ensures that `this` is always `jane` when we call `func()`.

You can also use arrow functions to extract methods:

```
const func3 = text => jane.says(text);
assert.equal(func3('hello'), 'Jane says “hello”');
```

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(elem) {
    elem.addEventListener('click', this.handleClick); // (A)
  }
  handleClick(event) {
    alert('Clicked!');
  }
}
```

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.js`

25.4.5 `this` pitfall: accidentally shadowing `this`

Accidentally shadowing `this` is only an issue if you use ordinary functions.

Consider the following problem: When you are inside an ordinary function, you can't access the `this` of the surrounding scope, because the ordinary function has its own `this`. In other words: a variable in an inner scope hides a variable in an outer scope. That is called *shadowing*. The following code is an example:

```
const obj = {
  name: 'Jane',
  sayHiTo(friends) {
    return friends.map(
      function (friend) { // (A)
        return `${this.name} says hi to ${friend}`; // (B)
      });
  }
};
assert.throws(
  () => obj.sayHiTo(['Tarzan', 'Cheeta']),
  {
    name: 'TypeError',
    message: "Cannot read property 'name' of undefined",
  });
```

Why the error? The `this` in line B isn't the `this` of `.sayHiTo()`, it is the `this` of the ordinary function starting in line B.

There are several ways to fix this. The easiest is to use an arrow function – which doesn't have its own `this`, so shadowing is not an issue.

```
const obj = {
  name: 'Jane',
  sayHiTo(friends) {
    return friends.map(
      (friend) => {
        return `${this.name} says hi to ${friend}`;
      });
  }
};
assert.deepEqual(
  obj.sayHiTo(['Tarzan', 'Cheeta']),
  ['Jane says hi to Tarzan', 'Jane says hi to Cheeta']);
```

25.4.6 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.

Let's break down this rule:

- If all real functions are arrow functions, the second pitfall can never occur.
- Using method definitions means that you'll only see `this` inside methods, which makes this feature less confusing.

However, even though I don't use (ordinary) function *expressions*, anymore, I do like function *declarations* syntactically. You can use them safely if you don't refer to `this` inside them. The checking tool ESLint has a rule that helps with that.

Alas, there is no simple way around the first pitfall: Whenever you extract a method, you have to be careful and do it properly. For example, by binding `this`.

25.4.7 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`
  - Arrow functions: `this` is same as in surrounding scope (lexical `this`)
- Method call: `this` is receiver of call
- `new`: `this` refers to newly created instance

You can also access `this` in all common top-level scopes:

- `<script>` element: `this === window`
- ES modules: `this === undefined`
- CommonJS modules: `this === module.exports`

However, I like to pretend that you can't access `this` in top-level scopes, because top-level `this` is confusing and not that useful.

25.5 Objects as dictionaries

Objects work best as records. But before ES6, JavaScript did not have a data structure for dictionaries (ES6 brought Maps). Therefore, objects had to be used as dictionaries. As a consequence, keys had to be strings, but values could have arbitrary types.

We first look at features of objects that are related to dictionaries, but also occasionally useful for objects-as-records. This section concludes with tips for actually using objects as dictionaries (spoiler: avoid, use Maps if you can).

25.5.1 Arbitrary fixed strings as property keys

When going from objects-as-records to objects-as-dictionaries, one important change is that we must be able to use arbitrary strings as property keys. This subsection explains how to achieve that for fixed string keys. The next subsection explains how to dynamically compute arbitrary keys.

So far, we have only seen legal JavaScript identifiers as property keys (with the exception of symbols):

```
const obj = {
  mustBeAnIdentifier: 123,
};

// Get property
assert.equal(obj.mustBeAnIdentifier, 123);

// Set property
obj.mustBeAnIdentifier = 'abc';
assert.equal(obj.mustBeAnIdentifier, 'abc');
```

Two techniques allow us to use arbitrary strings as property keys.

First – when creating property keys via object literals, we can quote property keys (with single or double quotes):

```
const obj = {
  'Can be any string!': 123,
};
```

Second – when getting or setting properties, we can use square brackets with strings inside them:

```
// Get property
assert.equal(obj['Can be any string!'], 123);

// Set property
obj['Can be any string!'] = 'abc';
assert.equal(obj['Can be any string!'], 'abc');
```

You can also quote the keys of methods:

```
const obj = {
  'A nice method'() {
    return 'Yes!';
  },
};

assert.equal(obj['A nice method'](), 'Yes!');
```

25.5.2 Computed property keys

So far, we were limited by what we could do with property keys inside object literals: They were always fixed and they were always strings. We can dynamically compute arbitrary keys if we put expressions in square brackets:

```
const obj = {
  ['Hello world!']: true,
  ['f'+'o'+'o']: 123,
  [Symbol.toStringTag]: 'Goodbye', // (A)
};

assert.equal(obj['Hello world!'], true);
assert.equal(obj.foo, 123);
assert.equal(obj[Symbol.toStringTag], 'Goodbye');
```

The main use case for computed keys is having symbols as property keys (line A).

Note that the square brackets operator for getting and setting properties works with arbitrary expressions:

```
assert.equal(obj['f'+'o'+'o'], 123);
assert.equal(obj['==> foo'.slice(-3)], 123);
```

Methods can have computed property keys, too:

```
const methodKey = Symbol();
const obj = {
  [methodKey]() {
    return 'Yes!';
  },
};

assert.equal(obj[methodKey](), 'Yes!');
```

We are now switching back to fixed property keys, but you can always use square brackets if you need computed property keys.

> **Exercise: Non-destructively updating properties via spreading (computed key)**
>
> `exercises/single-objects/update_property_test.js`

25.5.3 The `in` operator: is there a property with a given key?

The `in` operator checks if an object has a property with a given key:

```
const obj = {
  foo: 'abc',
  bar: 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.unknownKey ? 'exists' : 'does not exist',
  'does not exist');
assert.equal(
  obj.foo ? 'exists' : 'does not exist',
  'exists');
```

The previous check works, because reading a non-existent property returns `undefined`, which is falsy. And because `obj.foo` is truthy.

There is, however, one important caveat: Truthiness checks fail if the property exists, but has a falsy value (`undefined`, `null`, `false`, `0`, `""`, etc.):

```
assert.equal(
  obj.bar ? 'exists' : 'does not exist',
  'does not exist'); // 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 Dictionary pitfalls

If you use plain objects (created via object literals) as dictionaries, you have to look out for two pitfalls.

The first pitfall is that the `in` operator also finds inherited properties:

```
const dict = {};
assert.equal('toString' in dict, true);
```

We want `dict` to be treated as empty, but the `in` operator detects the properties it inherits from its prototype, `Object.prototype`.

The second pitfall is that you can't use the property key `__proto__`, because it has special powers (it sets the prototype of the object):

```
const dict = {};

dict['__proto__'] = 123;
// No property was added to dict:
assert.deepEqual(Object.keys(dict), []);
```

So how do we navigate around these pitfalls?

- Whenever you can, use Maps. They are the best solution for dictionaries.
- If you can't: use a library for objects-as-dictionaries that does everything safely.
- If you can't: use an object without a prototype. That eliminates the two pitfalls in modern JavaScript.

  ```
  const dict = Object.create(null); // no prototype

  assert.equal('toString' in dict, false);

  dict['__proto__'] = 123;
  assert.deepEqual(Object.keys(dict), ['__proto__']);
  ```

Exercise: Using an object as a dictionary

`exercises/single-objects/simple_dict_test.js`

25.5.6 Listing property keys

Table 25.1: Standard library methods for listing *own* (non-inherited) property keys. All of them return Arrays with strings and/or symbols.

	enumerable	non-e.	string	symbol
`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 the distinction between property keys (strings and symbols), property names (only strings) and property symbols (only symbols) that we discussed previously.

Enumerability is an *attribute* of properties. By default, properties are enumerable, but

there are ways to change that (shown in the next example and described in slightly more detail later).

For example:

```
const enumerableSymbolKey = Symbol('enumerableSymbolKey');
const nonEnumSymbolKey = Symbol('nonEnumSymbolKey');

// We create the enumerable properties via an object literal
const obj = {
  enumerableStringKey: 1,
  [enumerableSymbolKey]: 2,
}

// For the non-enumerable properties,
// we need a more powerful tool:
Object.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 ]);
```

25.5.7 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]);
```

25.5.8 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.js`

25.5.9 Properties are listed deterministically

Own (non-inherited) properties of objects are always listed in the following order:

1. Properties with integer indices (e.g. Array indices)
   - In ascending numeric order
2. Remaining properties with string keys
   - In the order in which they were added
3. Properties with symbol keys
   - In the order in which they were added

The following example demonstrates how property keys are sorted according to these rules:

```
> Object.keys({b:0,a:0, 2:0,1:0})
[ '1', '2', 'b', 'a' ]
```

(You can look up the details in the spec.)

25.5.10 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,
    }
  );
```

It does the opposite of `Object.entries()`.

Next, we'll use `Object.entries()` and `Object.fromEntries()` to implement several tool functions from the library Underscore.

25.5.10.1 Example: `pick(object, ...keys)`

`pick()` removes all properties from `object` whose keys are not among `keys`. The removal is *non-destructive*: `pick()` creates a modified copy and does not change the original. For example:

```
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);
}
```

25.5.10.2 Example: `invert(object)`

`invert()` non-destructively swaps the keys and the values of an object:

```
assert.deepEqual(
  invert({a: 1, b: 2, c: 3}),
  {1: 'a', 2: 'b', 3: 'c'}
);
```

We can implement it like this:

```
function invert(object) {
  const mappedEntries = Object.entries(object)
    .map(([key, value]) => [value, key]);
  return Object.fromEntries(mappedEntries);
}
```

25.5.10.3 A simple implementation of `Object.fromEntries()`

`Object.fromEntries()` could be implemented as follows (I've omitted a few checks):

```
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);
    }
    Object.defineProperty(result, coercedKey, {
      value,
      writable: true,
      enumerable: true,
      configurable: true,
    });
  }
  return result;
}
```

Notes:

- `Object.defineProperty()` is explained later in this chapter.
- The official polyfill is available via the npm package `object.fromentries`.

> **Exercise: `Object.entries()` and `Object.fromEntries()`**
>
> `exercises/single-objects/omit_properties_test.js`

25.6 Standard methods

`Object.prototype` defines several standard methods that can be overridden. Two important ones are:

- `.toString()`
- `.valueOf()`

Roughly, `.toString()` configures how objects are converted to strings:

```
> String({toString() { return 'Hello!' }})
'Hello!'
> String({})
'[object Object]'
```

And `.valueOf()` configures how objects are converted to numbers:

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

25.7 Advanced topics

The following subsections give a brief overviews of topics that are beyond the scope of this book.

25.7.1 `Object.assign()`

`Object.assign()` is a tool method:

```
Object.assign(target, source_1, source_2, ···)
```

This expression (destructively) merges `source_1` into `target`, then `source_2` etc. At the end, it returns `target`. For example:

```
const target = { foo: 1 };
const result = Object.assign(
  target,
  {bar: 2},
  {baz: 3, bar: 4});
assert.deepEqual(
  result, { foo: 1, bar: 4, baz: 3 });
// target was changed!
assert.deepEqual(target, result);
```

The use cases for `Object.assign()` are similar to those for spread properties. In a way, it spreads destructively.

For more information on `Object.assign()`, consult "Exploring ES6".

25.7.2 Freezing objects

`Object.freeze(obj)` makes `obj` immutable: You can't change or add properties or change the prototype of `obj`.

For example:

```
const frozen = Object.freeze({ x: 2, y: 5 });
assert.throws(
  () => { frozen.x = 7 },
  {
    name: 'TypeError',
    message: /^Cannot assign to read only property 'x'/,
  });
```

There is one caveat: `Object.freeze(obj)` freezes shallowly. That is, only the properties of `obj` are frozen, but not objects stored in properties.

For more information on `Object.freeze()`, consult "Speaking JavaScript".

25.7.3 Property attributes and property descriptors

Just as objects are composed of properties, properties are composed of *attributes*. That is, you can configure more than just the value of a property – which is just one of several attributes. Other attributes include:

- `writable`: Is it possible to change the value of the property?
- `enumerable`: Is the property listed by `Object.keys()`?

When you are using one of the operations for accessing property attributes, attributes are specified via *property descriptors*: objects where each property represents one attribute. For example, this is how you read the attributes of a property `obj.foo`:

```
const obj = { foo: 123 };
assert.deepEqual(
  Object.getOwnPropertyDescriptor(obj, 'foo'),
  {
    value: 123,
    writable: true,
    enumerable: true,
    configurable: true,
  });
```

And this is how you set the attributes of a property `obj.bar`:

```
const obj = {
  foo: 1,
  bar: 2,
};

assert.deepEqual(Object.keys(obj), ['foo', 'bar']);

// Hide property `bar` from Object.keys()
Object.defineProperty(obj, 'bar', {
  enumerable: false,
});

assert.deepEqual(Object.keys(obj), ['foo']);
```

For more on property attributes and property descriptors, consult "Speaking JavaScript".

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: How do *objects*, JavaScript's basic OOP building blocks, work in isolation?

2. **Prototype chains:** Each object has a chain of zero or more *prototype objects*. Prototypes are JavaScript's core inheritance mechanism.
3. **Classes:** JavaScript's *classes* are factories for objects. The relationship between a class and its instances is based on prototypal inheritance.
4. **Subclassing:** The relationship between a *subclass* and its *superclass* is also based on prototypal inheritance.

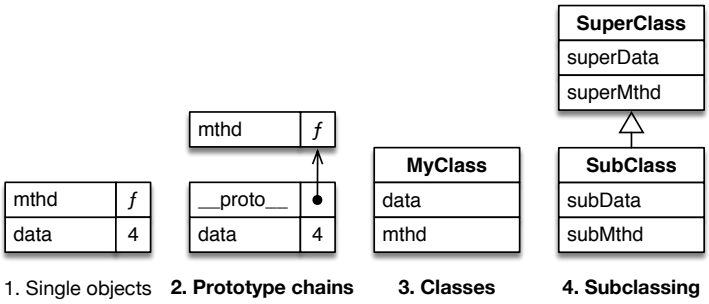

Figure 26.1: This book introduces object-oriented programming in JavaScript in four steps.

26.1 Prototype chains

Prototypes are JavaScript's only inheritance mechanism: Each object has a prototype that is either `null` or an object. In the latter case, the object inherits all of the prototype's properties.

In an object literal, you can set the prototype via the special property `__proto__`:

```
const proto = {
  protoProp: 'a',
};
const obj = {
  __proto__: proto,
  objProp: 'b',
};

// obj inherits .protoProp:
assert.equal(obj.protoProp, 'a');
assert.equal('protoProp' in obj, true);
```

Given that a prototype object can have a prototype itself, we get a chain of objects – the so-called *prototype chain*. That means that inheritance gives us the impression that we are dealing with single objects, but we are actually dealing with chains of objects.

Fig. 26.2 shows what the prototype chain of `obj` looks like.

Non-inherited properties are called *own properties*. `obj` has one own property, `.objProp`.

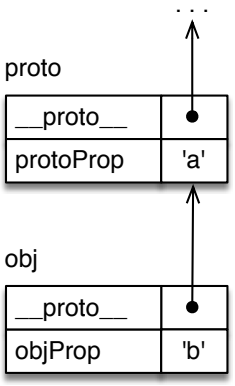

Figure 26.2: `obj` starts a chain of objects that continues with `proto` and other objects.

26.1.1 Pitfall: only first member of prototype chain is mutated

One aspect of prototype chains that may be counter-intuitive is that setting *any* property via an object – even an inherited one – only changes that object – never one of the prototypes.

Consider the following object `obj`:

```
const proto = {
  protoProp: 'a',
};
const obj = {
  __proto__: proto,
  objProp: 'b',
};
```

When we set the inherited property `obj.protoProp` in line A, we "change" it by creating an own property: When reading `obj.protoProp`, the own property is found first and its value overrides the value of the inherited property.

```
assert.deepEqual(Object.keys(obj), ['objProp']);

obj.protoProp = 'x'; // (A)

// We created a new own property:
assert.deepEqual(Object.keys(obj), ['objProp', 'protoProp']);

// The inherited property itself is unchanged:
assert.equal(proto.protoProp, 'a');
```

The prototype chain of `obj` is depicted in fig. 26.3.

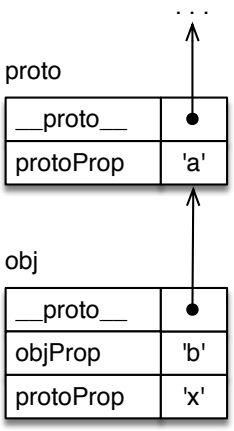

Figure 26.3: The own property `.protoProp` of `obj` overrides the property inherited from `proto`.

26.1.2 Tips for working with prototypes (advanced)

26.1.2.1 Avoid `__proto__` (except in object literals)

I recommend to avoid the special property `__proto__`: It is implemented via a getter and a setter in `Object.prototype` and therefore only available if `Object.prototype` is in the prototype chain of an object. That is usually the case, but to be safe, you can use these alternatives:

- The best way to set a prototype is when creating an object. E.g. via:

  ```
  Object.create(proto: Object) : Object
  ```

  If you have to, you can use `Object.setPrototypeOf()` to change the prototype of an existing object.

- The best way to get a prototype is via the following method:

  ```
  Object.getPrototypeOf(obj: Object) : Object
  ```

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);
```

Note that `__proto__` in object literals is different. There, it is a built-in feature and always safe to use.

26.1.2.2 Check: is an object a prototype of another one?

A looser definition of "`o` is a prototype of `p`" is "`o` is in the prototype chain of `p`". This relationship can be checked via:

```
p.isPrototypeOf(o)
```

For example:

```
const p = {};
const o = {__proto__: p};

assert.equal(p.isPrototypeOf(o), true);
assert.equal(o.isPrototypeOf(p), false);

// Object.prototype is almost always in the prototype chain
// (more on that later)
assert.equal(Object.prototype.isPrototypeOf(p), true);
```

26.1.3 Sharing data via prototypes

Consider the following code:

```
const jane = {
  name: 'Jane',
  describe() {
    return 'Person named '+this.name;
  },
};
const tarzan = {
  name: 'Tarzan',
  describe() {
    return 'Person named '+this.name;
  },
};

assert.equal(jane.describe(), 'Person named Jane');
assert.equal(tarzan.describe(), 'Person named Tarzan');
```

We have two objects that are very similar. Both have two properties whose names are `.name` and `.describe`. Additionally, method `.describe()` is the same. How can we avoid that method being duplicated?

We can move it to a shared prototype, `PersonProto`:

```
const PersonProto = {
  describe() {
    return 'Person named ' + this.name;
  },
};
```

```
const jane = {
  __proto__: PersonProto,
  name: 'Jane',
};
const tarzan = {
  __proto__: PersonProto,
  name: 'Tarzan',
};
```

The name of the prototype reflects that both `jane` and `tarzan` are persons.

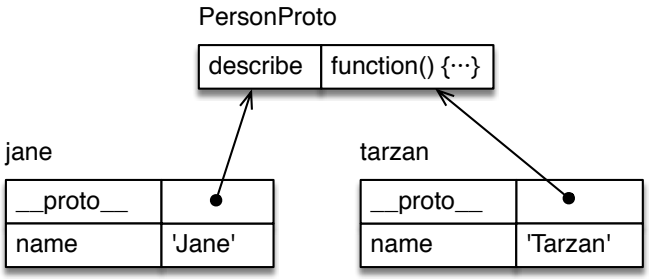

Figure 26.4: Objects `jane` and `tarzan` share method `.describe()`, via their common prototype `PersonProto`.

The diagram in fig. 26.4 illustrates how the three objects are connected: The objects at the bottom now contain the properties that are specific to `jane` and `tarzan`. The object at the top contains the properties that is shared between them.

When you make the method call `jane.describe()`, `this` points to the receiver of that method call, `jane` (in the bottom left corner of the diagram). That's why the method still works. The analogous thing happens when you call `tarzan.describe()`.

```
assert.equal(jane.describe(), 'Person named Jane');
assert.equal(tarzan.describe(), 'Person named Tarzan');
```

26.2 Classes

We are now ready to take on classes, which are basically a compact syntax for setting up prototype chains. While their foundations may be unconventional, working with JavaScript's classes should still feel familiar – if you have used an object-oriented language before.

26.2.1 A class for persons

We have previously worked with `jane` and `tarzan`, single objects representing persons. Let's use a class to implement a factory for persons:

```
class Person {
  constructor(name) {
    this.name = name;
  }
  describe() {
    return 'Person named '+this.name;
  }
}
```

`jane` and `tarzan` can now be created via `new Person()`:

```
const jane = new Person('Jane');
assert.equal(jane.describe(), 'Person named Jane');

const tarzan = new Person('Tarzan');
assert.equal(tarzan.describe(), 'Person named Tarzan');
```

26.2.2 Class expressions

The previous class definition was a *class declaration*. There are also *anonymous class expressions*:

```
const Person = class { ··· };
```

And *named class expressions*:

```
const Person = class MyClass { ··· };
```

26.2.3 Classes under the hood (advanced)

There is a lot going on under the hood of classes. Let's look at the diagram for `jane` (fig. 26.5).

The main purpose of class `Person` is to set up the prototype chain on the right (`jane`, followed by `Person.prototype`). It is interesting to note that both constructs inside class `Person` (`.constructor` and `.describe()`) created properties for `Person.prototype`, not for `Person`.

The reason for this slightly odd approach is backward compatibility: Prior to classes, *constructor functions* (ordinary functions, invoked via the `new` operator) were often used as factories for objects. Classes are mostly better syntax for constructor functions and therefore remain compatible with old code. That explains why classes are functions:

```
> typeof Person
'function'
```

In this book, I use the terms *constructor (function)* and *class* interchangeably.

Many people confuse `.__proto__` and `.prototype`. Hopefully, the diagram in fig. 26.5 makes it clear, how they differ:

- `.__proto__` is a special property for accessing the prototype of an object.

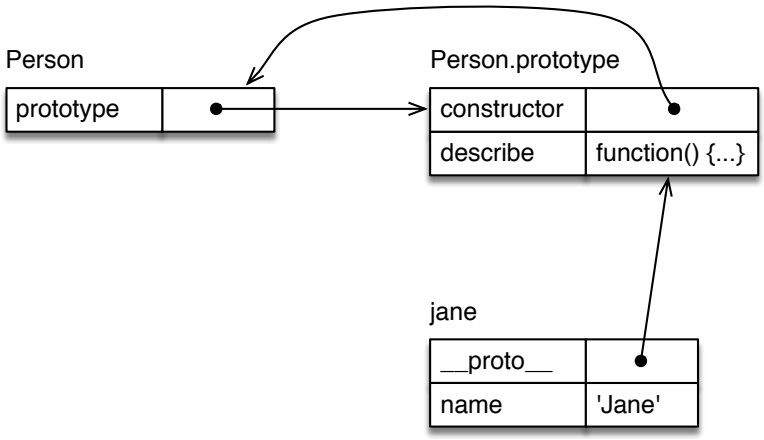

Figure 26.5: The class `Person` has the property `.prototype` that points to an object that is the prototype of all instances of `Person`. `jane` is one such instance.

- `.prototype` is a normal property that is only special due to how the `new` operator uses it. The name is not ideal: `Person.prototype` does not point to the prototype of `Person`, it points to the prototype of all instances of `Person`.

26.2.3.1 `Person.prototype.constructor`

There is one detail in fig. 26.5 that we haven't looked at, yet: `Person.prototype.constructor` points back to `Person`:

```
> Person.prototype.constructor === Person
true
```

This setup is also there for historical reasons. But it also has two benefits.

First, each instance of a class inherits property `.constructor`. Therefore, given an instance, you can make "similar" objects via it:

```
const jane = new Person('Jane');

const cheeta = new jane.constructor('Cheeta');
// 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');
```

26.2.4 Class definitions: prototype properties

The following code demonstrates all parts of a class definition `Foo` that create properties of `Foo.prototype`:

```
class Foo {
  constructor(prop) {
    this.prop = prop;
  }
  protoMethod() {
    return 'protoMethod';
  }
  get protoGetter() {
    return 'protoGetter';
  }
}
```

Let's examine them in order:

- `.constructor()` is called after creating a new instance of `Foo`, to set up that instance.
- `.protoMethod()` is a normal method. It is stored in `Foo.prototype`.
- `.protoGetter` is a getter that is stored in `Foo.prototype`.

The following interaction uses class `Foo`:

```
> const foo = new Foo(123);
> foo.prop
123

> foo.protoMethod()
'protoMethod'
> foo.protoGetter
'protoGetter'
```

26.2.5 Class definitions: static properties

The following code demonstrates all parts of a class definition that create so-called *static properties* – properties of the class itself.

```
class Bar {
  static staticMethod() {
    return 'staticMethod';
  }
  static get staticGetter() {
    return 'staticGetter';
  }
}
```

The static method and the static getter are used as follows.

```
> Bar.staticMethod()
'staticMethod'
> Bar.staticGetter
'staticGetter'
```

26.2.6 The `instanceof` operator

The `instanceof` operator tells you if a value is an instance of a given class:

```
> new Person('Jane') instanceof Person
true
> ({}) instanceof Person
false
> ({}) instanceof Object
true
> [] instanceof Array
true
```

We'll explore the `instanceof` operator in more detail later, after we have looked at subclassing.

26.2.7 Why I recommend classes

I recommend using classes for the following reasons:

- Classes are a common standard for object creation and inheritance that is now widely supported across frameworks (React, Angular, Ember, etc.).
- They help tools such as IDEs and type checkers with their work and enable new features.
- They are a foundation for future features such as value objects, immutable objects, decorators, etc.
- They make it easier for newcomers to get started with JavaScript.
- JavaScript engines optimize them. That is, code that uses classes is usually faster than code that uses a custom inheritance library.

That doesn't mean that classes are perfect. One issue I have with them, is:

- Classes look different from what they are under the hood. In other words, there is a disconnect between syntax and semantics.

It would be nice if classes were (syntax for) constructor *objects* (`new`-able prototype objects) and not to constructor *functions*. But backward compatibility is a legitimate reason for them being the latter.

Exercise: Implementing a class

`exercises/proto-chains-classes/point_class_test.js`

26.3 Private data for classes

This section describes techniques for hiding some of the data of an object from the outside. We discuss them in the context of classes, but they also work for objects created directly, via object literals etc.

26.3.1 Private data: naming convention

The first technique makes a property private by prefixing its name with an underscore. This doesn't protect the property in any way; it merely signals to the outside: "You don't need to know about this property."

In the following code, the properties `._counter` and `._action` are private.

```
class Countdown {
  constructor(counter, action) {
    this._counter = counter;
    this._action = action;
  }
  dec() {
    if (this._counter < 1) return;
    this._counter--;
    if (this._counter === 0) {
      this._action();
    }
  }
}

// The two properties aren't really private:
assert.deepEqual(
  Reflect.ownKeys(new Countdown()),
  ['_counter', '_action']);
```

With this technique, you don't get any protection and private names can clash. On the plus side, it is easy to use.

26.3.2 Private data: WeakMaps

Another technique is to use WeakMaps. How exactly that works is explained in the chapter on WeakMaps. This is a preview:

```
let _counter = new WeakMap();
let _action = new WeakMap();

class Countdown {
  constructor(counter, action) {
    _counter.set(this, counter);
    _action.set(this, action);
```

```
  }
  dec() {
    let counter = _counter.get(this);
    if (counter < 1) return;
    counter--;
    _counter.set(this, counter);
    if (counter === 0) {
      _action.get(this)();
    }
  }
}

// The two pseudo-properties are truly private:
assert.deepEqual(
  Reflect.ownKeys(new Countdown()),
  []);
```

This technique offers you considerable protection from outside access and there can't be any name clashes. But it is also more complicated to use.

26.3.3 More techniques for private data

There are more techniques for private data for classes. These are explained in "Exploring ES6".

The reason why this section does not go into much depth, is that JavaScript will probably soon have built-in support for private data. Consult the ECMAScript proposal "Class Public Instance Fields & Private Instance Fields" for details.

26.4 Subclassing

Classes can also subclass ("extend") existing classes. As an example, the following class `Employee` subclasses `Person`:

```
class Person {
  constructor(name) {
    this.name = name;
  }
  describe() {
    return `Person named ${this.name}`;
  }
  static logNames(persons) {
    for (const person of persons) {
      console.log(person.name);
    }
  }
}
```

```
class Employee extends Person {
  constructor(name, title) {
    super(name);
    this.title = title;
  }
  describe() {
    return super.describe() +
      ` (${this.title})`;
  }
}

const jane = new Employee('Jane', 'CTO');
assert.equal(
  jane.describe(),
  'Person named Jane (CTO)');
```

Two comments:

- Inside a `.constructor()` method, you must call the super-constructor via `super()`, before you can access `this`. That's because `this` doesn't exist before the super-constructor was called (this phenomenon is specific to classes).
- Static methods are also inherited. For example, `Employee` inherits the static method `.logNames()`:

  ```
  > 'logNames' in Employee
  true
  ```

Exercise: Subclassing

`exercises/proto-chains-classes/color_point_class_test.js`

26.4.1 Subclasses under the hood (advanced)

The classes `Person` and `Employee` from the previous section are made up of several objects (fig. 26.6). 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:

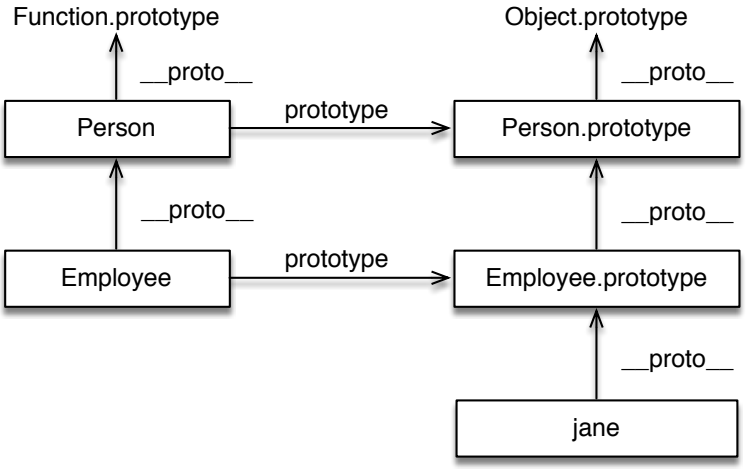

Figure 26.6: These are the objects that make up class `Person` and its subclass, `Employee`. The left column is about classes. The right column is about the `Employee` instance `jane` and its prototype chain.

```
> Object.getPrototypeOf(Person.prototype) === Object.prototype
true
```

26.4.1.2 The class prototype chain (left column)

In the class prototype chain, `Employee` comes first, `Person` next. Afterwards, the chain continues with `Function.prototype`, which is only there, because `Person` is a function and functions need the services of `Function.prototype`.

```
> Object.getPrototypeOf(Person) === Function.prototype
true
```

26.4.2 `instanceof` in more detail (advanced)

We have not yet seen how `instanceof` really works. Given the expression `x instanceof C`, how does `instanceof` determine if `x` is an instance of `C`? It does so by checking if `C.prototype` is in the prototype chain of `x`. That is, the following two expressions are equivalent:

```
x instanceof C
C.prototype.isPrototypeOf(x)
```

If we go back to fig. 26.6, we can confirm that the prototype chain does lead us to the following answers:

```
> jane instanceof Employee
true
> jane instanceof Person
```

```
true
> jane instanceof Object
true
```

26.4.3 Prototype chains of built-in objects (advanced)

Next, we'll use our knowledge of subclassing to understand the prototype chains of a few built-in objects. The following tool function `p()` helps us with our explorations.

```
const p = Object.getPrototypeOf.bind(Object);
```

We extracted method `.getPrototypeOf()` of `Object` and assigned it to `p`.

26.4.3.1 The prototype chain of {}

Let's start by examining plain objects:

```
> p({}) === Object.prototype
true
> p(p({})) === null
true
```

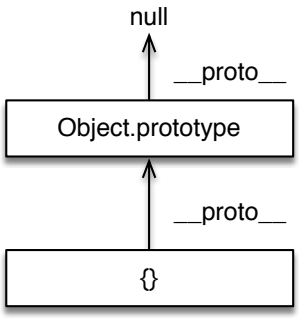

Figure 26.7: The prototype chain of an object created via an object literal starts with that object, continues with `Object.prototype` and ends with `null`.

Fig. 26.7 shows a diagram for this prototype chain. We can see that `{}` really is an instance of `Object` – `Object.prototype` is in its prototype chain.

`Object.prototype` is a curious value: It is an object, but it is not an instance of `Object`:

```
> typeof Object.prototype
'object'
> Object.prototype instanceof Object
false
```

That can't be avoided, because `Object.prototype` can't be in its own prototype chain.

26.4.3.2 The prototype chain of `[]`

What does the prototype chain of an Array look like?

```
> p([]) === Array.prototype
true
> p(p([])) === Object.prototype
true
> p(p(p([]))) === null
true
```

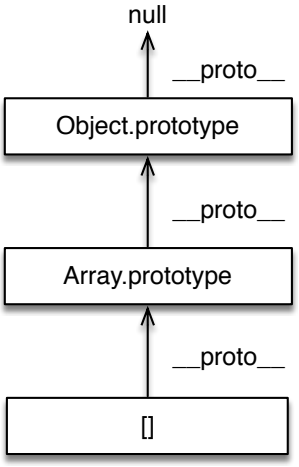

Figure 26.8: The prototype chain of an Array has these members: the Array instance, `Array.prototype`, `Object.prototype`, `null`.

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
```

26.4.4 Dispatched vs. direct method calls (advanced)

Let's examine how method calls work with classes. We revisit `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.

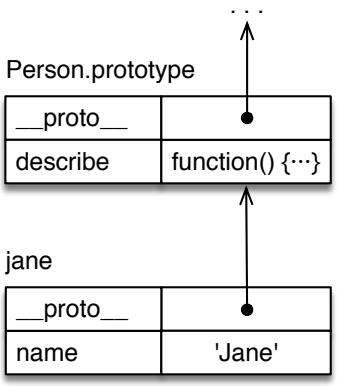

Figure 26.9: The prototype chain of `jane` starts with `jane` and continues with `Person.prototype`.

Normal method calls are *dispatched*. To make the method call `jane.describe()`:

- JavaScript first looks for the value of `jane.describe`, by traversing the prototype chain.
- Then it calls the function it found, while setting `this` to `jane`. `this` is the *receiver* of the method call (where the search for property `.describe` started).

This way of dynamically looking for methods, is called *dynamic dispatch*.

You can make the same method call while bypassing dispatch:

```
Person.prototype.describe.call(jane)
```

This time, `Person.prototype.describe` is an own property and there is no need to search the prototypes. We also specify `this` ourselves, via `.call()`.

Note how `this` always points to the beginning of a prototype chain. That enables `.describe()` to access `.name`. And it is where the mutations happen (should a method want to set the `.name`).

26.4.4.1 Borrowing methods

Direct method calls become useful when you are working with methods of `Object.prototype`. For example, `Object.prototype.hasOwnProperty()` checks if an object has a non-inherited property whose key is as given:

```
> const obj = { foo: 123 };
> obj.hasOwnProperty('foo')
true
> obj.hasOwnProperty('bar')
false
```

However, this method may be overridden. Then a dispatched method call doesn't work:

```
> const obj = { hasOwnProperty: true };
> obj.hasOwnProperty('bar')
TypeError: obj.hasOwnProperty is not a function
```

The work-around 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
```

JavaScript engines optimize this pattern, so that 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. A way around this limitation is via a technique called *mixin classes* (short: *mixins*).

The idea is as follows: Let's assume there is a class `C` that extends a class `S` – its superclass. Mixins are class fragments that are inserted between `C` and `S`.

In JavaScript, you can implement a mixin `Mix` via a function whose input is a class and whose output is the mixin class fragment – a new class that extends the input. To use `Mix()`, you create `C` as follows.

```
class C extends Mix(S) {
  ···
}
```

Let's look at an example:

```
const Branded = S => class extends S {
  setBrand(brand) {
    this._brand = brand;
    return this;
  }
  getBrand() {
```

```
      return this._brand;
    }
  };
```

We use this mixin to insert a class between `Car` and `Object`:

```
class Car extends Branded(Object) {
  constructor(model) {
    super();
    this._model = model;
  }
  toString() {
    return `${this.getBrand()} ${this._model}`;
  }
}
```

The following code confirms that the mixin worked: `Car` has method `.setBrand()` of `Branded`.

```
const modelT = new Car('Model T').setBrand('Ford');
assert.equal(modelT.toString(), 'Ford Model T');
```

Mixins are more flexible than normal classes:

- First, you can use the same mixin multiple times, in multiple classes.
- Second, you can use multiple mixins at the same time. As an example, consider an additional mixin called `Stringifiable` that helps with implementing `.toString()`. We could use both `Branded` and `Stringifiable` as follows:

  ```
  class Car extends Stringifiable(Branded(Object)) {
    ···
  }
  ```

Quiz

See quiz app.

Chapter 27

Where are the remaining chapters?

You are reading a preview of the book: For now, you have to buy it to get access to the complete contents (you can take a look at the full table of contents, which is also linked to from the book's homepage).

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