Exploring JavaScript (ES2024 Edition)

Dr. Axel Rauschmayer

2024-07-06

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

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

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

Table of contents

7 FAQ: JavaScript

II First steps

Part I

Background

Chapter 1

Before you buy the book

1.1 About the content

1.1.1 What's in this book?

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

Highlights:

- Get started quickly by initially focusing on modern features.
- Test-driven exercises available for most chapters.
- Covers all essential features of JavaScript, up to and including ES2022.
- Optional advanced sections let you dig deeper.

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

1.1.2 What is not covered by this book?

- Some advanced language features are not explained, but references to appropriate material are provided – for example, to my other JavaScript books at `Explor-`

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

1.2   Previewing and buying this book

1.2.1   How can I preview the book and its bundled material?

Go to the homepage of this book:

- All chapters of this book are free to read online.
- Most material has free preview versions (with about 50% of their content) that are available at the homepage.

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

The homepage of *Exploring JavaScript* explains how you can buy one of its digital packages.

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

An older edition of *Exploring JavaScript* is called *JavaScript for impatient programmers*. Its paper version is available on Amazon.

1.3   About the author

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

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

He lives in Munich, Germany.

1.4   Acknowledgements

- Cover image by Fran Caye
- Thanks for answering questions, discussing language topics, etc.:
  - Allen Wirfs-Brock
  - Benedikt Meurer
  - Brian Terlson
  - Daniel Ehrenberg
  - Jordan Harband
  - Maggie Johnson-Pint
  - Mathias Bynens
  - Myles Borins

- – Rob Palmer
  – Šime Vidas
  – And many others
- Thanks for reviewing:
  – Johannes Weber

Chapter 2

FAQ: book and supplementary material

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

2.1   How to read this book

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

This book is three books in one:

- You can use it to get started with JavaScript as quickly as possible:
  - Start reading with "Using JavaScript: the big picture" (§8).
  - Skip all chapters and sections marked as "advanced", and all quick references.

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

Exercises play an important part in helping you practice and retain what you have learned.

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

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

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

2.2   I own a digital version

2.2.1   How do I submit feedback and corrections?

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

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

- The receipt email for the purchase includes a link. You'll always be able to download the latest version of the files at that location.

- If you opted into emails while buying, you'll get an email whenever there is new content. To opt in later, you must contact Payhip (see bottom of `payhip.com`).

2.2.3   How do I upgrade from a smaller package to a larger one or an older package to a newer one?

The book's homepage explains how to do that.

2.3   I own the print version ("JavaScript for impatient programmers")

2.3.1   Can I get a discount for a digital version?

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

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

2.3.2 How do I submit feedback and corrections?

- Before reporting an error, please go to the online version of "Exploring JavaScript" and check the latest release of this book. The error may already have been corrected online.
- If the error is still there, you can use the comment link at the end of each chapter to report it.
- You can also use the comments to give feedback.

2.4 Notations and conventions

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

For example, you may see:

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

That is called the *type signature* of Number.isFinite(). This notation, especially the static types number of num and boolean of the result, are not real JavaScript. The notation is borrowed from the compile-to-JavaScript language TypeScript (which is mostly just JavaScript plus static typing).

Why is this notation being used? It helps give you a quick idea of how a function works. The notation is explained in detail in "Tackling TypeScript", but is usually relatively intuitive.

2.4.2 What do the notes with icons mean?

👁 Reading instructions

Explains how to best read the content.

↗ External content

Points to additional, external, content.

💡 Tip

Gives a tip related to the current content.

❓ Question

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

⚠️ **Warning**

Warns about pitfalls, etc.

⚙️ **Details**

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

🧩 **Exercise**

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

Chapter 3

Why JavaScript?

In this chapter, we examine the pros and cons of JavaScript.

> 👁 **"ECMAScript 6" and "ES6" refer to versions of JavaScript**
>
> ECMAScript is the name of the language standard; the number refers to the version of that standard. For more information, consult "Standardization: JavaScript vs. ECMAScript" (§5.2).

3.1 The cons of JavaScript

Among programmers, JavaScript isn't always well liked. One reason is that it has a fair amount of quirks. Some of them are just unusual ways of doing something. Others are considered bugs. Either way, learning *why* JavaScript does something the way it does, helps with dealing with the quirks and with accepting JavaScript (maybe even liking it). Hopefully, this book can be of assistance here.

Additionally, many traditional quirks have been eliminated now. For example:

- Traditionally, JavaScript variables weren't block-scoped. ES6 introduced `let` and `const`, which let you declare block-scoped variables.

- Prior to ES6, implementing object factories and inheritance via `function` and `.pro-totype` was clumsy. ES6 introduced classes, which provide more convenient syntax for these mechanisms.
- Traditionally, JavaScript did not have built-in modules. ES6 added them to the language.

Lastly, JavaScript's standard library is limited, but:

- There are plans for adding more functionality.
- Many libraries are easily available via the npm software registry.

3.2   The pros of JavaScript

On the plus side, JavaScript offers many benefits.

3.2.1   Community

JavaScript's popularity means that it's well supported and well documented. Whenever you create something in JavaScript, you can rely on many people being (potentially) interested. And there is a large pool of JavaScript programmers from which you can hire, if you need to.

No single party controls JavaScript – it is evolved by TC39, a committee comprising many organizations. The language is evolved via an open process that encourages feedback from the public.

3.2.2   Practically useful

With JavaScript, you can write apps for many client platforms. These are a few example technologies:

- *Progressive Web Apps* can be installed natively on Android and many desktop operating systems.
- *Electron* lets you build cross-platform desktop apps.
- *React Native* lets you write apps for iOS and Android that have native user interfaces.
- *Node.js* provides extensive support for writing shell scripts (in addition to being a platform for web servers).

JavaScript is supported by many server platforms and services – for example:

- Node.js (many of the following services are based on Node.js or support its APIs)
- ZEIT Now
- Microsoft Azure Functions
- AWS Lambda
- Google Cloud Functions

There are many data technologies available for JavaScript: many databases support it and intermediate layers (such as GraphQL) exist. Additionally, the standard data format *JSON (JavaScript Object Notation)* is based on JavaScript and supported by its standard library.

Lastly, many, if not most, tools for JavaScript are written in JavaScript. That includes IDEs, build tools, and more. As a consequence, you install them the same way you install your libraries and you can customize them in JavaScript.

3.2.3 Language

- Many libraries are available, via the de-facto standard in the JavaScript universe, the npm software registry.
- If you are unhappy with "plain" JavaScript, it is relatively easy to add more features:
    - You can compile future and modern language features to current and past versions of JavaScript, via Babel.
    - You can add static typing, via TypeScript and Flow.
    - You can work with ReasonML, which is, roughly, OCaml with JavaScript syntax. It can be compiled to JavaScript or native code.
- The language is flexible: it is dynamic and supports both object-oriented programming and functional programming.
- JavaScript has become suprisingly fast for such a dynamic language.
    - Whenever it isn't fast enough, you can switch to WebAssembly, a universal virtual machine built into most JavaScript engines. It can run static code at nearly native speeds.

3.3 Pro and con of JavaScript: innovation

There is much innovation in the JavaScript ecosystem: new approaches to implementing user interfaces, new ways of optimizing the delivery of software, and more. The upside is that you will constantly learn new things. The downside is that the constant change can be exhausting at times. Thankfully, things have somewhat slowed down, recently: all of ES6 (which was a considerable modernization of the language) is becoming established, as are certain tools and workflows.

Chapter 4

The nature of JavaScript

4.1 JavaScript's influences

When JavaScript was created in 1995, it was influenced by several programming languages:

- JavaScript's syntax is largely based on Java.
- Self inspired JavaScript's prototypal inheritance.
- Closures and environments were borrowed from Scheme.
- AWK influenced JavaScript's functions (including the keyword `function`).
- JavaScript's strings, Arrays, and regular expressions take cues from Perl.
- HyperTalk inspired event handling via `onclick` in web browsers.

With ECMAScript 6, new influences came to JavaScript:

- Generators were borrowed from Python.
- The syntax of arrow functions came from CoffeeScript.
- C++ contributed the keyword `const`.
- Destructuring was inspired by Lisp's *destructuring bind*.
- Template literals came from the E language (where they are called *quasi literals*).

4.2 The nature of JavaScript

These are a few traits of the language:

- Its syntax is part of the C family of languages (curly braces, etc.).

- It is a dynamic language: most objects can be changed in various ways at runtime, objects can be created directly, etc.

- It is a dynamically typed language: variables don't have fixed static types and you can assign any value to a given (mutable) variable.

- It has functional programming features: first-class functions, closures, partial application via `bind()`, etc.

- It has object-oriented features: mutable state, objects, inheritance, classes, etc.

- It often fails silently: see the next subsection for details.

- It is deployed as source code. But that source code is often *minified* (rewritten to require less storage). And there are plans for a binary source code format.

- JavaScript is part of the web platform – it is the language built into web browsers. But it is also used elsewhere – for example, in Node.js, for server things, and shell scripting.

- JavaScript engines often optimize less-efficient language mechanisms under the hood. For example, in principle, JavaScript Arrays are dictionaries. But under the hood, engines store Arrays contiguously if they have contiguous indices.

4.2.1   JavaScript often fails silently

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

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

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

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

```
> 1 / 0
Infinity
```

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

4.3   Tips for getting started with JavaScript

These are a few tips to help you get started with JavaScript:

- Take your time to really get to know this language. The conventional C-style syntax hides that this is a very unconventional language. Learn especially the quirks and the rationales behind them. Then you will understand and appreciate the language better.

  - In addition to details, this book also teaches simple rules of thumb to be safe – for example, "Always use === to determine if two values are equal, never ==."

- Language tools make it easier to work with JavaScript. For example:
    - You can statically type JavaScript via TypeScript.
    - You can check for problems and anti-patterns via linters such as ESLint.
    - You can format your code automatically via code formatters such as Prettier.
    - For more information on JavaScript tooling, see "Next steps: overview of web development".

- Get in contact with the community:
    - Social media services such as Mastodon are popular among JavaScript programmers. As a mode of communication that sits between the spoken and the written word, it is well suited for exchanging knowledge.
    - Many cities have regular free meetups where people come together to learn topics related to JavaScript.
    - JavaScript conferences are another convenient way of meeting other JavaScript programmers.

- Read books and blogs. Much material is free online!

Chapter 5

History and evolution of JavaScript

5.1   How JavaScript was created

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

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

Initially, JavaScript's name changed several times:

- Its code name was *Mocha*.

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

5.2   Standardization: JavaScript vs. ECMAScript

There are two standards for JavaScript:

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

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

The original name of that organization was *ECMA*, an acronym for *European Computer Manufacturers Association*. It was later changed to *Ecma International* (with "Ecma" being a proper name, not an acronym) because the organization's activities had expanded beyond Europe. The initial all-caps acronym explains the spelling of ECMAScript.

Often, 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).

5.3   Timeline of ECMAScript versions

This is a brief timeline of ECMAScript versions:

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

- ECMAScript 2016 (June 2016): First yearly release. The shorter release life cycle resulted in fewer new features compared to the large ES6.
- ECMAScript 2017 (June 2017). Second yearly release.
- Subsequent ECMAScript versions (ES2018, etc.) are always ratified in June.

5.4   Evolving JavaScript: TC39 (Ecma Technical Committee 39)

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

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

Outside of meetings, TC39 also collaborates with various members and groups of the JavaScript community.

5.5   The TC39 process for proposed ECMAScript features

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

- If too much time passes between releases then features that are ready early, have to wait a long time until they can be released. And features that are ready late, risk being rushed to make the deadline.

- Features were often designed long before they were implemented and used. Design deficiencies related to implementation and use were therefore discovered too late.

In response to these issues, TC39 instituted the new *TC39 process*:

- ECMAScript features are designed independently and go through six stages: a straw-person stage 0 and five "maturity" stages (1, 2, 2.7, 3, 4).
- 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.

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

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

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

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

5.5.2   The details of the TC39 process (advanced)

ECMAScript features are designed via *proposals* that go through the so-called *TC39 process*. That process comprises six stages:

- Stage 0 means a proposal has yet to enter the actual process. This is where most proposals start.

- Then the proposal goes through the five maturity stages 1, 2, 2.7, 3 and 4. If it reaches stage 4, it is complete and ready for inclusion in the ECMAScript standard.

Artifacts associated with an ECMAScript proposal

The following artifacts are associated with an ECMAScript proposal:

- Proposal document: Describes the proposal to JavaScript programmers, with English prose and code examples. Usually the readme of a GitHub repository.

- Specification: Written in *Ecmarkup*, an HTML and Markdown dialect that is supported by a toolchain. That toolchain checks Ecmarkup and renders it to HTML with features tailored to reading specifications (cross-references, highlighting of variable occurrences, etc.).

  - The HTML can also be printed to a PDF.
  - If a proposal makes it to stage 4, its specification is integrated into the full ECMAScript specification (which is also written in Ecmarkup).

- Tests: Written in JavaScript that check if an implementation conforms to the specification.

  - If a proposal makes it to stage 4, its tests are integrated into Test262, the official ECMAScript conformance test suite.

- Implementations: The functionality of the proposal, implemented in engines and transpilers (such as Babel and TypeScript).

Each stage has entrance criteria regarding the state of the artifacts:

Stage	Proposal	Specification	Tests	Implementations
0				
1	draft			
2	finished	draft		
2.7		finished		
3			finished	prototypes
4				2 implementations

Roles of the people that manage a proposal

- Author: A proposal is written by one or more authors.
- Champion: Each proposal has one or more TC39 delegates that guide the proposal through the TC39 process. This is especially important if an author has no experience with the process.

- Reviewer: Reviewers give feedback for the specification during stage 2 and must sign off on it before the proposal can reach stage 2.7. They are appointed by TC39 (excluding the authors and champions of the proposal).
- Editor: Someone in charge of managing the ECMAScript specification. The current editors are listed at the beginning of the ECMAScript specification.

The stages of a proposal

- Stage 0: ideation and exploration

  - Not part of the usual advancement process. Any author can create a draft proposal and assign it stage 0.

- Stage 1: designing a solution

  - Entrance criteria:
    * Pick champions
    * Repository with proposal
  - Status:
    * Proposal is under consideration.

- Stage 2: refining the solution

  - Entrance criteria:
    * Proposal is complete.
    * Draft of specification.
  - Status:
    * Proposal is likely (but not guaranteed) to be standardized.

- Stage 2.7: testing and validation

  - Entrance criteria:
    * Specification is complete and approved by reviewers and editors.
  - Status:
    * The specification is finished. It's time to validate it through tests and spec-compliant prototypes.
    * No more changes, aside from issues discovered through validation.

- Stage 3: gaining implementation experience

  - Entrance criteria:
    * Tests are finished.
  - Status:
    * The proposal is ready to be implemented.
    * No changes except if web incompatibilities are discovered.

- Stage 4: integration into draft specification and eventual inclusion in standard

  - Entrance criteria:
    * Two implementation that pass the tests
    * Significant in-the-field experience with shipping implementations
    * Pull request for TC39 repository, approved by editors
  - Status:
    * Proposed feature is complete:

  · Its specification is ready to be included in the ECMAScript specifica-
    tion.
  · Its tests are ready to be included in the ECMAScript conformance test
    suite Test262.

Figure 5.1 illustrates the TC39 process. Sources of this section:

```
┌─────────────────────────────────────────┐
│ Stage 0: ideation and exploration        │
└─────────────────────────────────────────┘
         │
         pick champions, draft proposal
         ↓
┌─────────────────────────────────────────┐
│ Stage 1: designing a solution            │
└─────────────────────────────────────────┘
         │
         finished proposal, draft specification
         ↓
┌─────────────────────────────────────────┐
│ Stage 2: refining the solution           │
└─────────────────────────────────────────┘
         │
         finished specification
         ↓
┌─────────────────────────────────────────┐
│ Stage 2.7: testing and validation        │
└─────────────────────────────────────────┘
         │
         finished tests, prototype implementations
         ↓
┌─────────────────────────────────────────┐
│ Stage 3: gaining implementation experience│
└─────────────────────────────────────────┘
         │
         two implementations
         ↓
┌─────────────────────────────────────────┐
│ Stage 4: integration into draft specification│
└─────────────────────────────────────────┘
```

Figure 5.1: Each ECMAScript feature proposal goes through stages that are numbered from
0 to 4.

  • "The TC39 Process" (official document by TC39)
  • The TC39 GitHub repository `how-we-work`, especially the document that explains the
    work of a proposal champion.
  • The *colophon* of the ECMAScript specification. A colophon is content at the end of a
    book. It usually contains information about the book's production.

5.6   How to not break the web while changing JavaScript

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.
- We can either migrate all of an existing code base to the new version (which can be a lot of work). Or we can mix versions and refactoring becomes harder because we can't move code between versions without changing it.
- We somehow have to specify per piece of code – be it a file or code embedded in a web page – what version it is written in. Every conceivable solution has pros and cons. For example, *strict mode* is a slightly cleaner version of ES5. One of the reasons why it wasn't as popular as it should have been: it was a hassle to opt in via a directive at the beginning of a file or a function.

So what is the solution? This is how JavaScript is evolved:

- New versions are always completely backward compatible (but there may occasionally be minor, hardly noticeable clean-ups).

- Old features aren't removed or fixed. Instead, better versions of them are introduced. One example is declaring variables via `let` – which is an improved version of `var`.

- If aspects of the language are changed, it is done inside new syntactic constructs. That is, we opt in implicitly – for example:

  - `yield` is only a keyword inside generators (which were introduced in ES6).
  - All code inside modules and classes (both introduced in ES6) is implicitly in strict mode.

5.7 FAQ: ECMAScript and TC39

5.7.1 Where can I look up which features were added in a given ECMAScript version?

There are several places where you can look up what's new in each ECMAScript version:

- In this book, there is a chapter that lists what's new in each ECMAScript version. It also links to explanations.

- The TC39 repository has a table with finished proposals that states in which ECMAScript versions they were (or will be) introduced.

- Section "Introduction" of the ECMAScript language specification lists the new features of each ECMAScript version.

- The ECMA-262 repository has a page with releases.

5.7.2 How is [my favorite proposed JavaScript feature] doing?

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

5.7.3   Why does stage 2.7 have such a peculiar number?

Stage 2.7 was added in late 2023, after stages 0, 1, 2, 3, 4 had already been in use for years.

- Q: Why not renumber the stages?
    - A: Renumbering was not in the cards because it would have made old documents difficult to read.
- Q: Why not another number such as 2.5?
    - The .7 reflects that stage 2.7 is closer to stage 3 than to stage 2.
- Q: How about 3a for the new stage and 3b for the old stage 3?
    - A: If you read "stage 3" in an old document, it can be confusing as to whether this refers to the new stage 3a or the new stage 3b.

Source: TC39 discussion on 2023-11-30

Chapter 6

New JavaScript features

This chapter lists what's new in recent ECMAScript versions – in reverse chronological order. It ends before ES6 (ES2015): ES2016 was the first truly incremental release of ECMAScript – which is why ES6 has too many features to list here. If you want to get a feeling for earlier releases:

- My book "Exploring ES6" describes what was added in ES6 (ES2015).
- My book "Speaking JavaScript" describes all of the features of ES5 – and is therefore a useful time capsule.

6.1 New in ECMAScript 2024

- Grouping synchronous iterables:

  Map.groupBy() groups the items of an iterable into Map entries whose keys are provided by a callback:

  ```
  assert.deepEqual(
    Map.groupBy([0, -5, 3, -4, 8, 9], x => Math.sign(x)),
    new Map()
  ```

```
      .set(0, [0])
      .set(-1, [-5,-4])
      .set(1, [3,8,9])
  );
```

There is also `Object.groupBy()` which produces an object instead of a Map:

```
assert.deepEqual(
  Object.groupBy([0, -5, 3, -4, 8, 9], x => Math.sign(x)),
  {
    '0': [0],
    '-1': [-5,-4],
    '1': [3,8,9],
    __proto__: null,
  }
);
```

- `Promise.withResolvers()` provides a new way of creating Promises that we want to resolve:

  ```
  const { promise, resolve, reject } = Promise.withResolvers();
  ```

- The new regular expression flag `/v` (`.unicodeSets`) enables these features:

  - Escapes for Unicode string properties (😵 consists of three code points):

    ```
    // Previously: Unicode code point property `Emoji` via /u
    assert.equal(
      /^\p{Emoji}$/u.test('😵'), false
    );
    // New: Unicode string property `RGI_Emoji` via /v
    assert.equal(
      /^\p{RGI_Emoji}$/v.test('😵'), true
    );
    ```

  - String literals via `\q{}` in character classes:

    ```
    > /^[\q{😵}]$/v.test('😵')
    true
    > /^[\q{abc|def}]$/v.test('abc')
    true
    ```

  - Set operations for character classes:

    ```
    > /^[\w--[a-g]]$/v.test('a')
    false
    > /^[\p{Number}--[0-9]]$/v.test('٣')
    true
    > /^[\p{RGI_Emoji}--\q{😵}]$/v.test('😵')
    false
    ```

  - Improved matching with `/i` if a Unicode property escape is negated via `[^···]`

- ArrayBuffers get two new features:

– They can be resized in place:

```
const buf = new ArrayBuffer(2, {maxByteLength: 4});
// `typedArray` starts at offset 2
const typedArray = new Uint8Array(buf, 2);
assert.equal(
  typedArray.length, 0
);
buf.resize(4);
assert.equal(
  typedArray.length, 2
);
```

– They get a method `.transfer()` for transferring them.

- SharedArrayBuffers can be resized, but they can only grow and never shrink. They are not transferrable and therefore don't get the method `.transfer()` that Array-Buffers got.

- Two new methods help us ensure that strings are well-formed (w.r.t. UTF-16 code units):

  – String method `.isWellFormed()` checks if a JavaScript string is *well-formed* and does not contain any *lone surrogates*.
  – String method `.toWellFormed()` returns a copy of the receiver where each lone surrogate is replaced with the code unit 0xFFFD (which represents the code point with the same number, whose name is "replacement character"). The result is therefore well-formed.

- `Atomics.waitAsync()` lets us wait asynchronously for a change to shared memory. Its functionality is beyond the scope of this book. See the MDN Web Docs for more information.

6.2 New in ECMAScript 2023

- "Change Array by copy": Arrays and Typed Arrays get new non-destructive methods that copy receivers before changing them:

  – `.toReversed()` is the non-destructive version of `.reverse()`:

  ```
  const original = ['a', 'b', 'c'];
  const reversed = original.toReversed();
  assert.deepEqual(reversed, ['c', 'b', 'a']);
  // The original is unchanged
  assert.deepEqual(original, ['a', 'b', 'c']);
  ```

  – `.toSorted()` is the non-destructive version of `.sort()`:

  ```
  const original = ['c', 'a', 'b'];
  const sorted = original.toSorted();
  assert.deepEqual(sorted, ['a', 'b', 'c']);
  ```

```
            // The original is unchanged
            assert.deepEqual(original, ['c', 'a', 'b']);
```

– `.toSpliced()` is the non-destructive version of `.splice()`:

```
        const original = ['a', 'b', 'c', 'd'];
        const spliced = original.toSpliced(1, 2, 'x');
        assert.deepEqual(spliced, ['a', 'x', 'd']);
        // The original is unchanged
        assert.deepEqual(original, ['a', 'b', 'c', 'd']);
```

– `.with()` is the non-destructive version of setting a value with square brackets:

```
        const original = ['a', 'b', 'c'];
        const updated = original.with(1, 'x');
        assert.deepEqual(updated, ['a', 'x', 'c']);
        // The original is unchanged
        assert.deepEqual(original, ['a', 'b', 'c']);
```

- "Array find from last": Arrays and Typed Arrays get two new methods:

  – `.findLast()` is similar to `.find()` but starts searching at the end of an Array:

  ```
  > ['', 'a', 'b', ''].findLast(s => s.length > 0)
  'b'
  ```

  – `.findLastIndex()` is similar to `.findIndex()` but starts searching at the end of an Array:

  ```
  > ['', 'a', 'b', ''].findLastIndex(s => s.length > 0)
  2
  ```

- Symbols as WeakMap keys: Before this feature, only objects could be used as keys in WeakMaps. This feature also lets us use symbols – except for *registered symbols* (created via `Symbol.for()`).

- "Hashbang grammar": JavaScript now ignores the first line of a file if it starts with a hash (`#`) and a bang (`!`). Some JavaScript runtimes, such as Node.js, have done this for a long time. Now it is also part of the language proper. This is an example of a "hashbang" line:

  ```
  #!/usr/bin/env node
  ```

6.3   New in ECMAScript 2022

- New members of classes:

  – Properties (public slots) can now be created via:
    * Instance public fields
    * Static public fields
  – Private slots are new and can be created via:
    * Private fields (instance private fields and static private fields)
    * Private methods and accessors (non-static and static)
  – Static initialization blocks

- Private slot checks ("ergonomic brand checks for private fields"): The following expression checks if obj has a private slot #privateSlot:

  ```
  #privateSlot in obj
  ```

- Top-level await in modules: We can now use await at the top levels of modules and don't have to enter async functions or methods anymore.

- error.cause: Error and its subclasses now let us specify which error caused the current one:

  ```
  new Error('Something went wrong', {cause: otherError})
  ```

- Method .at() of indexable values lets us read an element at a given index (like the bracket operator []) and supports negative indices (unlike the bracket operator).

  ```
  > ['a', 'b', 'c'].at(0)
  'a'
  > ['a', 'b', 'c'].at(-1)
  'c'
  ```

  The following "indexable" types have method .at():

  - string
  - Array
  - All Typed Array classes: Uint8Array etc.

- RegExp match indices: If we add a flag to a regular expression, using it produces match objects that record the start and end index of each group capture.

- Object.hasOwn(obj, propKey) provides a safe way to check if an object obj has an own property with the key propKey.

6.4 New in ECMAScript 2021

- String.prototype.replaceAll() lets us replace all matches of a regular expression or a string (.replace() only replaces the first occurrence of a string):

  ```
  > 'abbbaab'.replaceAll('b', 'x')
  'axxxaax'
  ```

- Promise.any() and AggregateError: Promise.any() returns a Promise that is fulfilled as soon as the first Promise in an iterable of Promises is fulfilled. If there are only rejections, they are put into an AggregateError which becomes the rejection value.

  We use Promise.any() when we are only interested in the first fulfilled Promise among several.

- Logical assignment operators:

  ```
  a ||= b
  a &&= b
  a ??= b
  ```

- Underscores (_) as separators in:

    - Number literals: `123_456.789_012`
    - Bigint literals: `6_000_000_000_000_000_000_000_000n`

- WeakRefs: This feature is beyond the scope of this book. Quoting its proposal states:

    - [This proposal] encompasses two major new pieces of functionality:
        * Creating weak references to objects with the `WeakRef` class
        * Running user-defined finalizers after objects are garbage-collected, with the `FinalizationRegistry` class
    - Their correct use takes careful thought, and they are best avoided if possible.

- `Array.prototype.sort` has been stable since ES2019. In ES2021, "[it] was made more precise, reducing the amount of cases that result in an implementation-defined sort order" [source]. For more information, see the pull request for this improvement.

6.5 New in ECMAScript 2020

- New module features:

    - Dynamic imports via `import()`: The normal `import` statement is static: We can only use it at the top levels of modules and its module specifier is a fixed string. `import()` changes that. It can be used anywhere (including conditional statements) and we can compute its argument.

    - `import.meta` contains metadata for the current module. Its first widely supported property is `import.meta.url` which contains a string with the URL of the current module's file.

    - Namespace re-exporting: The following expression imports all exports of module `'mod'` in a namespace object `ns` and exports that object.

        ```
        export * as ns from 'mod';
        ```

- Optional chaining for property accesses and method calls. One example of optional chaining is:

    ```
    value?.prop
    ```

    This expression evaluates to `undefined` if `value` is either `undefined` or `null`. Otherwise, it evaluates to `value.prop`. This feature is especially useful in chains of property reads when some of the properties may be missing.

- Nullish coalescing operator (`??`):

    ```
    value ?? defaultValue
    ```

    This expression is `defaultValue` if `value` is either `undefined` or `null` and `value` otherwise. This operator lets us use a default value whenever something is missing.

    Previously the Logical Or operator (`||`) was used in this case but it has downsides here because it returns the default value whenever the left-hand side is falsy (which isn't always correct).

- Bigints – arbitrary-precision integers: Bigints are a new primitive type. It supports integer numbers that can be arbitrarily large (storage for them grows as necessary).

- `String.prototype.matchAll()`: This method throws if flag `/g` isn't set and returns an iterable with all match objects for a given string.

- `Promise.allSettled()` receives an iterable of Promises. It returns a Promise that is fulfilled once all the input Promises are settled. The fulfillment value is an Array with one object per input Promise – either one of:

    – `{ status: 'fulfilled', value: «fulfillment value» }`
    – `{ status: 'rejected', reason: «rejection value» }`

- `globalThis` provides a way to access the global object that works both on browsers and server-side platforms such as Node.js and Deno.

- `for-in` mechanics: This feature is beyond the scope of this book. For more information on it, see its proposal.

- Namespace re-exporting:

    ```
    export * as ns from './internal.mjs';
    ```

6.6 New in ECMAScript 2019

- Array method `.flatMap()` works like `.map()` but lets the callback return Arrays of zero or more values instead of single values. The returned Arrays are then concatenated and become the result of `.flatMap()`. Use cases include:

    – Filtering and mapping at the same time
    – Mapping single input values to multiple output values

- Array method `.flat()` converts nested Arrays into flat Arrays. Optionally, we can tell it at which depth of nesting it should stop flattening.

- `Object.fromEntries()` creates an object from an iterable over *entries*. Each entry is a two-element Array with a property key and a property value.

- String methods: `.trimStart()` and `.trimEnd()` work like `.trim()` but remove whitespace only at the start or only at the end of a string.

- Optional `catch` binding: We can now omit the parameter of a `catch` clause if we don't use it.

- `Symbol.prototype.description` is a getter for reading the description of a symbol. Previously, the description was included in the result of `.toString()` but couldn't be accessed individually.

- `.sort()` for Arrays and Typed Arrays is now guaranteed to be *stable*: If elements are considered equal by sorting, then sorting does not change the order of those elements (relative to each other).

These ES2019 features are beyond the scope of this book:

- JSON superset: See 2ality blog post.

- Well-formed `JSON.stringify()`: See 2ality blog post.
- `Function.prototype.toString()` revision: See 2ality blog post.

6.7   New in ECMAScript 2018

- Asynchronous iteration is the asynchronous version of synchronous iteration. It is based on Promises:

  - With synchronous iterables, we can immediately access each item. With asynchronous iterables, we have to `await` before we can access an item.
  - With synchronous iterables, we use `for-of` loops. With asynchronous iterables, we use `for-await-of` loops.

- Spreading into object literals: By using spreading (`...`) inside an object literal, we can copy the properties of another object into the current one. One use case is to create a shallow copy of an object `obj`:

  ```
  const shallowCopy = {...obj};
  ```

- Rest properties (destructuring): When object-destructuring a value, we can now use rest syntax (`...`) to get all previously unmentioned properties in an object.

  ```
  const {a, ...remaining} = {a: 1, b: 2, c: 3};
  assert.deepEqual(remaining, {b: 2, c: 3});
  ```

- `Promise.prototype.finally()` is related to the `finally` clause of a try-catch-finally statement – similarly to how the Promise method `.then()` is related to the `try` clause and `.catch()` is related to the `catch` clause.

  On other words: The callback of `.finally()` is executed regardless of whether a Promise is fulfilled or rejected.

- New Regular expression features:

  - RegExp named capture groups: In addition to accessing groups by number, we can now name them and access them by name:

    ```
    const matchObj = '---756---'.match(/(?<digits>[0-9]+)/)
    assert.equal(matchObj.groups.digits, '756');
    ```

  - RegExp lookbehind assertions complement lookahead assertions:

    * Positive lookbehind: `(?<=X)` matches if the current location is preceded by `'X'`.
    * Negative lookbehind: `(?<!X)` matches if the current location is not preceded by `'(?<!X)'`.

  - `s` (dotAll) flag for regular expressions. If this flag is active, the dot matches line terminators (by default, it doesn't).

  - RegExp Unicode property escapes give us more power when matching sets of Unicode code points – for example:

    ```
    > /^\p{Lowercase_Letter}+$/u.test('aüп')
    true
    ```

```
> /^\p{White_Space}+$/u.test('\n \t')
true
> /^\p{Script=Greek}+$/u.test('ΩΔΨ')
true
```

- Template literal revision allows text with backslashes in tagged templates that is illegal in string literals – for example:

  ```
  windowsPath`C:\uuu\xxx\111`
  latex`\unicode`
  ```

6.8 New in ECMAScript 2017

- Async functions (`async/await`) let us use synchronous-looking syntax to write asynchronous code.

- `Object.values()` returns an Array with the values of all enumerable string-keyed properties of a given object.

- `Object.entries()` returns an Array with the key-value pairs of all enumerable string-keyed properties of a given object. Each pair is encoded as a two-element Array.

- String padding: The string methods `.padStart()` and `.padEnd()` insert padding text until the receivers are long enough:

  ```
  > '7'.padStart(3, '0')
  '007'
  > 'yes'.padEnd(6, '!')
  'yes!!!'
  ```

- Trailing commas in function parameter lists and calls: Trailing commas have been allowed in Arrays literals since ES3 and in Object literals since ES5. They are now also allowed in function calls and method calls.

- `Object.getOwnPropertyDescriptors()` lets us define properties via an object with property descriptors:

- The feature "Shared memory and atomics" is beyond the scope of this book. For more information on it, see:

  – The documentation on `SharedArrayBuffer` and `Atomics` on MDN Web Docs
  – The ECMAScript proposal "Shared memory and atomics"

6.9 New in ECMAScript 2016

- `Array.prototype.includes()` checks if an Array contains a given value.

- Exponentiation operator (`**`):

  ```
  > 4 ** 2
  16
  ```

6.10   Source of this chapter

ECMAScript feature lists were taken from the TC39 page on finished proposals.

Chapter 7

FAQ: JavaScript

7.1 What are good references for JavaScript?

Please see "JavaScript references" (§8.3).

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

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

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

47

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

Please see the following sources:

- "The TC39 process for proposed ECMAScript features" (§5.5)
- "FAQ: ECMAScript and TC39" (§5.7)

7.4 Why does JavaScript fail silently so often?

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

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

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

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

```
> 1 / 0
Infinity
```

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

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

This question is answered in "How to not break the web while changing JavaScript" (§5.6).

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

"Trying out JavaScript code" (§10.1) explains how to do that.

Part II

First steps

Chapter 8

Using JavaScript: the big picture

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

8.1   What are you learning in this book?

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

- Web browser
- Node.js

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

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

8.2   The structure of browsers and Node.js

The structures of the two JavaScript platforms *web browser* and *Node.js* are similar (figure 8.1):

JS standard library	Platform API
JavaScript engine	Platform core
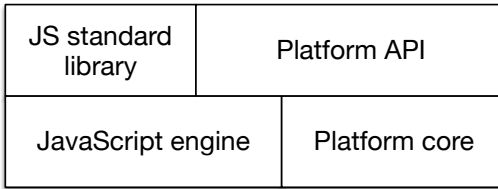
Figure 8.1: The structure of the two JavaScript platforms *web browser* and *Node.js*. The APIs "standard library" and "platform API" are hosted on top of a foundational layer with a JavaScript engine and a platform-specific "core".

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

8.3   JavaScript references

If you have a question about JavaScript, I can recommend the following online resources:

- MDN Web Docs: cover various web technologies such as CSS, HTML, JavaScript, and more. An excellent reference.
- Node.js Docs: document the Node.js API.
- ExploringJS.com: My other books cover various aspects of web development:
    - "Deep JavaScript: Theory and techniques" describes JavaScript at a level of detail that is beyond the scope of "Exploring JavaScript".
    - "Tackling TypeScript: Upgrading from JavaScript"
    - "Shell scripting with Node.js"

8.4   Further reading

- "Next steps: overview of web development" provides a more comprehensive look at web development.

Chapter 9

Syntax

9.1   An overview of JavaScript's syntax

This is a very first look at JavaScript's syntax. Don't worry if some things don't make sense, yet. They will all be explained in more detail later in this book.

This overview is not exhaustive, either. It focuses on the essentials.

9.1.1   Basic constructs

Comments

```
// single-line comment

/*
Comment with
multiple lines
*/
```

Primitive **(atomic) values**

Booleans:

```
true
false
```

Numbers:

```
1.141
-123
```

The basic number type is used for both floating point numbers (doubles) and integers.

Bigints:

```
17n
-49n
```

The basic number type can only properly represent integers within a range of 53 bits plus sign. Bigints can grow arbitrarily large in size.

Strings:

```
'abc'
"abc"
`String with interpolated values: ${256} and ${true}`
```

JavaScript has no extra type for characters. It uses strings to represent them.

Assertions

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

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

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

There is also assert.deepEqual() that compares objects deeply.

Logging to the console

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

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

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

Operators

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

// Operators for numbers
assert.equal(3 + 4, 7);
assert.equal(5 - 1, 4);
assert.equal(3 * 4, 12);
assert.equal(10 / 4, 2.5);

// Operators for bigints
assert.equal(3n + 4n, 7n);
assert.equal(5n - 1n, 4n);
assert.equal(3n * 4n, 12n);
assert.equal(10n / 4n, 2n);

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

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

JavaScript also has a == comparison operator. I recommend to avoid it – why is explained in "Recommendation: always use strict equality" (§15.4.3).

Declaring variables

const creates *immutable variable bindings*: Each variable must be initialized immediately and we can't assign a different value later. However, the value itself may be mutable and we may be able to change its contents. In other words: const does not make values immutable.

```
// Declaring and initializing x (immutable binding):
const x = 8;

// Would cause a TypeError:
// x = 9;
```

let creates *mutable variable bindings*:

```
// Declaring y (mutable binding):
let y;

// We can assign a different value to y:
y = 3 * 5;

// Declaring and initializing z:
let z = 3 * 5;
```

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

Arrow function expressions are used especially as arguments of function calls and method calls:

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

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

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

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

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

Plain objects

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

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

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

Arrays

```
// Creating an Array via an Array literal
const arr = ['a', 'b', 'c'];
assert.equal(arr.length, 3);

// Getting an Array element
assert.equal(arr[1], 'b');
// Setting an Array element
arr[1] = 'β';

// Adding an element to an Array:
arr.push('d');

assert.deepEqual(
  arr, ['a', 'β', 'c', 'd']);
```

Control flow statements

Conditional statement:

```
if (x < 0) {
  x = -x;
}
```

for-of loop:

```
const arr = ['a', 'b'];
for (const element of arr) {
  console.log(element);
}
```

Output:

```
a
b
```

9.1.2 Modules

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

```
file-tools.mjs
main.mjs
```

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

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

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

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

9.1.3 Classes

```
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);
```

```js
    this.title = title;
  }
  describe() {
    return super.describe() +
      ` (${this.title})`;
  }
}

const jane = new Employee('Jane', 'CTO');
assert.equal(
  jane.describe(),
  'Person named Jane (CTO)');
```

9.1.4 Exception handling

```js
function throwsException() {
  throw new Error('Problem!');
}

function catchesException() {
  try {
    throwsException();
  } catch (err) {
    assert.ok(err instanceof Error);
    assert.equal(err.message, 'Problem!');
  }
}
```

Note:

- `try-finally` and `try-catch-finally` are also supported.
- We can throw any value, but features such as stack traces are only supported by `Error` and its subclasses.

9.1.5 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:

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

But they are allowed as names of properties:

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

9.1.6 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`

9.1.7 Capitalization of names

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

Lowercase:

- Functions, variables: `myFunction`

- Methods: `obj.myMethod`

- CSS:

    - CSS names: `my-utility-class` (dash case)
    - Corresponding JavaScript names: `myUtilityClass`

- Module file names are usually dash-cased:

    ```
    import * as theSpecialLibrary from './the-special-library.mjs';
    ```

Uppercase:

- Classes: `MyClass`

All-caps:

- Constants (as shared between modules etc.): `MY_CONSTANT` (underscore case)

9.1.8 More naming conventions

The following naming conventions are popular in JavaScript.

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

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

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

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

```
    }
}
```

9.1.9   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() {
  // ···
};
```

9.2   (Advanced)

All remaining sections of this chapter are advanced.

9.3   Hashbang lines (Unix shell scripts)

In a Unix shell script, we can add a first line that starts with #! to tell Unix which executable should be used to run the script. These two characters have several names, including *hashbang*, *sharp-exclamation*, *sha-bang* ("sha" as in "sharp") and *shebang*. Otherwise, hashbang lines are treated as comments by most shell scripting languages and JavaScript does so, too. This is a common hashbang line for Node.js:

```
#!/usr/bin/env node
```

If we want to pass arguments to node, we have to use the env option -S (to be safe, some Unixes don't need it):

```
#!/usr/bin/env -S node --enable-source-maps --no-warnings=ExperimentalWarning
```

9.4   Identifiers

9.4.1   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 ɑ)
- $
- _

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;
```

9.4.2   Reserved words

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

All JavaScript *keywords* are reserved words:

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

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

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

The following literals are reserved words:

```
true false null
```

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

```
Infinity NaN undefined async
```

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

9.5   Statement vs. expression

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

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

9.5.1 Statements

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

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

One more example of a statement: a function declaration.

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

9.5.2 Expressions

An *expression* is a piece of code that can be *evaluated* to produce a value. For example, the code between the parentheses is an expression:

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

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

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

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

9.5.3 What is allowed where?

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

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

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

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

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

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

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

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

9.6   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.

9.6.1   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;
};
```

9.6.2   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):

```
{
}
```

9.6.3   Disambiguation

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

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

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

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

Output:

```
abc
```

In this code:

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

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

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

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

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

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

9.7   Semicolons

9.7.1   Rule of thumb for semicolons

Each statement is terminated by a semicolon:

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

except statements ending with blocks:

```
function foo() {
  // ···
```

```
}
if (y > 0) {
  // ···
}
```

The following case is slightly tricky:

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

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

9.7.2   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) {}
```

9.8   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.

9.8.1 ASI triggered unexpectedly

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

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

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

This code is parsed as:

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

That is:

- Return statement without operand: `return;`
- Start of code block: {
- Expression statement `'jane';` with label `first:`
- End of code block: }
- Empty statement: ;

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

9.8.2 ASI unexpectedly not triggered

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

Example 1: Unintended function call.

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

Parsed as:

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

Example 2: Unintended division.

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

Parsed as:

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

Example 3: Unintended property access.

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

Executed as:

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

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

9.9   Semicolons: best practices

I recommend that you always write semicolons:

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

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

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

9.10   Strict mode vs. sloppy mode

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

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

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

9.10.1   Switching on strict mode

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

```
'use strict';
```

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

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

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

9.10.2 Improvements in strict mode

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

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

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

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

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

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

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

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

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

```
function strictFunc() {
  'use strict';
  {
    function foo() { return 123 }
  }
  return foo(); // ReferenceError
}
assert.throws(
```

```
  () => strictFunc(),
  {
    name: 'ReferenceError',
    message: 'foo is not defined',
  });
```

In sloppy mode, function declarations are function-scoped:

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

Sloppy mode doesn't throw exceptions when changing immutable data

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

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

In sloppy mode, the assignment fails silently:

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

> 🔗 **Further reading: sloppy mode**
>
> For more information on how sloppy mode differs from strict mode, see MDN.

Chapter 10

Consoles: interactive JavaScript command lines

10.1 Trying out JavaScript code

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

10.1.1 Browser consoles

Web browsers have so-called *consoles*: interactive command lines to which you can print text via `console.log()` and where you can run pieces of code. How to open the console differs from browser to browser. Figure 10.1 shows the console of Google Chrome.

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

- Apple Safari
- Google Chrome
- Microsoft Edge
- Mozilla Firefox
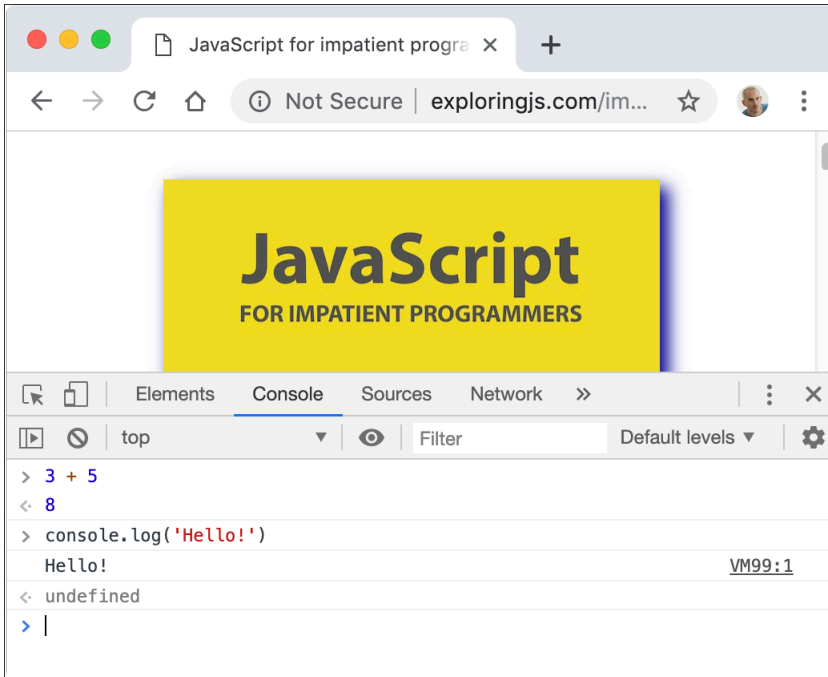
Figure 10.1: The console of the web browser "Google Chrome" is open (in the bottom half of window) while visiting a web page.

10.1.2   The Node.js REPL

REPL stands for *read-eval-print loop* and basically means *command line*. To use it, you must first start Node.js from an operating system command line, via the command `node`. Then an interaction with it looks as depicted in figure 10.2: The text after > is input from the user; everything else is output from Node.js.
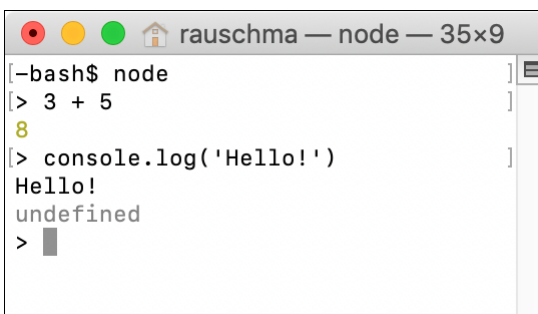
Figure 10.2: Starting and using the Node.js REPL (interactive command line).

> **Reading: REPL interactions**
>
> I occasionally demonstrate JavaScript via REPL interactions. Then I also use greater-than symbols (>) to mark input – for example:
> ```
> > 3 + 5
> 8
> ```

10.1.3 Other options

Other options include:

- There are many web apps that let you experiment with JavaScript in web browsers – for example, Babel's REPL.

- There are also native apps and IDE plugins for running JavaScript.

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

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

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

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

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

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

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

There are two variants of this operation:

```
console.log(...values: Array<any>): void
console.log(pattern: string, ...values: Array<any>): void
```

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.

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%
  ```

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

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

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

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

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

Output:

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

Chapter 11

Assertion API

11.1 Assertions in software development

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

```
import assert from 'node:assert/strict';
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`.

11.2 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.

11.2.1   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:

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

11.2.2   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, we must implement the function hello().
// The test checks if we 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, see "Getting started with exercises" (§12).

11.3   Normal comparison vs. deep comparison

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

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

deepEqual() is a better choice for comparing objects:

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

This method works for Arrays, too:

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

11.4   Quick reference: module **assert**

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

11.4.1   Normal equality: **assert.equal()**

- assert.equal(actual, expected, message?)

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

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

- assert.notEqual(actual, expected, message?)

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

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

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

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

11.4.2   Deep equality: **assert.deepEqual()**

- assert.deepEqual(actual, expected, message?)

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

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

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

- assert.notDeepEqual(actual, expected, message?)

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

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

11.4.3   Expecting exceptions: **assert.throws()**

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

- assert.throws(callback, message?): void

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

- assert.throws(callback, errorClass, message?): void

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

- assert.throws(callback, errorRegExp, message?): void

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

- assert.throws(callback, errorObject, message?): void

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

11.4.4 Always fail: `assert.fail()`

- assert.fail(messageOrError?)

  By default, it throws an AssertionError when it is called. That is occasionally useful
  for unit testing. messageOrError can be:

    – A string. That enables to override the default error message.
    – An instance of Error (or a subclass). That enables us to throw a different value.

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

```
    // Success
}
```

Chapter 12

Getting started with exercises

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

12.1 Exercises

12.1.1 Installing the exercises

To install the exercises:

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

12.1.2 Running exercises

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

12.2   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.

12.2.1   A typical test

Typical test code is split into two parts:

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

Take, for example, the following two files:

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

Part 1: the code

The code itself resides in id.mjs:

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

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

Part 2: the tests

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

The tests for the code reside in id_test.mjs:

```
// npm t demos/exercises/id_test.mjs
suite('id_test.mjs');

import assert from 'node:assert/strict'; // (A)
import {id} from './id.mjs'; // (B)

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

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

As for the other lines:

- The comment at the very beginning shows the shell command for running the test.
- Line A: We import the Node.js assertion library (in *strict assertion mode*).
- 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, which is provided via an arrow function. The parameter `t` gives us access to AVA's testing API (assertions, etc.).

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

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

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

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

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

12.2.2 Asynchronous tests in Mocha

> **👁 Reading**
>
> You may want to 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.

Asynchronicity via callbacks

If the callback we pass to `test()` has a parameter (e.g., `done`), Mocha switches to callback-based asynchronicity. When we are done with our asynchronous work, we have to call `done`:

```
test('divideCallback', (done) => {
  divideCallback(8, 4, (error, result) => {
    if (error) {
      done(error);
    } else {
      assert.strictEqual(result, 2);
      done();
    }
  });
});
```

This is what `divideCallback()` looks like:

```
function divideCallback(x, y, callback) {
  if (y === 0) {
    callback(new Error('Division by zero'));
  } else {
    callback(null, x / y);
  }
}
```

Asynchronicity via Promises

If a test returns a Promise, Mocha switches to Promise-based asynchronicity. A test is considered successful if the Promise is fulfilled and failed if the Promise is rejected or if a settlement takes longer than a timeout.

```
test('dividePromise 1', () => {
  return dividePromise(8, 4)
  .then(result => {
    assert.strictEqual(result, 2);
  });
});
```

`dividePromise()` is implemented as follows:

```
function dividePromise(x, y) {
  return new Promise((resolve, reject) => {
    if (y === 0) {
      reject(new Error('Division by zero'));
    } else {
      resolve(x / y);
    }
  });
}
```

Async functions as test "bodies"

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

```
test('dividePromise 2', async () => {
  const result = await dividePromise(8, 4);
  assert.strictEqual(result, 2);
  // No explicit return necessary!
});
```

We 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 13

Variables and assignment

These are JavaScript's main ways of declaring variables:

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

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

13.1 `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;
```

13.2 `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.',
  }
);
```

13.2.1 `const` and immutability

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

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

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

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

13.2.2 `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);
}
```

13.3 Deciding between `const` and `let`

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

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

Exercise: `const`

`exercises/variables-assignment/const_exrc.mjs`

13.4 The scope of a variable

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

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

```
      assert.equal(y, 1);
      assert.equal(z, 2);
    }
  }
}
// Outside. Not accessible: x, y, z
assert.throws(
  () => console.log(x),
  {
    name: 'ReferenceError',
    message: 'x is not defined',
  }
);
```

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

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

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

13.4.1   Shadowing variables

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

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

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

You can, however, nest a block and use the same variable name x that you used outside the block:

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

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

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

13.5 (Advanced)

All remaining sections are advanced.

13.6 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.

13.6.1 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).

13.6.2 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).

13.7 Global variables and the global object

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

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

- And so on.

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

- *Global declarative variables* are normal variables.

    - They can only be created while at the top level of a script, via `const`, `let`, and class declarations.

- *Global object variables* are stored in properties of the so-called *global object*.

    - They are created in the top level of a script, via `var` and function declarations.
    - The global object can be accessed via the global variable `globalThis`. It can be used to create, read, and delete global object variables.
    - Other than that, global object variables work like normal variables.

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

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

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

Each ECMAScript module has its own scope. Therefore, variables that exist at the top level of a module are not global. Figure 13.1 illustrates how the various scopes are related.

13.7.1 `globalThis` [ES2020]

The global variable `globalThis` is the new standard way of accessing the global object. It got its name from the fact that it has the same value as `this` in global scope (script scope, not module scope).

> ⚙️ **`globalThis` does not always directly point to the global object**
>
> For example, in browsers, there is an indirection. That indirection is normally not noticeable, but it is there and can be observed.
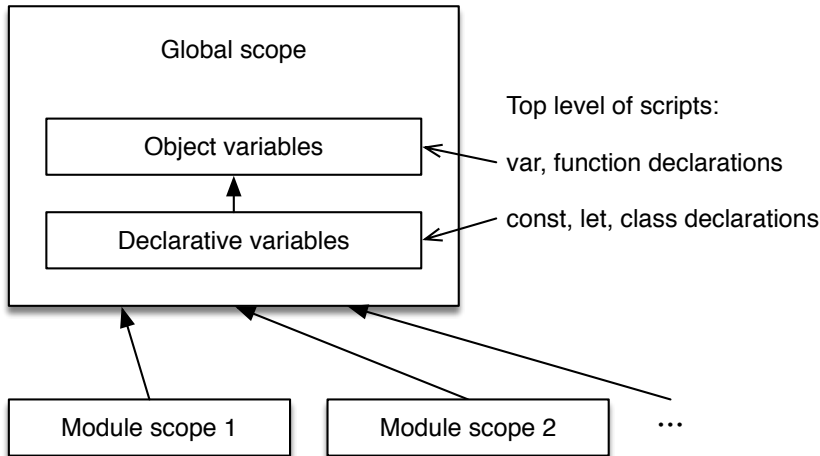
Figure 13.1: The global scope is JavaScript's outermost scope. It has two kinds of variables: *object variables* (managed via the *global object*) and normal *declarative variables*. Each ECMAScript module has its own scope which is contained in the global scope.

Alternatives to `globalThis`

The following global variables let us access the global object on *some* platforms:

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

	Main browser thread	**Web Workers**	**Node.js**
`globalThis`	✔	✔	✔
`window`	✔		
`self`	✔	✔	
`global`			✔

Use cases for `globalThis`

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

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

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

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

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

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

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

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

13.8   Declarations: scope and activation

These are two key aspects of declarations:

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

Table 13.1 summarizes how various declarations handle these aspects. `import` is described

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

Table 13.1: Aspects of declarations:
- "Duplicates" describes if a declaration can be used twice with the same name (per scope).
- "Global prop." describes if a declaration adds a property to the global object, when it is executed in the global scope of a script.
- *TDZ* means *temporal dead zone* (which is explained later).

(*) Function declarations are normally block-scoped, but function-scoped in sloppy mode.

in "ECMAScript modules" (§29.5). The following sections describe the other constructs in more detail.

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

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

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

Some possible approaches are:

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

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

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

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

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

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

The following code illustrates the temporal dead zone:

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

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

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

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

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

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

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

13.8.2   Function declarations and early activation

> 👁 **More information on functions**
>
> In this section, we are using functions – before we had a chance to learn them properly. Hopefully, everything still makes sense. Whenever it doesn't, please see "Callable values" (§27).

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

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

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

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

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

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

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

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

Calling ahead without early activation

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

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

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

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

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

A pitfall of early activation

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

```
funcDecl();

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

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

The pros and cons of early activation

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

13.8.3   Class declarations are not activated early

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

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

class MyClass {}

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

Why is that? Consider the following class declaration:

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

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

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

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

13.8.4   `var`: hoisting (partial early activation)

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

```
var x = 123;
```

This declaration has two parts:

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

The following code demonstrates the effects of `var`:

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

13.9 Closures

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

13.9.1 Bound variables vs. free variables

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

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

Consider the following code:

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

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

13.9.2 What is a closure?

What is a closure then?

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

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

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

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

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

> 💡 **All functions in JavaScript are closures**
>
> Static scoping is supported via closures in JavaScript. Therefore, every function is a closure.

13.9.3   Example: A factory for incrementors

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

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

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

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

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

```
assert.deepEqual(inc(2), [1, 9]);
assert.deepEqual(inc(2), [2, 11]);
```

13.9.4   Use cases for closures

What are closures good for?

- For starters, they are simply an implementation of static scoping. As such, they provide context data for callbacks.

- They can also be used by functions to store state that persists across function calls. `createInc()` is an example of that.

- And they can provide private data for objects (produced via literals or classes). The details of how that works are explained in *Exploring ES6*.

Chapter 14

Values

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

> 👁 **Supporting tool: `===`**
>
> In this chapter, we'll occasionally use the strict equality operator. `a === b` evaluates to `true` if `a` and `b` are equal. What exactly that means is explained in "Strict equality (=== and !==)" (§15.4.2).

14.1 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` }.

14.2 JavaScript's type hierarchy
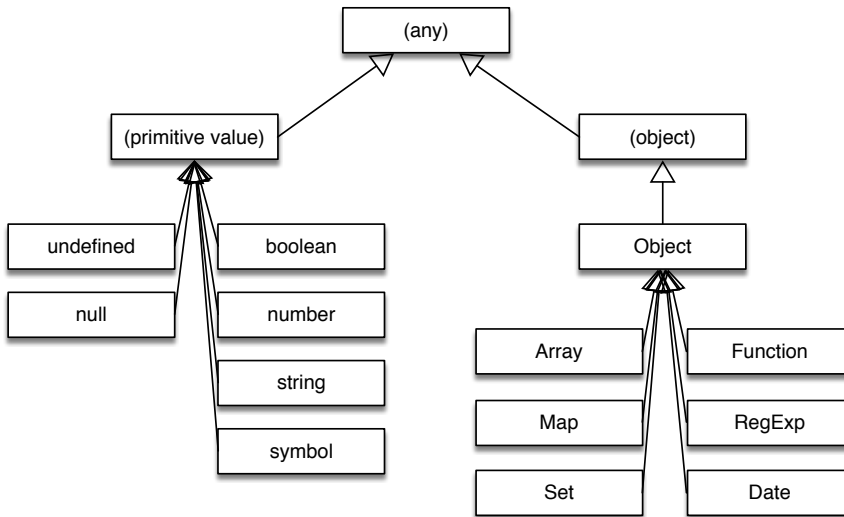
Figure 14.1: A partial hierarchy of JavaScript's types. Missing are the classes for errors, the classes associated with primitive types, and more. The diagram hints at the fact that not all objects are instances of `Object`.

Figure 14.1 shows JavaScript's type hierarchy. What do we learn from that diagram?

- JavaScript distinguishes two kinds of values: primitive values and objects. We'll see soon what the difference is.
- The diagram differentiates objects and instances of class `Object`. Each instance of `Object` is also an object, but not vice versa. However, virtually all objects that you'll encounter in practice are instances of `Object` – for example, objects created via object literals. More details on this topic are explained in "Not all objects are instances of `Object`" (§31.7.3).

14.3 The types of the language specification

The ECMAScript specification only knows a total of eight 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`)
- `bigint` the type of all big integers (e.g., `-123n`)
- `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)

14.4   Primitive values vs. objects

The specification makes an important distinction between values:

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

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

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

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

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

14.4.1   Primitive values (short: primitives)

Primitives are immutable

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

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

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.

```
const x = 123;
const y = x;
// `y` is the same as any other number 123
assert.equal(y, 123);
```

> ⚙️ **Observing the difference between passing by value and passing by reference**
>
> Due to primitive values being immutable and compared by value (see next subsection), there is no way to observe the difference between passing by value and passing by identity (as used for objects in JavaScript).

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.

14.4.2 Objects

Objects are covered in detail in "Objects" (§30) and the following chapter. Here, we mainly focus on how they differ from primitive values.

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

- Object literal:

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

    The object literal starts and ends with curly braces {}. It creates an object with two properties. The first property has the key `'first'` (a string) and the value `'Jane'`. The second property has the key `'last'` and the value `'Doe'`. For more information on object literals, consult "Object literals: properties" (§30.3.1).

- Array literal:

    ```
    const fruits = ['strawberry', 'apple'];
    ```

    The Array literal starts and ends with square brackets []. It creates an Array with two *elements*: `'strawberry'` and `'apple'`. For more information on Array literals, consult "Creating, reading, writing Arrays".

Objects are mutable by default

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

```
const obj = {};

obj.count = 2; // add a property
```

```
assert.equal(obj.count, 2);

obj.count = 3; // change a property
assert.equal(obj.count, 3);
```

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.name = 'Tessa';
assert.equal(b.name, 'Tessa');
```

JavaScript uses *garbage collection* to automatically manage memory:

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

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

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

Objects are *compared by identity*

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

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

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

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

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

How do they differ?

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

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

14.5.1   `typeof`

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

Table 14.1: The results of the `typeof` operator.

Table 14.1 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.

These are a few examples of using `typeof`:

```
> typeof undefined
'undefined'
> typeof 123n
'bigint'
> typeof 'abc'
```

```
'string'
> typeof {}
'object'
```

> 🧩 **Exercises: Two exercises on typeof**
>
> - exercises/values/typeof_exrc.mjs
> - Bonus: exercises/values/is_object_test.mjs

14.5.2 `instanceof`

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

```
x instanceof C
```

For example:

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

Primitive values are not instances of anything:

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

> 🧩 **Exercise: instanceof**
>
> exercises/values/instanceof_exrc.mjs

14.6 Classes and constructor functions

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

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

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

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

14.6.1   Constructor functions associated with primitive types

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

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

Each of these functions plays several roles – for example, `Number`:

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

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

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

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

- `Number` is a namespace/container object for tool functions for numbers – for example:

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

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

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

Wrapping primitive values

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

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

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

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

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

14.7   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.

14.7.1   Explicit conversion between types

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

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

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

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

The following table describes in more detail how this conversion works:

x	Object(x)
undefined	{}
null	{}
boolean	new Boolean(x)
number	new Number(x)
bigint	An instance of BigInt (new throws TypeError)
string	new String(x)
symbol	An instance of Symbol (new throws TypeError)
object	x

14.7.2   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, `Number.parseInt()` coerces its parameter to a string before parsing it. That explains the following result:

```
> Number.parseInt(123.45)
123
```

The number `123.45` is converted to the string `'123.45'` before it is parsed. Parsing stops before the first non-digit character, which is why the result is `123`.

🧩 **Exercise: Converting values to primitives**

`exercises/values/conversion_exrc.mjs`

Chapter 15

Operators

15.1 Making sense of operators

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

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

15.1.1 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);
```

15.1.2   Most operators only work with primitive values

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

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

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

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

Next, it concatenates the two strings:

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

15.2   The plus operator (+)

The plus operator works as follows in JavaScript:

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

String mode lets us use + to assemble strings:

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

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

```
> 4 + true
5
```

`Number(true)` is 1.

15.3   Assignment operators

15.3.1   The plain assignment operator

The plain assignment operator is used to change storage locations:

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

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

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

15.3.2   Compound assignment operators

JavaScript supports the following assignment operators:

- Arithmetic assignment operators: `+= -= *= /= %=` [ES1]
    - `+=` can also be used for string concatenation
    - Introduced later: `**=` [ES2016]
- Bitwise assignment operators: `&= ^= |=` [ES1]
- Bitwise shift assignment operators: `<<= >>= >>>=` [ES1]
- Logical assignment operators: `||= &&= ??=` [ES2021]

Logical assignment operators [ES2021]

Logical assignment operators work differently from other compound assignment operators:

Assignment operator	Equivalent to	Only assigns if **a** is
`a ||= b`	`a || (a = b)`	Falsy
`a &&= b`	`a && (a = b)`	Truthy
`a ??= b`	`a ?? (a = b)`	Nullish

Why is `a ||= b` equivalent to the following expression?

```
a || (a = b)
```

Why not to this expression?

```
a = a || b
```

The former expression has the benefit of short-circuiting: The assignment is only evaluated if `a` evaluates to `false`. Therefore, the assignment is only performed if it's necessary. In contrast, the latter expression always performs an assignment.

For more on `??=`, see "The nullish coalescing assignment operator (??=)" (§16.4.5).

The remaining compound assignment operators

For operators `op` other than `||` `&&` `??`, 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>');
```

15.4   Equality: == vs. ===

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

> ⚙ **Other names for == and ===**
>
> - `==` is also called *double equals*. Its official name in the language specification is *abstract equality comparison*.
> - `===` is also called *triple equals*.

15.4.1   Loose equality (== and !=)

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

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

Others less so:

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

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

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

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

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

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

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

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

15.4.2 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
```

15.4.3   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.

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.

Use case for ==: comparing with undefined or null

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

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

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

I prefer either of the following two alternatives:

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

A downside of the second alternative is that it accepts values other than undefined and null, but it is a well-established pattern in JavaScript (to be explained in detail in "Truthiness-based existence checks" (§17.3)).

The following three conditions are also roughly equivalent:

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

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

Method Object.is() compares two values:

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

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

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

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

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

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

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

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

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

15.5   Ordering operators

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

Table 15.1: JavaScript's ordering operators.

JavaScript's ordering operators (table 15.1) work for both numbers and strings:

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

<= and >= are based on strict equality.

⚠️ **The ordering operators don't work well for human languages**

The ordering operators don't work well for comparing text in a human language, e.g., when capitalization or accents are involved. The details are explained in "Comparing strings" (§22.6).

15.6   Various other operators

The following operators are covered elsewhere in this book:

- Operators for booleans, numbers, strings, objects
- The nullish coalescing operator (??) for default values

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

15.6.1   Comma operator

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

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

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

15.6.2   **void** operator

The void operator evaluates its operand and returns undefined:

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

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

Part IV

Primitive values

Chapter 16

The non-values `undefined` and `null`

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

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

16.1 `undefined` vs. `null`

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

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

Programmers may make the following distinction:

- `undefined` is the non-value used by the language (when something is uninitialized, etc.).
- `null` means "explicitly switched off". That is, it helps implement a type that comprises both meaningful values and a meta-value that stands for "no meaningful value". Such a type is called *option type* or *maybe type* in functional programming.

16.2   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.

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

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

16.2.2   Occurrences of `null`

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

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

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

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

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

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

16.3 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, 0n, ''
}
```

Is x either `undefined` or `null`?

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

Truthy means "is `true` if coerced to boolean". *Falsy* means "is `false` if coerced to boolean". Both concepts are explained properly in "Falsy and truthy values" (§17.2).

16.4 The nullish coalescing operator (??) for default values
      [ES2020]

Sometimes we receive a value and only want to use it if it isn't either `null` or `undefined`. Otherwise, we'd like to use a default value, as a fallback. We can do that via the *nullish coalescing operator* (??):

```
const valueToUse = receivedValue ?? defaultValue;
```

The following two expressions are equivalent:

```
a ?? b
a !== undefined && a !== null ? a : b
```

16.4.1   Example: counting matches

The following code shows a real-world example:

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

assert.equal(
  countMatches(/a/g, 'ababa'), 3);
assert.equal(
  countMatches(/b/g, 'ababa'), 2);
assert.equal(
  countMatches(/x/g, 'ababa'), 0);
```

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.

We also could have used optional chaining:

```
return matchResult?.length ?? 0;
```

16.4.2   Example: specifying a default value for a property

```
function getTitle(fileDesc) {
  return fileDesc.title ?? '(Untitled)';
}

const files = [
  {path: 'index.html', title: 'Home'},
  {path: 'tmp.html'},
];
assert.deepEqual(
  files.map(f => getTitle(f)),
  ['Home', '(Untitled)']);
```

16.4.3   Using destructuring for default values

In some cases, destructuring can also be used for default values – for example:

```
function getTitle(fileDesc) {
  const {title = '(Untitled)'} = fileDesc;
  return title;
}
```

16.4.4   Legacy approach: using logical Or (||) for default values

Before ECMAScript 2020 and the nullish coalescing operator, logical Or was used for default values. That has a downside.

|| works as expected for `undefined` and `null`:

```
> undefined || 'default'
'default'
> null || 'default'
'default'
```

But it also returns the default for all other falsy values – for example:

```
> false || 'default'
'default'
> 0 || 'default'
'default'
> 0n || 'default'
'default'
> '' || 'default'
'default'
```

Compare that to how ?? works:

```
> undefined ?? 'default'
'default'
> null ?? 'default'
'default'

> false ?? 'default'
false
> 0 ?? 'default'
0
> 0n ?? 'default'
0n
> '' ?? 'default'
''
```

16.4.5 The nullish coalescing assignment operator (??=) [ES2021]

??= is a logical assignment operator. The following two expressions are roughly equivalent:

```
a ??= b
a ?? (a = b)
```

That means that ??= is short-circuiting: The assignment is only made if `a` is `undefined` or `null`.

Example: using ??= to add missing properties

```
const books = [
  {
    isbn: '123',
  },
  {
```

```
    title: 'ECMAScript Language Specification',
    isbn: '456',
  },
];

// Add property .title where it's missing
for (const book of books) {
  book.title ??= '(Untitled)';
}

assert.deepEqual(
  books,
  [
    {
      isbn: '123',
      title: '(Untitled)',
    },
    {
      title: 'ECMAScript Language Specification',
      isbn: '456',
    },
  ]);
```

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

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

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

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

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

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

16.6   The history of **`undefined`** and **`null`**

In Java (which inspired many aspects of JavaScript), initialization values depend on the static type of a variable:

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

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

Chapter 17

Booleans

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

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

17.1   Converting to boolean

> ⚙️ **The meaning of "converting to [type]"**
>
> "Converting to [type]" is short for "Converting arbitrary values to values of type [type]".

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

- `Boolean(x)`
  Most descriptive; recommended.

- `x ? true : false`
  Uses the conditional operator (explained later in this chapter).

- `!!x`
  Uses the logical Not operator (!). This operator coerces its operand to boolean. It is applied a second time to get a non-negated result.

Table 17.1 describes how various values are converted to boolean.

x	Boolean(x)
undefined	false
null	false
boolean	x (no change)
number	0 →false, NaN →false
	Other numbers →true
bigint	0 →false
	Other numbers →true
string	'' →false
	Other strings →true
symbol	true
object	Always true

Table 17.1: Converting values to booleans.

17.2   Falsy and truthy values

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

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

In many programming languages, this condition is equivalent to:

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

However, in JavaScript, it is equivalent to:

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

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

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

Each value is either truthy or falsy. Consulting table 17.1, we can make an exhaustive list of falsy values:

- `undefined`
- `null`
- Boolean: `false`
- Numbers: `0`, `NaN`
- Bigint: `0n`
- String: `''`

All other values (including all objects) are truthy:

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

17.2.1   Checking for truthiness or falsiness

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

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

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

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

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

Exercise: Truthiness

```
exercises/booleans/truthiness_exrc.mjs
```

17.3   Truthiness-based existence checks

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

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

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

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

17.3.1   Pitfall: truthiness-based existence checks are imprecise

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

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

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

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

However, it is also skipped if:

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

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

17.3.2   Use case: was a parameter provided?

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

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

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

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

An alternative is to check for `undefined`:

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

17.3.3 Use case: does a property exist?

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

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

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

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

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

17.4 Conditional operator (? :)

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

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

It is evaluated as follows:

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

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

Examples:

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

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

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

Output:

```
then
```

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

The binary logical operators `&&` and `||` are *value-preserving* and *short-circuiting*.

17.5.1   Value-preservation

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

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

17.5.2   Short-circuiting

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

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

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

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

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

Output:

```
hello
```

17.5.3   Logical And (x && y)

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

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

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

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

Examples:

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

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

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

17.5.4 Logical Or (||)

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

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

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

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

Examples:

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

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

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

Legacy use case for logical Or (||): providing default values

ECMAScript 2020 introduced the nullish coalescing operator (`??`) for default values. Before that, logical Or was used for this purpose:

```
const valueToUse = receivedValue || defaultValue;
```

See "The nullish coalescing operator (??) for default values" (§16.4) for more information on `??` and the downsides of `||` in this case.

⚓ **Legacy exercise: Default values via the Or operator (||)**

`exercises/booleans/default_via_or_exrc.mjs`

17.6   Logical Not (!)

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

1. Evaluate x.
2. Is it truthy? Return false.
3. Otherwise, return true.

Examples:

```
> !false
true
> !true
false

> !0
true
> !123
false

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

Chapter 18

Numbers

JavaScript has two kinds of numeric values:

- *Numbers* are 64-bit floating point numbers and are also used for smaller integers (within a range of plus/minus 53 bits).
- *Bigints* represent integers with an arbitrary precision.

This chapter covers numbers. Bigints are covered later in this book.

18.1   Numbers are used for both floating point numbers and integers

The type `number` is used for both integers and floating point numbers in JavaScript:

```
98
123.45
```

However, all numbers are *doubles*, 64-bit floating point numbers implemented according to the *IEEE Standard for Floating-Point Arithmetic* (IEEE 754).

Integer numbers are simply floating point numbers without a decimal fraction:

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

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

18.2   Number literals

Let's examine literals for numbers.

18.2.1   Integer literals

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

```
// Binary (base 2)
assert.equal(0b11, 3); // ES6

// Octal (base 8)
assert.equal(0o10, 8); // ES6

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

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

18.2.2   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
```

18.2.3   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
```

18.2.4   Underscores (_) as separators in number literals [ES2021]

Grouping digits to make long numbers more readable has a long tradition. For example:

- In 1825, London had 1,335,000 inhabitants.
- The distance between Earth and Sun is 149,600,000 km.

Since ES2021, we can use underscores as separators in number literals:

```
const inhabitantsOfLondon = 1_335_000;
const distanceEarthSunInKm = 149_600_000;
```

With other bases, grouping is important, too:

```
const fileSystemPermission = 0b111_111_000;
const bytes = 0b1111_10101011_11110000_00001101;
const words = 0xFAB_F00D;
```

We can also use the separator in fractions and exponents:

```
const massOfElectronInKg = 9.109_383_56e-31;
const trillionInShortScale = 1e1_2;
```

Where can we put separators?

The locations of separators are restricted in two ways:

- We can only put underscores between two digits. Therefore, all of the following
  number literals are illegal:

  ```
  3_.141
  3._141

  1_e12
  1e_12

  _1464301  // valid variable name!
  1464301_

  0_b111111000
  0b_111111000
  ```

- We can't use more than one underscore in a row:

  ```
  123__456 // two underscores – not allowed
  ```

The motivation behind these restrictions is to keep parsing simple and to avoid strange
edge cases.

Parsing numbers with separators

The following functions for parsing numbers do not support separators:

- `Number()`
- `Number.parseInt()`
- `Number.parseFloat()`

For example:

```
> Number('123_456')
NaN
> Number.parseInt('123_456')
123
```

The rationale is that numeric separators are for code. Other kinds of input should be pro-
cessed differently.

18.3   Arithmetic operators

18.3.1   Binary arithmetic operators

Table 18.1 lists JavaScript's binary arithmetic operators.

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

Table 18.1: Binary arithmetic operators.

% is a remainder operator

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

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

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

18.3.2 Unary plus (+) and negation (-)

Table 18.2 summarizes the two operators *unary plus* (+) and *negation* (-). Both operators

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

Table 18.2: The operators unary plus (+) and negation (-).

coerce their operands to numbers:

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

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

18.3.3 Incrementing (++) and decrementing (--)

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

The decrementation operator -- works the same, but subtracts one from its operand. The next two examples explain the difference between the prefix and the suffix version.

Table 18.3 summarizes the incrementation and decrementation operators. Next, we'll look

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

Table 18.3: Incrementation operators and decrementation operators.

```
assert.equal(--bar, 2);
assert.equal(bar, 2);
```

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

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

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

Operands: not just variables

We can also apply these operators to property values:

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

And to Array elements:

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

⊞ **Exercise: Number operators**

exercises/numbers-math/is_odd_test.mjs

18.4   Converting to number

These are three ways of converting values to numbers:

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

Recommendation: use the descriptive `Number()`. Table 18.4 summarizes how it works. Examples:

x	Number(x)
`undefined`	`NaN`
`null`	`0`
boolean	`false →0`, `true →1`
number	x (no change)
bigint	`-1n →-1`, `1n →1`, etc.
string	`'' →0`
	Other →parsed number, ignoring leading/trailing whitespace
symbol	Throws `TypeError`
object	Configurable (e.g. via `.valueOf()`)

Table 18.4: Converting values to numbers.

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

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

assert.equal(Number(-123n), -123);
```

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

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

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

18.5   Error values

Two number values are returned when errors happen:

- `NaN`
- `Infinity`

18.5.1   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
```

Checking for `NaN`

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

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

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

```
const x = NaN;

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

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

Finding `NaN` in Arrays

Some Array methods can't find `NaN`:

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

Others can:

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

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

18.5.2 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
```

`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);
```

Checking for `Infinity`

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

```
const x = Infinity;

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

🧩 **Exercise: Comparing numbers**

`exercises/numbers-math/find_max_test.mjs`

18.6   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
```

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

Read on for an explanation of this phenomenon.

18.7   (Advanced)

All remaining sections of this chapter are advanced.

18.8   Background: floating point precision

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

```
> 0.1 + 0.2
0.30000000000000004
```

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

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

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

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

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

18.8.1 A simplified representation of floating point numbers

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

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

The new representation works like this:

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

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

- To encode the integer 123, we use the mantissa 123 and multiply it with 1 (10^0):

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

- To encode the integer −45, we use the mantissa −45 and, again, the exponent zero:

  ```
  > -45 * (10 ** 0)
  -45
  ```

- For the number 1.5, we imagine there being a point after the mantissa. We use the negative exponent −1 to move that point one digit to the left:

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

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

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

In other words: As soon as we have decimal digits, the exponent becomes negative. We can also write such a number as a fraction:

- Numerator (above the horizontal fraction bar): the mantissa
- Denominator (below the horizontal fraction bar): a 10 with a positive exponent ≥ 1.

For example:

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

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

- `1/10` can be represented. It already has the required format: a power of 10 in the denominator.

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

- 1/4 can be represented as `25/100`. We turned the 4 in the denominator into a power of 10 by multiplying the numerator and denominator by 25.

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

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

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

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

The only way to compute precisely with decimal fractions is by internally switching to base 10. For many programming languages, base 2 is the default and base 10 an option. For example:

- Java has the class `BigDecimal`.
- Python has the module `decimal`.

There are plans to add something similar to JavaScript: the ECMAScript proposal "Decimal". Until that happens, we can use libraries such as big.js.

18.9   Integer numbers in JavaScript

Integer numbers are normal (floating point) numbers without decimal fractions:

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

In this section, we'll look at a few tools for working with these pseudo-integers. JavaScript also supports *bigints*, which are real integers.

18.9.1   Converting to integer

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

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

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

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

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

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

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

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

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

For more information on rounding, consult "Rounding" (§19.3).

18.9.2 Ranges of integer numbers in JavaScript

These are important ranges of integer numbers in JavaScript:

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

18.9.3 Safe integers

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

$$[-(2^{53})+1, 2^{53}-1]$$

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

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

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

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

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

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

⟨puzzle⟩ Exercise: Detecting safe integers

```
exercises/numbers-math/is_safe_integer_test.mjs
```

Safe computations

Let's look at computations involving unsafe integers.

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

```
> 9007199254740990 + 3
9007199254740992
```

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

```
> 9007199254740995 - 10
9007199254740986
```

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

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

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

18.10 Bitwise operators

18.10.1 Internally, bitwise operators work with 32-bit integers

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

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

The types of operands and results

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

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

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

Displaying JavaScript numbers as unsigned 32-bit integers

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

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

18.10.2 Bitwise Not

The bitwise Not operator (table 18.5) inverts each binary digit of its operand:

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

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

Table 18.5: The bitwise Not operator.

This so-called *ones' complement* is similar to a negative for some arithmetic operations. For example, adding an integer to its ones' complement is always `-1`:

```
> 4 + ~4
-1
> -11 + ~-11
-1
```

18.10.3   Binary bitwise operators

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

Table 18.6: Binary bitwise operators.

The binary bitwise operators (table 18.6) combine the bits of their operands to produce their results:

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

18.10.4   Bitwise shift operators

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

Table 18.7: Bitwise shift operators.

The shift operators (table 18.7) move binary digits to the left or to the right:

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

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

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

18.10.5   **b32(): displaying unsigned 32-bit integers in binary notation**

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

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

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

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

18.11   Quick reference: numbers

18.11.1   Global functions for numbers

JavaScript has the following four global functions for numbers:

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

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

18.11.2   **Number.*: data properties**

- Number.EPSILON [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: $2.220446049250313080847263336181 \times 10^{-16}$

- Number.MAX_SAFE_INTEGER [ES6]

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

- `Number.MAX_VALUE` [ES1]

  The largest positive finite JavaScript number.

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

- `Number.MIN_SAFE_INTEGER` [ES6]

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

- `Number.MIN_VALUE` [ES1]

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

- `Number.NaN` [ES1]

  The same as the global variable `NaN`.

- `Number.NEGATIVE_INFINITY` [ES1]

  The same as `-Number.POSITIVE_INFINITY`.

- `Number.POSITIVE_INFINITY` [ES1]

  The same as the global variable `Infinity`.

18.11.3 `Number.*`: methods

- `Number.isFinite(num)` [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
  ```

- `Number.isInteger(num)` [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) [ES6]

  Returns true if num is the value NaN:

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

- Number.isSafeInteger(num) [ES6]

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

- Number.parseFloat(str) [ES6]

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

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

- Number.parseInt(str, radix=10) [ES6]

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

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

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

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

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

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

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

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

18.11.4 `Number.prototype.*`

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

- `Number.prototype.toExponential(fractionDigits?)` [ES3]
    - Returns a string that represents the number via exponential notation.
    - With `fractionDigits`, we can specify, how many digits should be shown of the number that is multiplied with the exponent.
        * The default is to show as many digits as necessary.

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

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

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

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

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

- `Number.prototype.toFixed(fractionDigits=0)` [ES3]

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

```
> 0.00000012.toString() // with exponent
'1.2e-7'
```

```
> 0.00000012.toFixed(10) // no exponent
'0.0000001200'
> 0.00000012.toFixed()
'0'
```

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

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

- `Number.prototype.toPrecision(precision?)` [ES3]

    - Works like `.toString()`, but `precision` specifies how many digits should be shown overall.
    - 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'
    ```

- `Number.prototype.toString(radix=10)` [ES1]

    Returns a string representation of the number.

    By default, we get a base 10 numeral as a result:

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

    If we want the numeral to have a different base, we can specify it via `radix`:

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

    ```
    > 255..toString(16) // hexadecimal (base 16)
    'ff'
    > 255.66796875.toString(16)
    'ff.ab'
    ```

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

    `Number.parseInt()` provides the inverse operation: it converts a string that contains an integer (no fraction!) numeral with a given base, to a number.

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

18.11.5 Sources

- Wikipedia
- TypeScript's built-in typings
- MDN web docs for JavaScript

- [ECMAScript language specification](#)

Chapter 19

`Math`

`Math` is an object with data properties and methods for processing numbers. You can see it as a poor man's module: It was created long before JavaScript had modules.

19.1 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.

19.2   Exponents, roots, logarithms

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

  Returns the cube root of x.

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

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

  Returns e^x (*e* being Euler's number). The inverse of `Math.log()`.

  ```
  > Math.exp(0)
  1
  > Math.exp(1) === Math.E
  true
  ```

- `Math.expm1(x: number): number` [ES6]

  Returns `Math.exp(x)-1`. The inverse of `Math.log1p()`. Very small numbers (fractions close to 0) are represented with a higher precision. Therefore, this function returns more precise values whenever `.exp()` returns values close to 1.

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

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

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

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

  Returns `Math.log(1 + x)`. The inverse of `Math.expm1()`. Very small numbers (fractions close to 0) are represented with a higher precision. Therefore, you can provide this function with a more precise argument whenever the argument for `.log()` is close to 1.

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

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

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

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

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

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

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

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

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

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

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

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

19.3   Rounding

Rounding means converting an arbitrary number to an integer (a number without a decimal fraction). The following functions implement different approaches to rounding.

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

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

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

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

Returns the largest (closest to +∞) integer i with i ≤ x.

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

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

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

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

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

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

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

Table 19.1 shows the results of the rounding functions for a few representative inputs.

	-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

Table 19.1: Rounding functions of Math. Note how things change with negative numbers because "larger" always means "closer to positive infinity".

19.4   Trigonometric Functions

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

```
function degreesToRadians(degrees) {
  return degrees / 180 * Math.PI;
}
assert.equal(degreesToRadians(90), Math.PI/2);

function radiansToDegrees(radians) {
  return radians / Math.PI * 180;
```

```
}
assert.equal(radiansToDegrees(Math.PI), 180);
```

- Math.acos(x: number): number [ES1]

  Returns the arc cosine (inverse cosine) of x.

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

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

  Returns the inverse hyperbolic cosine of x.

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

  Returns the arc sine (inverse sine) of x.

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

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

  Returns the inverse hyperbolic sine of x.

- Math.atan(x: number): number [ES1]

  Returns the arc tangent (inverse tangent) of x.

- Math.atanh(x: number): number [ES6]

  Returns the inverse hyperbolic tangent of x.

- Math.atan2(y: number, x: number): number [ES1]

  Returns the arc tangent of the quotient y/x.

- Math.cos(x: number): number [ES1]

  Returns the cosine of x.

  ```
  > Math.cos(0)
  1
  > Math.cos(Math.PI)
  -1
  ```

- Math.cosh(x: number): number [ES6]

  Returns the hyperbolic cosine of x.

- Math.hypot(...values: Array<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.

19.5   Various other functions

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

  Returns the absolute value of x.

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

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

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

  ```
  > Math.clz32(0b01000000000000000000000000000000)
  1
  > Math.clz32(0b00100000000000000000000000000000)
  2
  > Math.clz32(2)
  30
  > Math.clz32(1)
  31
  ```

- `Math.max(...values: Array<number>): number` [ES1]

  Converts `values` to numbers and returns the largest one.

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

- `Math.min(...values: Array<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 $0 \le n < 1$.

  ```
  /** Returns a random integer i with 0 <= 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
  ```

19.6 Sources

- Wikipedia
- TypeScript's built-in typings
- MDN web docs for JavaScript
- ECMAScript language specification

Chapter 20

Bigints – arbitrary-precision integers [ES2020] (advanced)

In this chapter, we take a look at *bigints*, JavaScript's integers whose storage space grows and shrinks as needed.

20.1   Why bigints?

Before ECMAScript 2020, JavaScript handled integers as follows:

- There only was a single type for floating point numbers and integers: 64-bit floating point numbers (IEEE 754 double precision).

- Under the hood, most JavaScript engines transparently supported integers: If a number has no decimal digits and is within a certain range, it can internally be stored as a genuine integer. This representation is called *small integer* and usually fits into 32 bits. For example, the range of small integers on the 64-bit version of the V8 engine is from -2^{31} to $2^{31}-1$ (source).

- JavaScript numbers could also represent integers beyond the small integer range, as floating point numbers. Here, the safe range is plus/minus 53 bits. For more information on this topic, see "Safe integers" (§18.9.3).

Sometimes, we need more than signed 53 bits – for example:

- X (formerly Twitter) uses 64-bit integers as IDs for posts (source). In JavaScript, these IDs had to be stored in strings.
- Financial technology uses so-called *big integers* (integers with arbitrary precision) to represent amounts of money. Internally, the amounts are multiplied so that the decimal numbers disappear. For example, USD amounts are multiplied by 100 so that the cents disappear.

20.2   Bigints

Bigint is a new primitive data type for integers. Bigints don't have a fixed storage size in bits; their sizes adapt to the integers they represent:

- Small integers are represented with fewer bits than large integers.
- There is no negative lower limit or positive upper limit for the integers that can be represented.

A bigint literal is a sequence of one or more digits, suffixed with an `n` – for example:

```
123n
```

Operators such as `-` and `*` are overloaded and work with bigints:

```
> 123n * 456n
56088n
```

Bigints are primitive values. `typeof` returns a new result for them:

```
> typeof 123n
'bigint'
```

20.2.1 Going beyond 53 bits for integers

JavaScript numbers are internally represented as a fraction multiplied by an exponent (see "Background: floating point precision" (§18.8) for details). As a consequence, if we go beyond the highest *safe integer* $2^{53}-1$, there are still *some* integers that can be represented, but with gaps between them:

```
> 2**53 - 2 // safe
9007199254740990
> 2**53 - 1 // safe
9007199254740991

> 2**53 // unsafe, same as next integer
9007199254740992
> 2**53 + 1
9007199254740992
> 2**53 + 2
9007199254740994
> 2**53 + 3
9007199254740996
> 2**53 + 4
9007199254740996
> 2**53 + 5
9007199254740996
```

Bigints enable us to go beyond 53 bits:

```
> 2n**53n
9007199254740992n
> 2n**53n + 1n
9007199254740993n
> 2n**53n + 2n
9007199254740994n
```

20.2.2 Example: using bigints

This is what using bigints looks like (code based on an example in the proposal):

```
/**
 * Takes a bigint as an argument and returns a bigint
 */
function nthPrime(nth) {
  if (typeof nth !== 'bigint') {
    throw new TypeError();
  }
  function isPrime(p) {
    for (let i = 2n; i < p; i++) {
      if (p % i === 0n) return false;
    }
    return true;
```

```
    }
  for (let i = 2n; ; i++) {
    if (isPrime(i)) {
      if (--nth === 0n) return i;
    }
  }
}

assert.deepEqual(
  [1n, 2n, 3n, 4n, 5n].map(nth => nthPrime(nth)),
  [2n, 3n, 5n, 7n, 11n]
);
```

20.3   Bigint literals

Like number literals, bigint literals support several bases:

- Decimal: `123n`
- Hexadecimal: `0xFFn`
- Binary: `0b1101n`
- Octal: `0o777n`

Negative bigints are produced by prefixing the unary minus operator: `-0123n`

20.3.1   Underscores (_) as separators in bigint literals [ES2021]

Just like in number literals, we can use underscores (_) as separators in bigint literals:

```
  const massOfEarthInKg = 6_000_000_000_000_000_000_000_000n;
```

Bigints are often used to represent money in the financial technical sector. Separators can help here, too:

```
  const priceInCents = 123_000_00n; // 123 thousand dollars
```

As with number literals, two restrictions apply:

- We can only put an underscore between two digits.
- We can use at most one underscore in a row.

20.4   Reusing number operators for bigints (overloading)

With most operators, we are not allowed to mix bigints and numbers. If we do, exceptions are thrown:

```
  > 2n + 1
  TypeError: Cannot mix BigInt and other types, use explicit conversions
```

The reason for this rule is that there is no general way of coercing a number and a bigint to a common type: numbers can't represent bigints beyond 53 bits, bigints can't represent fractions. Therefore, the exceptions warn us about typos that may lead to unexpected results.

For example, should the result of the following expression be `9007199254740993n` or `9007199254740992`?

```
2**53 + 1n
```

It is also not clear what the result of the following expression should be:

```
2n**53n * 3.3
```

20.4.1   Arithmetic operators

Binary +, binary -, *, ** work as expected:

```
> 7n * 3n
21n
```

It is OK to mix bigints and strings:

```
> 6n + ' apples'
'6 apples'
```

/, % round towards zero (like `Math.trunc()`):

```
> 1n / 2n
0n
```

Unary - works as expected:

```
> -(-64n)
64n
```

Unary + is not supported for bigints because much code relies on it coercing its operand to number:

```
> +23n
TypeError: Cannot convert a BigInt value to a number
```

20.4.2   Ordering operators

Ordering operators <, >, >=, <= work as expected:

```
> 17n <= 17n
true
> 3n > -1n
true
```

Comparing bigints and numbers does not pose any risks. Therefore, we can mix bigints and numbers:

```
> 3n > -1
true
```

20.4.3   Bitwise operators

Bitwise operators for numbers

Bitwise operators interpret numbers as 32-bit integers. These integers are either unsigned or signed. If they are signed, the negative of an integer is its *two's complement* (adding an

integer to its two's complement – while ignoring overflow – produces zero):

```
> 2**32-1 >> 0
-1
```

Due to these integers having a fixed size, their highest bits indicate their signs:

```
> 2**31 >> 0 // highest bit is 1
-2147483648
> 2**31 - 1 >> 0 // highest bit is 0
2147483647
```

Bitwise operators for bigints

For bigints, bitwise operators interpret a negative sign as an infinite two's complement – for example:

- -1 is ···111111 (ones extend infinitely to the left)
- -2 is ···111110
- -3 is ···111101
- -4 is ···111100

That is, a negative sign is more of an external flag and not represented as an actual bit.

Bitwise Not (~)

Bitwise Not (~) inverts all bits:

```
> ~0b10n
-3n
> ~0n
-1n
> ~-2n
1n
```

Binary bitwise operators (&, |, ^)

Applying binary bitwise operators to bigints works analogously to applying them to numbers:

```
> (0b1010n |  0b0111n).toString(2)
'1111'
> (0b1010n &  0b0111n).toString(2)
'10'

> (0b1010n | -1n).toString(2)
'-1'
> (0b1010n & -1n).toString(2)
'1010'
```

Bitwise signed shift operators (<< and >>)

The signed shift operators for bigints preserve the sign of a number:

```
> 2n << 1n
4n
> -2n << 1n
-4n

> 2n >> 1n
1n
> -2n >> 1n
-1n
```

Recall that `-1n` is a sequence of ones that extends infinitely to the left. That's why shifting it left doesn't change it:

```
> -1n >> 20n
-1n
```

Bitwise unsigned right shift operator (>>>)

There is no unsigned right shift operator for bigints:

```
> 2n >>> 1n
 TypeError: BigInts have no unsigned right shift, use >> instead
```

Why? The idea behind unsigned right shifting is that a zero is shifted in "from the left". In other words, the assumption is that there is a finite amount of binary digits.

However, with bigints, there is no "left", their binary digits extend infinitely. This is especially important with negative numbers.

Signed right shift works even with an infinite number of digits because the highest digit is preserved. Therefore, it can be adapted to bigints.

20.4.4   Loose equality (==) and inequality (!=)

Loose equality (==) and inequality (!=) coerce values:

```
> 0n == false
true
> 1n == true
true

> 123n == 123
true

> 123n == '123'
true
```

20.4.5   Strict equality (===) and inequality (!==)

Strict equality (===) and inequality (!==) only consider values to be equal if they have the same type:

```
> 123n === 123
false
> 123n === 123n
true
```

20.5   The wrapper constructor `BigInt`

Analogously to numbers, bigints have the associated wrapper constructor BigInt.

20.5.1   `BigInt` as a constructor and as a function

- new BigInt(): throws a TypeError.

- BigInt(x) converts arbitrary values x to bigint. This works similarly to Number(), with several differences which are summarized in table 20.1 and explained in more detail in the following subsections.

x	BigInt(x)
undefined	Throws TypeError
null	Throws TypeError
boolean	false →0n, true →1n
number	Example: 123 →123n
	Non-integer →throws RangeError
bigint	x (no change)
string	Example: '123' →123n
	Unparsable →throws SyntaxError
symbol	Throws TypeError
object	Configurable (e.g. via .valueOf())

Table 20.1: Converting values to bigints.

Converting `undefined` and `null`

A TypeError is thrown if x is either undefined or null:

```
> BigInt(undefined)
TypeError: Cannot convert undefined to a BigInt
> BigInt(null)
TypeError: Cannot convert null to a BigInt
```

Converting strings

If a string does not represent an integer, BigInt() throws a SyntaxError (whereas Number() returns the error value NaN):

```
> BigInt('abc')
SyntaxError: Cannot convert abc to a BigInt
```

The suffix 'n' is not allowed:

```
> BigInt('123n')
SyntaxError: Cannot convert 123n to a BigInt
```

All bases of bigint literals are allowed:

```
> BigInt('123')
123n
> BigInt('0xFF')
255n
> BigInt('0b1101')
13n
> BigInt('0o777')
511n
```

Non-integer numbers produce exceptions

```
> BigInt(123.45)
RangeError: The number 123.45 cannot be converted to a BigInt because
it is not an integer
> BigInt(123)
123n
```

Converting objects

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

```
> BigInt({valueOf() {return 123n}})
123n
```

20.5.2 `BigInt.prototype.*` methods

BigInt.prototype holds the methods "inherited" by primitive bigints:

- BigInt.prototype.toLocaleString(locales?, options?)
- BigInt.prototype.toString(radix?)
- BigInt.prototype.valueOf()

20.5.3 `BigInt.*` methods

- BigInt.asIntN(width, theInt)

Casts theInt to width bits (signed). This influences how the value is represented internally.

- BigInt.asUintN(width, theInt)

Casts theInt to width bits (unsigned).

20.5.4 Casting and 64-bit integers

Casting allows us to create integer values with a specific number of bits. If we want to restrict ourselves to just 64-bit integers, we have to always cast:

```
const uint64a = BigInt.asUintN(64, 12345n);
const uint64b = BigInt.asUintN(64, 67890n);
const result = BigInt.asUintN(64, uint64a * uint64b);
```

20.6 Coercing bigints to other primitive types

This table show what happens if we convert bigints to other primitive types:

Convert to	Explicit conversion	Coercion (implicit conversion)
boolean	`Boolean(0n)` →`false`	`!0n` →`true`
	`Boolean(int)` →`true`	`!int` →`false`
number	`Number(7n)` →`7` (example)	`+int` →`TypeError` (1)
string	`String(7n)` →`'7'` (example)	`''+7n` →`'7'` (example)

Footnote:

- (1) Unary + is not supported for bigints, because much code relies on it coercing its operand to number.

20.7 TypedArrays and DataView operations for 64-bit values

Thanks to bigints, Typed Arrays and DataViews can support 64-bit values:

- Typed Array constructors:
    - `BigInt64Array`
    - `BigUint64Array`
- DataView methods:
    - `DataView.prototype.getBigInt64()`
    - `DataView.prototype.setBigInt64()`
    - `DataView.prototype.getBigUint64()`
    - `DataView.prototype.setBigUint64()`

20.8 Bigints and JSON

The JSON standard is fixed and won't change. The upside is that old JSON parsing code will never be outdated. The downside is that JSON can't be extended to contain bigints.

Stringifying bigints throws exceptions:

```
> JSON.stringify(123n)
TypeError: Do not know how to serialize a BigInt
> JSON.stringify([123n])
TypeError: Do not know how to serialize a BigInt
```

20.8.1   Stringifying bigints

Therefore, our best option is to store bigints in strings:

```
const bigintPrefix = '[[bigint]]';

function bigintReplacer(_key, value) {
  if (typeof value === 'bigint') {
    return bigintPrefix + value;
  }
  return value;
}

const data = { value: 9007199254740993n };
assert.equal(
  JSON.stringify(data, bigintReplacer),
  '{"value":"[[bigint]]9007199254740993"}'
);
```

20.8.2   Parsing bigints

The following code shows how to parse strings such as the one that we have produced in the previous example.

```
function bigintReviver(_key, value) {
  if (typeof value === 'string' && value.startsWith(bigintPrefix)) {
    return BigInt(value.slice(bigintPrefix.length));
  }
  return value;
}

const str = '{"value":"[[bigint]]9007199254740993"}';
assert.deepEqual(
  JSON.parse(str, bigintReviver),
  { value: 9007199254740993n }
);
```

20.9   FAQ: Bigints

20.9.1   How do I decide when to use numbers and when to use bigints?

My recommendations:

- Use numbers for up to 53 bits and for Array indices. Rationale: They already appear everywhere and are handled efficiently by most engines (especially if they fit into 31 bits). Appearances include:
    - `Array.prototype.forEach()`
    - `Array.prototype.entries()`
- Use bigints for large numeric values: If your fraction-less values don't fit into 53 bits, you have no choice but to move to bigints.

All existing web APIs return and accept only numbers and will only upgrade to bigint on a case-by-case basis.

20.9.2   Why not just increase the precision of numbers in the same manner as is done for bigints?

One could conceivably split `number` into `integer` and `double`, but that would add many new complexities to the language (several integer-only operators etc.). I've sketched the consequences in a Gist.

Chapter 21

Unicode – a brief introduction (advanced)

Unicode is a standard for representing and managing text in most of the world's writing systems. Virtually all modern software that works with text, supports Unicode. The standard is maintained by the Unicode Consortium. A new version of the standard is published every year (with new emojis, etc.). Unicode version 1.0.0 was published in October 1991.

21.1   Code points vs. code units

Two concepts are crucial for understanding Unicode:

- *Code points* are numbers that represent the atomic parts of Unicode text. Most of them represent visible symbols but they can also have other meanings such as specifying an aspect of a symbol (the accent of a letter, the skin tone of an emoji, etc.).
- *Code units* are numbers that encode code points, to store or transmit Unicode text. One or more code units encode a single code point. Each code unit has the same size, which depends on the *encoding format* that is used. The most popular format,

UTF-8, has 8-bit code units.

21.1.1   Code points

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

- Plane 0: **Basic Multilingual Plane (BMP)**, 0x0000–0xFFFF
  - Contains characters for almost all modern languages (Latin characters, Asian characters, etc.) and many symbols.
- Plane 1: Supplementary Multilingual Plane (SMP), 0x10000–0x1FFFF
  - Supports historic writing systems (e.g., Egyptian hieroglyphs and cuneiform) and additional modern writing systems.
  - Supports emojis and many other symbols.
- Plane 2: Supplementary Ideographic Plane (SIP), 0x20000–0x2FFFF
  - Contains additional CJK (Chinese, Japanese, Korean) ideographs.
- Plane 3–13: Unassigned
- Plane 14: Supplementary Special-Purpose Plane (SSP), 0xE0000–0xEFFFF
  - Contains non-graphical characters such as tag characters and glyph variation selectors.
- Plane 15–16: Supplementary Private Use Area (S PUA A/B), 0x0F0000–0x10FFFF
  - Available for character assignment by parties outside the ISO and the Unicode Consortium. Not standardized.

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

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

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

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

21.1.2   Encoding Unicode code points: UTF-32, UTF-16, UTF-8

The main ways of encoding code points are three *Unicode Transformation Formats* (UTFs): UTF-32, UTF-16, UTF-8. The number at the end of each format indicates the size (in bits) of its code units.

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.

UTF-16 (Unicode Transformation Format 16)

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

- The BMP (first 16 bits of Unicode) is stored in single code units.

- Astral planes: The BMP comprises 0x10_000 code points. Given that Unicode has a total of 0x110_000 code points, we still need to encode the remaining 0x100_000 code points (20 bits). The BMP has two ranges of unassigned code points that provide the necessary storage:

  – Most significant 10 bits (*leading surrogate*, *high surrogate*): 0xD800-0xDBFF
  – Least significant 10 bits (*trailing surrogate*, *low surrogate*): 0xDC00-0xDFFF

As a consequence, each UTF-16 code unit is either:

- A BMP code point (a *scalar*)
- A leading surrogate
- A trailing surrogate

If a surrogate appears on its own, without its partner, it is called a *lone surrogate*.

This is how the bits of the code points are distributed among the surrogates:

```
0bhhhhhhhhhhhllllllllll // code point - 0x10000
0b110110hhhhhhhhhh      // 0xD800 + 0bhhhhhhhhhh
0b110111llllllllll      // 0xDC00 + 0bllllllllll
```

As an example, consider code point 0x1F642 (🙂) that is represented by two UTF-16 code units – 0xD83D and 0xDE42:

```
> '🙂'.codePointAt(0).toString(16)
'1f642'
> '🙂'.length
2
> '🙂'.split('')
[ '\uD83D', '\uDE42' ]
```

Let's derive the code units from the code point:

```
> (0x1F642 - 0x10000).toString(2).padStart(20, '0')
'00001111011001000010'
> (0xD800 + 0b0000111101).toString(16)
'd83d'
> (0xDC00 + 0b1001000010).toString(16)
'de42'
```

In contrast, code point 0x03C0 (π) is part of the BMP and therefore represented by a single UTF-16 code unit – 0x03C0:

```
> 'π'.length
1
```

UTF-8 (Unicode Transformation Format 8)

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

Code points	Code units
0000–007F	0bbbbbbb (7 bits)
0080–07FF	110bbbbb, 10bbbbbb (5+6 bits)
0800–FFFF	1110bbbb, 10bbbbbb, 10bbbbbb (4+6+6 bits)
10000–1FFFFF	11110bbb, 10bbbbbb, 10bbbbbb, 10bbbbbb (3+6+6+6 bits)

Notes:

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

Three examples:

Character	Code point	Code units
A	0x0041	01000001
π	0x03C0	11001111, 10000000
☺	0x1F642	11110000, 10011111, 10011001, 10000010

21.2   Encodings used in web development: UTF-16 and UTF-8

The Unicode encoding formats that are used in web development are: UTF-16 and UTF-8.

21.2.1   Source code internally: UTF-16

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

21.2.2   Strings: UTF-16

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

```
> const smiley = '☺';
> smiley.length
2
> smiley === '\uD83D\uDE42' // code units
true
```

For more information on Unicode and strings, consult "Atoms of text: code points, JavaScript characters, grapheme clusters" (§22.7).

21.2.3   Source code in files: UTF-8

HTML and JavaScript are almost always encoded as UTF-8 these days.

For example, this is how HTML files usually start now:

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

For HTML modules loaded in web browsers, the standard encoding is also UTF-8.

21.3   Grapheme clusters – the real characters

The concept of a character becomes remarkably complex once we consider the various writing systems of the world. That's why there are several different Unicode terms that all mean "character" in some way: *code point*, *grapheme cluster*, *glyph*, etc.

In Unicode, a *code point* is an atomic part of text.

However, a *grapheme cluster* corresponds most closely to a symbol displayed on screen or paper. It is defined as "a horizontally segmentable unit of text". Therefore, official Unicode documents also call it a *user-perceived character*. One or more code points are needed to encode a grapheme cluster.

For example, the Devanagari *kshi* is encoded by 4 code points. We use `Array.from()` to split a string into an Array with code points (for details, consult "Working with code points" (§22.7.1)):

```
> Array.from('क्षि')
[ 'क', 'ेा', 'ष', 'ि' ]
> 'क्षि'.length
4
```

Flag emojis are also grapheme clusters and composed of two code points – for example, the flag of Japan:

```
> Array.from('🇯🇵')
[ 'J', 'P' ]
> '🇯🇵'.length
4
```

21.3.1   Grapheme clusters vs. glyphs

A symbol is an abstract concept and part of written language:

- It is represented in computer memory by a *grapheme cluster* – a sequence of one or more numbers (code points).
- It is drawn on screen via *glyphs*. A glyph is an image and usually stored in a font. More than one glyph may be used to draw a single symbol – for example, the symbol "é" may be drawn by combining the glyph "e" with the glyph "´".

The distinction between a concept and its representation is subtle and can blur when talking about Unicode.

> ⬈ **More information on grapheme clusters**
>
> For more information, consult "Let's Stop Ascribing Meaning to Code Points" by Manish Goregaokar.

Chapter 22

Strings

22.1   Cheat sheet: strings

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

22.1.1   Working with strings

Literals for strings:

```
const str1 = 'Don\'t say "goodbye"'; // string literal
const str2 = "Don't say \"goodbye\""; // string literals
assert.equal(
  `As easy as ${123}!`, // template literal
  'As easy as 123!',
);
```

Backslashes are used to:

- Escape literal delimiters (first 2 lines of previous example)
- Represent special characters:
    - \\ represents a backslash
    - \n represents a newline
    - \r represents a carriage return
    - \t represents a tab

Inside a `String.raw` tagged template (line A), backslashes are treated as normal characters:

```
assert.equal(
  String.raw`\ \n\t`, // (A)
  '\\ \\n\\t',
);
```

Convertings values to strings:

```
> String(undefined)
'undefined'
> String(null)
'null'
> String(123.45)
'123.45'
> String(true)
'true'
```

Copying parts of a string

```
// There is no type for characters;
// reading characters produces strings:
const str3 = 'abc';
assert.equal(
  str3[2], 'c' // no negative indices allowed
);
assert.equal(
  str3.at(-1), 'c' // negative indices allowed
```

```
  );

  // Copying more than one character:
  assert.equal(
    'abc'.slice(0, 2), 'ab'
  );
```

Concatenating strings:

```
  assert.equal(
    'I bought ' + 3 + ' apples',
    'I bought 3 apples',
  );
```

```
  let str = '';
  str += 'I bought ';
  str += 3;
  str += ' apples';
  assert.equal(
    str, 'I bought 3 apples',
  );
```

22.1.2   JavaScript characters vs. code points vs. grapheme clusters

JavaScript characters are 16 bits in size. They are what is indexed in strings and what .length counts.

Code points are the atomic parts of Unicode text. Most of them fit into one JavaScript character, some of them occupy two (especially emojis):

```
  assert.equal(
    'A'.length, 1
  );
  assert.equal(
    '☺'.length, 2
  );
```

Grapheme clusters (*user-perceived characters*) represent written symbols. Each one comprises one or more code points.

Due to these facts, we shouldn't split text into JavaScript characters, we should split it into grapheme clusters. For more information on how to handle text, see "Atoms of text: code points, JavaScript characters, grapheme clusters" (§22.7).

22.1.3   String methods

This subsection gives a brief overview of the string API. There is a more comprehensive quick reference at the end of this chapter.

Finding substrings:

```
> 'abca'.includes('a')
true
> 'abca'.startsWith('ab')
true
> 'abca'.endsWith('ca')
true

> 'abca'.indexOf('a')
0
> 'abca'.lastIndexOf('a')
3
```

Splitting and joining:

```
assert.deepEqual(
  'a, b,c'.split(/, ?/),
  ['a', 'b', 'c']
);
assert.equal(
  ['a', 'b', 'c'].join(', '),
  'a, b, c'
);
```

Padding and trimming:

```
> '7'.padStart(3, '0')
'007'
> 'yes'.padEnd(6, '!')
'yes!!!'

> '\t abc\n '.trim()
'abc'
> '\t abc\n '.trimStart()
'abc\n '
> '\t abc\n '.trimEnd()
'\t abc'
```

Repeating and changing case:

```
> '*'.repeat(5)
'*****'
> '= b2b ='.toUpperCase()
'= B2B ='
> 'ΑΒΓ'.toLowerCase()
'αβγ'
```

22.2   Plain string literals

Plain string literals are delimited by either single quotes or double quotes:

```
const str1 = 'abc';
const str2 = "abc";
assert.equal(str1, str2);
```

Single quotes are used more often because it makes it easier to mention HTML, where double quotes are preferred.

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

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

22.2.1   Escaping

The backslash lets us create special characters:

- Unix line break: '\n'
- Windows line break: '\r\n'
- Tab: '\t'
- Backslash: '\\'

The backslash also lets us use the delimiter of a string literal inside that literal:

```
assert.equal(
  'She said: "Let\'s go!"',
  "She said: \"Let's go!\"");
```

22.3   Accessing JavaScript characters

JavaScript has no extra data type for characters – characters are always represented as strings.

```
const str = 'abc';

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

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

The characters we see on screen are called *grapheme clusters*. Most of them are represented by single JavaScript characters. However, there are also grapheme clusters (especially emojis) that are represented by multiple JavaScript characters:

```
> '🙂'.length
2
```

How that works is explained in "Atoms of text: code points, JavaScript characters, grapheme clusters" (§22.7).

22.4   String concatenation

22.4.1   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 we want to assemble a string, piece by piece:

```
let str = ''; // must be `let`!
str += 'Say it';
str += ' one more';
str += ' time';

assert.equal(str, 'Say it one more time');
```

> ⚙️ **Concatenating via + is efficient**
>
> Using + to assemble strings is quite efficient because most JavaScript engines internally optimize it.

> 🧩 **Exercise: Concatenating strings**
>
> `exercises/strings/concat_string_array_test.mjs`

22.4.2   Concatenating via Arrays (`.push()` and `.join()`)

Occasionally, taking a detour via an Array can be useful for concatenating strings – especially if there is to be a separator between them (such as `', '` in line A):

```
function getPackingList(isAbroad = false, days = 1) {
  const items = [];
  items.push('tooth brush');
  if (isAbroad) {
    items.push('passport');
  }
  if (days > 3) {
    items.push('water bottle');
  }
  return items.join(', '); // (A)
}
assert.equal(
  getPackingList(),
  'tooth brush'
);
assert.equal(
```

```
  getPackingList(true, 7),
  'tooth brush, passport, water bottle'
);
```

22.5 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 we convert a boolean to a string via `String()`, we generally can't convert it back via `Boolean()`:

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

The only string for which `Boolean()` returns `false`, is the empty string.

22.5.1 Stringifying objects

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

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

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

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

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

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

Stringifying functions, returns their source code:

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

22.5.2   Customizing the stringification of objects

We can override the built-in way of stringifying objects by implementing the method `toString()`:

```
const obj = {
  toString() {
    return 'hello';
  }
};

assert.equal(String(obj), 'hello');
```

22.5.3   An alternate way of stringifying values

The JSON data format is a text representation of JavaScript values. Therefore, `JSON.strin gify()` can also be used to convert values to strings:

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

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

Tip: The third parameter lets us switch on multiline output and specify how much to indent – for example:

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

This statement produces the following output:

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

22.6   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 ECMA-Script Internationalization API (`Intl`).

22.7   Atoms of text: code points, JavaScript characters, grapheme clusters

Quick recap of "Unicode – a brief introduction" (§21):

- *Code points* are the atomic parts of Unicode text. Each code point is 21 bits in size.

- JavaScript strings implement Unicode via the encoding format UTF-16. It uses one or two 16-bit *code units* to encode a single code point.

  - Each JavaScript character (as indexed in strings) is a code unit. In the JavaScript standard library, code units are also called *char codes*.

- *Grapheme clusters* (*user-perceived characters*) represent written symbols, as displayed on screen or paper. One or more code points are needed to encode a single grapheme cluster.

The following code demonstrates that a single code point comprises one or two JavaScript characters. We count the latter via `.length`:

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

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

The following table summarizes the concepts we have just explored:

Entity	Size	Encoded via
JavaScript character (UTF-16 code unit)	16 bits	–
Unicode code point	21 bits	1–2 code units
Unicode grapheme cluster		1+ code points

22.7.1 Working with code points

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

A *Unicode code point escape* lets us specify a code point hexadecimally (1–5 digits). It produces one or two JavaScript characters.

```
> '\u{1F642}'
'🙂'
```

> ⚙ **Unicode escape sequences**
>
> In the ECMAScript language specification, *Unicode code point escapes* and *Unicode code unit escapes* (which we'll encounter later) are called *Unicode escape sequences*.

`String.fromCodePoint()` converts a single code point to 1–2 JavaScript characters:

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

`.codePointAt()` converts 1–2 JavaScript characters to a single code point:

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

We can *iterate* over a string, which visits code points (not JavaScript characters). Iteration is described later in this book. One way of iterating is via a `for-of` loop:

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

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

Output:

```
🙂
a
```

`Array.from()` is also based on iteration and visits code points:

```
> Array.from('🙂a')
[ '🙂', 'a' ]
```

That makes it a good tool for counting code points:

```
> Array.from('🙂a').length
2
> '🙂a'.length
3
```

22.7.2   Working with code units (char codes)

Indices and lengths of strings are based on JavaScript characters (as represented by UTF-16 code units).

To specify a code unit hexadecimally, we can use a *Unicode code unit escape* with exactly four hexadecimal digits:

```
> '\uD83D\uDE42'
'🙂'
```

And we can use `String.fromCharCode()`. *Char code* is the standard library's name for *code unit*:

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

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

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

22.7.3   ASCII escapes

If the code point of a character is below 256, we can refer to it via a *ASCII escape* with exactly two hexadecimal digits:

```
> 'He\x6C\x6Co'
'Hello'
```

(The official name of ASCII escapes is *Hexadecimal escape sequences* – it was the first escape that used hexadecimal numbers.)

22.7.4   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 points.

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, we can use one of several libraries that are available (do a web search for "JavaScript grapheme").

22.8   Quick reference: Strings

22.8.1   Converting to string

Table 22.1 describes how various values are converted to strings.

x	`String(x)`
`undefined`	`'undefined'`
`null`	`'null'`
boolean	`false` →`'false'`, `true` →`'true'`
number	Example: `123` →`'123'`
bigint	Example: `123n` →`'123'`
string	`x` (input, unchanged)
symbol	Example: `Symbol('abc')` →`'Symbol(abc)'`
object	Configurable via, e.g., `toString()`

Table 22.1: Converting values to strings.

22.8.2   Numeric values of text atoms

- **Char code**: number representing a JavaScript character. JavaScript's name for *Unicode code unit*.
  - Size: 16 bits, unsigned
  - Convert number to string: `String.fromCharCode()` [ES1]
  - Convert string to number: string method `.charCodeAt()` [ES1]
- **Code point**: number representing an atomic part of Unicode text.
  - Size: 21 bits, unsigned (17 planes, 16 bits each)
  - Convert number to string: `String.fromCodePoint()` [ES6]
  - Convert string to number: string method `.codePointAt()` [ES6]

22.8.3   `String.prototype.*`: finding and matching

- `String.prototype.startsWith(searchString, startPos=0)` [ES6]

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

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

- `String.prototype.endsWith(searchString, endPos=this.length)` [ES6]

  Returns `true` if the string would end with `searchString` if its length were `endPos`. Returns `false` otherwise.

  ```
  > 'poem.txt'.endsWith('.txt')
  true
  > 'abcde'.endsWith('cd', 4)
  true
  ```

- `String.prototype.includes(searchString, startPos=0)` [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
```

- `String.prototype.indexOf(searchString, minIndex=0)` [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
```

- `String.prototype.lastIndexOf(searchString, maxIndex=Infinity)` [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
```

- `String.prototype.match(regExpOrString)` [ES3]

  - (1 of 2) `regExpOrString` is RegExp without `/g` or string.

    ```
    match(
      regExpOrString: string | RegExp
    ): null | RegExpMatchArray
    ```

  If `regExpOrString` is a regular expression with flag `/g` not set, then `.match()` returns the first match for `regExpOrString` within the string. Or `null` if there is no match.

  If `regExpOrString` is a string, it is used to create a regular expression (think parameter of `new RegExp()`) before performing the previously mentioned steps.

  The result has the following type:

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

```
    };
  }
```

Numbered capture groups become Array indices (which is why this type extends `Array`). Named capture groups](#named-capture-groups) (ES2018) become properties of `.groups`. In this mode, `.match()` works like [RegExp.prototype.exec().

Examples:

```
> 'ababb'.match(/a(b+)/)
{ 0: 'ab', 1: 'b', index: 0, input: 'ababb', groups: undefined }
> 'ababb'.match(/a(?<foo>b+)/)
{ 0: 'ab', 1: 'b', index: 0, input: 'ababb', groups: { foo: 'b' } }
> 'abab'.match(/x/)
null
```

– (2 of 2) `regExpOrString` is RegExp with `/g`.

```
match(
  regExpOrString: RegExp
): null | Array<string>
```

If flag `/g` of `regExpOrString` 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
```

- `String.prototype.search(regExpOrString)` [ES3]

  Returns the index at which `regExpOrString` occurs within the string. If `regExpOrString` is a string, it is used to create a regular expression (think parameter of `new RegExp()`).

```
> 'a2b'.search(/[0-9]/)
1
> 'a2b'.search('[0-9]')
1
```

22.8.4 `String.prototype.*`: extracting

- `String.prototype.slice(start=0, end=this.length)` [ES3]

  Returns the substring of the string that starts at (including) index `start` and ends at (excluding) index `end`. If an index is negative, it is added to `.length` before it is used (-1 becomes `this.length-1`, etc.).

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

```
  'bc'
> 'abc'.slice(-2)
  'bc'
```

- `String.prototype.at(index: number)` [ES2022]

    - Returns the JavaScript character at `index` as a string.
    - If the index is out of bounds, it returns `undefined`.
    - If `index` is negative, it is added to `.length` before it is used (`-1` becomes `this
      .length - 1`, etc.).

    ```
    > 'abc'.at(0)
    'a'
    > 'abc'.at(-1)
    'c'
    ```

- `String.prototype.split(separator, limit?)` [ES3]

  Splits the string into an Array of substrings – the strings that occur between the
  separators.

  The separator can be a string:

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

  It can also be a regular expression:

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

  The last invocation demonstrates that captures made by groups in the regular ex-
  pression become elements of the returned Array.

  If we want the separators to be part of the returned string fragments, we can use a
  regular expression with a lookbehind assertion:

  ```
  > 'a : b : c'.split(/(?<=:)/)
  [ 'a :', ' b :', ' c' ]
  ```

  Thanks to the lookbehind assertion, the regular expression used for splitting matches
  but doesn't capture any characters (which would be taken away from the output
  fragments).

  Warning about `.split('')`: Using the method this way splits a string into JavaScript
  characters. That doesn't work well when dealing with astral code points (which are
  encoded as two JavaScript characters). For example, emojis are astral:

  ```
  > '☺X☺'.split('')
  [ '\uD83D', '\uDE42', 'X', '\uD83D', '\uDE42' ]
  ```

  Instead, it is better to use `Array.from()` (or spreading):

```
> Array.from('😊X😊')
[ '😊', 'X', '😊' ]
```

- `String.prototype.substring(start, end=this.length)` [ES1]

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

22.8.5 `String.prototype.*`: combining

- `String.prototype.concat(...strings)` [ES3]

  Returns the concatenation of the string and `strings`. `'a'.concat('b')` is equivalent to `'a'+'b'`. The latter is much more popular.

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

- `String.prototype.padEnd(len, fillString=' ')` [ES2017]

  Appends (fragments of) `fillString` to the string until it has the desired length `len`. If it already has or exceeds `len`, then it is returned without any changes.

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

- `String.prototype.padStart(len, fillString=' ')` [ES2017]

  Prepends (fragments of) `fillString` to the string until it has the desired length `len`. If it already has or exceeds `len`, then it is returned without any changes.

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

- `String.prototype.repeat(count=0)` [ES6]

  Returns the string, concatenated `count` times.

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

22.8.6 `String.prototype.*`: transforming

- `String.prototype.replaceAll(searchValue, replaceValue)` [ES2021]

> 💡 **What to do if you can't use `.replaceAll()`**
>
> If `.replaceAll()` isn't available on your targeted platform, you can use `.replace()` instead. How is explained in "`str.replace(searchValue, replacementValue)` [ES3]".

  - (1 of 2) `replaceValue` is string.

    ```
    replaceAll(
      searchValue: string | RegExp,
      replaceValue: string
    ): string
    ```

    Replaces all matches of `searchValue` with `replaceValue`. If `searchValue` is a regular expression without flag `/g`, a `TypeError` is thrown.

    ```
    > 'x.x.'.replaceAll('.', '#') // interpreted literally
    'x#x#'
    > 'x.x.'.replaceAll(/./g, '#')
    '####'
    > 'x.x.'.replaceAll(/./, '#')
    TypeError: String.prototype.replaceAll called with
    a non-global RegExp argument
    ```

    Special characters in `replaceValue` are:

      * `$$`: becomes `$`
      * `$n`: becomes the capture of numbered group `n` (alas, `$0` stands for the string `'$0'`, it does not refer to the complete match)
      * `$&`: becomes the complete match
      * `` $` ``: becomes everything before the match
      * `$'`: becomes everything after the match

    Examples:

    ```
    > 'a 1995-12 b'.replaceAll(/([0-9]{4})-([0-9]{2})/g, '|$2|')
    'a |12| b'
    > 'a 1995-12 b'.replaceAll(/([0-9]{4})-([0-9]{2})/g, '|$&|')
    'a |1995-12| b'
    > 'a 1995-12 b'.replaceAll(/([0-9]{4})-([0-9]{2})/g, '|$`|')
    'a |a | b'
    ```

    Named capture groups (ES2018) are supported, too:

      * `$<name>` becomes the capture of named group `name`

    Example:

```
assert.equal(
  'a 1995-12 b'.replaceAll(
    /(?<year>[0-9]{4})-(?<month>[0-9]{2})/g, '|$<month>|'),
  'a |12| b');
```

– (2 of 2) `replaceValue` is function.

```
replaceAll(
  searchValue: string | RegExp,
  replaceValue: (...args: Array<any>) => string
): string
```

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})/g;
const replacer = (all, year, month) => '|' + all + '|';
assert.equal(
  'a 1995-12 b'.replaceAll(regexp, replacer),
  'a |1995-12| b');
```

Named capture groups (ES2018) are supported, too. If there are any, an argument is added at the end with an object whose properties contain the captures:

```
const regexp = /(?<year>[0-9]{4})-(?<month>[0-9]{2})/g;
const replacer = (...args) => {
  const groups=args.pop();
  return '|' + groups.month + '|';
};
assert.equal(
  'a 1995-12 b'.replaceAll(regexp, replacer),
  'a |12| b');
```

• `String.prototype.replace(searchValue, replaceValue)` [ES3]

For more information on this method, see "`str.replace(searchValue, replace-mentValue)` [ES3]".

– (1 of 2) `replaceValue` is string or RegExp without `/g`.

```
replace(
  searchValue: string | RegExp,
  replaceValue: string
): string
```

Works similarly to `.replaceAll()`, but only replaces the first occurrence:

```
> 'x.x.'.replace('.', '#') // interpreted literally
'x#x.'
> 'x.x.'.replace(/./, '#')
'#.x.'
```

- (1 of 2) `replaceValue` is RegExp with `/g`.

  ```
  replace(
    searchValue: string | RegExp,
    replaceValue: (...args: Array<any>) => string
  ): string
  ```

  Works exactly like `.replaceAll()`.

- `String.prototype.toUpperCase()` [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'
  ```

- `String.prototype.toLowerCase()` [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'
  ```

- `String.prototype.trim()` [ES5]

  Returns a copy of the string in which all leading and trailing whitespace (spaces, tabs, line terminators, etc.) is gone.

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

- `String.prototype.trimStart()` [ES2019]

  Similar to `.trim()` but only the beginning of the string is trimmed:

  ```
  > '   abc   '.trimStart()
  'abc   '
  ```

- `String.prototype.trimEnd()` [ES2019]

Similar to `.trim()` but only the end of the string is trimmed:

```
> '  abc  '.trimEnd()
'  abc'
```

- `String.prototype.normalize(form = 'NFC')` [ES6]

    – Normalizes the string according to the Unicode Normalization Forms.
    – Values of `form`: `'NFC'`, `'NFD'`, `'NFKC'`, `'NFKD'`

- `String.prototype.isWellFormed()` [ES2024]

    Returns `true` if a string is ill-formed and contains *lone surrogates* (see `.toWellFormed()` for more information). Otherwise, it returns `false`.

```
> '☺'.split('') // split into code units
[ '\uD83D', '\uDE42' ]
> '\uD83D\uDE42'.isWellFormed()
true
> '\uD83D\uDE42\uD83D'.isWellFormed() // lone surrogate 0xD83D
false
```

- `String.prototype.toWellFormed()` [ES2024]

    Each JavaScript string character is a UTF-16 code unit. One code point is encoded as either one UTF-16 code unit or two UTF-16 code unit. In the latter case, the two code units are called *leading surrogate* and *trailing surrogate*. A surrogate without its partner is called a *lone surrogate*. A string with one or more lone surrogates is *ill-formed*.

    `.toWellFormed()` converts an ill-formed string to a well-formed one by replacing each lone surrogate with code point 0xFFFD ("replacement character"). That character is often displayed as a ◆ (a black rhombus with a white question mark). It is located in the *Specials* Unicode block of characters, at the very end of the *Basic Multilingual Plane*. This is what Wikipedia says about the replacement character: "It is used to indicate problems when a system is unable to render a stream of data to correct symbols."

```
assert.deepEqual(
  '☺'.split(''), // split into code units
  ['\uD83D', '\uDE42']
);
assert.deepEqual(
  // 0xD83D is a lone surrogate
  '\uD83D\uDE42\uD83D'.toWellFormed().split(''),
  ['\uD83D', '\uDE42', '\uFFFD']
);
```

22.8.7 Sources of this quick reference

- ECMAScript language specification
- TypeScript's built-in typings
- MDN web docs for JavaScript

Exercise: Using string methods

`exercises/strings/remove_extension_test.mjs`

Chapter 23

Using template literals and tagged templates [ES6]

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

23.1 Disambiguation: "template"

The following three things are significantly different despite all having *template* in their names and despite all of them looking similar:

- A *text template* is a function from data to text. It is frequently used in web devel-

opment and often defined via text files. For example, the following text defines a template for the library Handlebars:

```
<div class="entry">
  <h1>{{title}}</h1>
  <div class="body">
    {{body}}
  </div>
</div>
```

This template has two blanks to be filled in: `title` and `body`. It is used like this:

```
// First step: retrieve the template text, e.g. from a text file.
const tmplFunc = Handlebars.compile(TMPL_TEXT); // compile string
const data = {title: 'My page', body: 'Welcome to my page!'};
const html = tmplFunc(data);
```

- A *template literal* is similar to a string literal, but has additional features – for example, interpolation. It is delimited by backticks:

  ```
  const num = 5;
  assert.equal(`Count: ${num}!`, 'Count: 5!');
  ```

- Syntactically, a *tagged template* is a template literal that follows a function (or rather, an expression that evaluates to a function). That leads to the function being called. Its arguments are derived from the contents of the template literal.

  ```
  const getArgs = (...args) => args;
  assert.deepEqual(
    getArgs`Count: ${5}!`,
    [['Count: ', '!'], 5] );
  ```

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

23.2 Template literals

A template literal has two new features compared to a normal string literal.

First, it supports *string interpolation*: if we put a dynamically computed value inside a `${}`, it is converted to a string and inserted into the string returned by the literal.

```
const MAX = 100;
function doSomeWork(x) {
  if (x > MAX) {
    throw new Error(`At most ${MAX} allowed: ${x}!`);
  }
  // ···
}
assert.throws(
  () => doSomeWork(101),
  {message: 'At most 100 allowed: 101!'});
```

Second, template literals can span multiple lines:

```
const str = `this is
a text with
multiple lines`;
```

Template literals always produce strings.

23.3   Tagged templates

The expression in line A is a *tagged template*. It is equivalent to invoking `tagFunc()` with the arguments listed in the Array in line B.

```
function tagFunc(...args) {
  return args;
}

const setting = 'dark mode';
const value = true;

assert.deepEqual(
  tagFunc`Setting ${setting} is ${value}!`, // (A)
  [['Setting ', ' is ', '!'], 'dark mode', true] // (B)
);
```

The function `tagFunc` before the first backtick is called a *tag function*. Its arguments are:

- *Template strings* (first argument): an Array with the text fragments surrounding the interpolations `${}`.

    – In the example: `['Setting ', ' is ', '!']`

- *Substitutions* (remaining arguments): the interpolated values.

    – In the example: `'dark mode'` and `true`

The static (fixed) parts of the literal (the template strings) are kept separate from the dynamic parts (the substitutions).

A tag function can return arbitrary values.

23.3.1   Cooked vs. raw template strings (advanced)

So far, we have only seen the *cooked interpretation* of template strings. But tag functions actually get two interpretations:

- A *cooked interpretation* where backslashes have special meaning. For example, `\t` produces a tab character. This interpretation of the template strings is stored as an Array in the first argument.

- A *raw interpretation* where backslashes do not have special meaning. For example, `\t` produces two characters – a backslash and a `t`. This interpretation of the template strings is stored in property `.raw` of the first argument (an Array).

The raw interpretation enables raw string literals via `String.raw` (described later) and similar applications.

The following tag function `cookedRaw` uses both interpretations:

```
function cookedRaw(templateStrings, ...substitutions) {
  return {
    cooked: Array.from(templateStrings), // copy only Array elements
    raw: templateStrings.raw,
    substitutions,
  };
}
assert.deepEqual(
  cookedRaw`\tab${'subst'}\newline\\`,
  {
    cooked: ['\tab', '\newline\\'],
    raw:    ['\\tab', '\\newline\\\\'],
    substitutions: ['subst'],
  });
```

We can also use Unicode code point escapes (`\u{1F642}`), Unicode code unit escapes (`\u03A9`), and ASCII escapes (`\x52`) in tagged templates:

```
assert.deepEqual(
  cookedRaw`\u{54}\u0065\x78t`,
  {
    cooked: ['Text'],
    raw:    ['\\u{54}\\u0065\\x78t'],
    substitutions: [],
  });
```

If the syntax of one of these escapes isn't correct, the corresponding cooked template string is `undefined`, while the raw version is still verbatim:

```
assert.deepEqual(
  cookedRaw`\uu\xx ${1} after`,
  {
    cooked: [undefined, ' after'],
    raw:    ['\\uu\\xx ', ' after'],
    substitutions: [1],
  });
```

Incorrect escapes produce syntax errors in template literals and string literals. Before ES2018, they even produced errors in tagged templates. Why was that changed? We can now use tagged templates for text that was previously illegal – for example:

```
windowsPath`C:\uuu\xxx\111`
latex`\unicode`
```

23.4   Examples of tagged templates (as provided via libraries)

Tagged templates are great for supporting small embedded languages (so-called *domain-specific languages*). We'll continue with a few examples.

23.4.1   Tag function library: lit-html

Lit is a library for building web components that uses tagged templates for HTML templating:

```
@customElement('my-element')
class MyElement extends LitElement {

  // ···

  render() {
    return html`
      <ul>
        ${repeat(
          this.items,
          (item) => item.id,
          (item, index) => html`<li>${index}: ${item.name}</li>`
        )}
      </ul>
    `;
  }
}
```

`repeat()` is a custom function for looping. Its second parameter produces unique keys for the values returned by the third parameter. Note the nested tagged template used by that parameter.

23.4.2   Tag function library: regex

The library "regex" by Steven Levithan provides template tags that help with creating regular expressions and enable advanced features. The following example demonstrates how it works:

```
import {regex, partial} from 'regex';

const RE_YEAR = partial`(?<year>[0-9]{4})`;
const RE_MONTH = partial`(?<month>[0-9]{2})`;
const RE_DAY = partial`(?<day>[0-9]{2})`;
const RE_DATE = regex('g')`
  ${RE_YEAR} # 4 digits
  -
  ${RE_MONTH} # 2 digits
  -
  ${RE_DAY} # 2 digits
`;
```

```
const match = RE_DATE.exec('2017-01-27');
assert.equal(match.groups.year, '2017');
```

The following flags are switched on by default:

- Flag /v
- Flag /x (emulated) enables insignificant whitespace and line comments via #.
- Flag /n (emulated) enables *named capture only mode*, which prevents the grouping metacharacters (···) from capturing.

23.4.3 Tag function library: graphql-tag

The library graphql-tag lets us 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.

23.5 Raw string literals

Raw string literals are implemented via the tag function String.raw. They are string literals where backslashes don't do anything special (such as escaping characters, etc.):

```
assert.equal(String.raw`\back`, '\\back');
```

This helps whenever data contains backslashes – for example, strings with regular expressions:

```
const regex1 = /^\./;
const regex2 = new RegExp('^\\.');
const regex3 = new RegExp(String.raw`^\.`);
```

All three regular expressions are equivalent. With a normal string literal, we have to write the backslash twice, to escape it for that literal. With a raw string literal, we don't have to do that.

Raw string literals are also useful for specifying Windows filename paths:

```
const WIN_PATH = String.raw`C:\foo\bar`;
assert.equal(WIN_PATH, 'C:\\foo\\bar');
```

23.6 (Advanced)

All remaining sections are advanced

23.7 Multiline template literals and indentation

If we put multiline text in template literals, two goals are in conflict: On one hand, the template literal should be indented to fit inside the source code. On the other hand, the lines of its content should start in the leftmost column.

For example:

```
function div(text) {
  return `
    <div>
      ${text}
    </div>
  `;
}
console.log('Output:');
console.log(
  div('Hello!')
  // Replace spaces with mid-dots:
  .replace(/ /g, '·')
  // Replace \n with #\n:
  .replace(/\n/g, '#\n')
);
```

Due to the indentation, the template literal fits well into the source code. Alas, the output is also indented. And we don't want the return at the beginning and the return plus two spaces at the end.

```
Output:
#
····<div>#
······Hello!#
····</div>#
··
```

There are two ways to fix this: via a tagged template or by trimming the result of the template literal.

23.7.1 Fix: template tag for dedenting

The first fix is to use a custom template tag that removes the unwanted whitespace. It uses the first line after the initial line break to determine in which column the text starts and shortens the indentation everywhere. It also removes the line break at the very beginning and the indentation at the very end. One such template tag is dedent by Desmond Brand:

```
import dedent from 'dedent';
function divDedented(text) {
```

```
  return dedent`
    <div>
      ${text}
    </div>
  `.replace(/\n/g, '#\n');
}
console.log('Output:');
console.log(divDedented('Hello!'));
```

This time, the output is not indented:

```
Output:
<div>#
  Hello!#
</div>
```

23.7.2   Fix: `.trim()`

The second fix is quicker, but also dirtier:

```
function divDedented(text) {
  return `
<div>
  ${text}
</div>
  `.trim().replace(/\n/g, '#\n');
}
console.log('Output:');
console.log(divDedented('Hello!'));
```

The string method `.trim()` removes the superfluous whitespace at the beginning and at
the end, but the content itself must start in the leftmost column. The advantage of this
solution is that we don't need a custom tag function. The downside is that it looks ugly.

The output is the same as with `dedent`:

```
Output:
<div>#
  Hello!#
</div>
```

23.8   Simple templating via template literals

While template literals look like text templates, it is not immediately obvious how to use
them for (text) templating: A text template gets its data from an object, while a template
literal gets its data from variables. The solution is to use a template literal in the body of a
function whose parameter receives the templating data – for example:

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

23.8.1 A more complex example

As a more complex example, we'd like to take an Array of addresses and produce an HTML table. This is the Array:

```
const addresses = [
  { first: '<Jane>', last: 'Bond' },
  { first: 'Lars', last: '<Croft>' },
];
```

The function `tmpl()` that produces the HTML table looks as follows:

```
1  const tmpl = (addrs) => `
2  <table>
3    ${addrs.map(
4      (addr) => `
5        <tr>
6          <td>${escapeHtml(addr.first)}</td>
7          <td>${escapeHtml(addr.last)}</td>
8        </tr>
9        `.trim()
10   ).join('')}
11 </table>
12 `.trim();
```

This code contains two templating functions:

- The first one (line 1) takes `addrs`, an Array with addresses, and returns a string with a table.
- The second one (line 4) takes `addr`, an object containing an address, and returns a string with a table row. Note the `.trim()` at the end, which removes unnecessary whitespace.

The first templating function produces its result by wrapping a table element around an Array that it joins into a string (line 10). That Array is produced by mapping the second templating function to each element of `addrs` (line 3). It therefore contains strings with table rows.

The helper function `escapeHtml()` is used to escape special HTML characters (line 6 and line 7). Its implementation is shown in the next subsection.

Let us call `tmpl()` with the addresses and log the result:

```
console.log(tmpl(addresses));
```

The output is:

```
<table>
  <tr>
        <td>&lt;Jane&gt;</td>
        <td>Bond</td>
      </tr><tr>
        <td>Lars</td>
        <td>&lt;Croft&gt;</td>
```

```
    </tr>
  </table>
```

23.8.2 Simple HTML-escaping

The following function escapes plain text so that it is displayed verbatim in HTML:

```
function escapeHtml(str) {
  return str
    .replace(/&/g, '&amp;') // first!
    .replace(/>/g, '&gt;')
    .replace(/</g, '&lt;')
    .replace(/"/g, '&quot;')
    .replace(/'/g, '&#39;')
    .replace(/`/g, '&#96;')
    ;
}
assert.equal(
  escapeHtml('Rock & Roll'), 'Rock &amp; Roll');
assert.equal(
  escapeHtml('<blank>'), '&lt;blank&gt;');
```

> **⤷ Exercise: HTML templating**
>
> Exercise with bonus challenge: `exercises/template-literals/templating_test.mjs`

Chapter 24

Symbols [ES6]

24.1 Symbols are primitives that are also like objects

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.

24.1.1 Symbols are primitive values

Symbols are primitive values:

- They have to be categorized via `typeof`:

  ```
  const sym = Symbol();
  assert.equal(typeof sym, 'symbol');
  ```

- They can be property keys in objects:

```
const obj = {
  [sym]: 123,
};
```

24.1.2   Symbols are also like objects

Even though symbols are primitives, they are also like objects in that each value created by Symbol() is unique and not compared by value:

```
> Symbol() === Symbol()
false
```

Prior to symbols, objects were the best choice if we needed values that were unique (only equal to themselves):

```
const string1 = 'abc';
const string2 = 'abc';
assert.equal(
  string1 === string2, true); // not unique

const object1 = {};
const object2 = {};
assert.equal(
  object1 === object2, false); // unique

const symbol1 = Symbol();
const symbol2 = Symbol();
assert.equal(
  symbol1 === symbol2, false); // unique
```

24.2   The descriptions of symbols

The parameter we pass to the symbol factory function provides a description for the created symbol:

```
const mySymbol = Symbol('mySymbol');
```

The description can be accessed in two ways.

First, it is part of the string returned by .toString():

```
assert.equal(mySymbol.toString(), 'Symbol(mySymbol)');
```

Second, since ES2019, we can retrieve the description via the property .description:

```
assert.equal(mySymbol.description, 'mySymbol');
```

24.3   Use cases for symbols

The main use cases for symbols, are:

- Values for constants

- Unique property keys

24.3.1 Symbols as values for constants

Let's assume you want to create constants representing the colors red, orange, yellow, green, blue, and violet. One simple way of doing so would be to use strings:

```
const COLOR_BLUE = 'Blue';
```

On the plus side, logging that constant produces helpful output. On the minus side, there is a risk of mistaking an unrelated value for a color because two strings with the same content are considered equal:

```
const MOOD_BLUE = 'Blue';
assert.equal(COLOR_BLUE, MOOD_BLUE);
```

We can fix that problem via symbols:

```
const COLOR_BLUE = Symbol('Blue');
const MOOD_BLUE = Symbol('Blue');


assert.notEqual(COLOR_BLUE, MOOD_BLUE);
```

Let's use symbol-valued constants to implement a function:

```
const COLOR_RED    = Symbol('Red');
const COLOR_ORANGE = Symbol('Orange');
const COLOR_YELLOW = Symbol('Yellow');
const COLOR_GREEN  = Symbol('Green');
const COLOR_BLUE   = Symbol('Blue');
const COLOR_VIOLET = Symbol('Violet');

function getComplement(color) {
  switch (color) {
    case COLOR_RED:
      return COLOR_GREEN;
    case COLOR_ORANGE:
      return COLOR_BLUE;
    case COLOR_YELLOW:
      return COLOR_VIOLET;
    case COLOR_GREEN:
      return COLOR_RED;
    case COLOR_BLUE:
      return COLOR_ORANGE;
    case COLOR_VIOLET:
      return COLOR_YELLOW;
    default:
      throw new Exception('Unknown color: '+color);
  }
}
assert.equal(getComplement(COLOR_YELLOW), COLOR_VIOLET);
```

24.3.2 Symbols as unique property keys

The keys of properties (fields) in objects are used at two levels:

- The program operates at a *base level*. The keys at that level reflect the *problem domain* – the area in which a program solves a problem – for example:

  - If a program manages employees, the property keys may be about job titles, salary categories, department IDs, etc.
  - If the program is a chess app, the property keys may be about chess pieces, chess boards, player colors, etc.

- ECMAScript and many libraries operate at a *meta-level*. They manage data and provide services that are not part of the problem domain – for example:

  - The standard method `.toString()` is used by ECMAScript when creating a string representation of an object (line A):

    ```
    const point = {
      x: 7,
      y: 4,
      toString() {
        return `(${this.x}, ${this.y})`;
      },
    };
    assert.equal(
      String(point), '(7, 4)'); // (A)
    ```

    `.x` and `.y` are base-level properties – they are used to solve the problem of computing with points. `.toString()` is a meta-level property – it doesn't have anything to do with the problem domain.

  - The standard ECMAScript method `.toJSON()`

    ```
    const point = {
      x: 7,
      y: 4,
      toJSON() {
        return [this.x, this.y];
      },
    };
    assert.equal(
      JSON.stringify(point), '[7,4]');
    ```

    `.x` and `.y` are base-level properties, `.toJSON()` is a meta-level property.

The base level and the meta-level of a program must be independent: Base-level property keys should not be in conflict with meta-level property keys.

If we use names (strings) as property keys, we are facing two challenges:

- When a language is first created, it can use any meta-level names it wants. Base-level code is forced to avoid those names. Later, however, when much base-level code already exists, meta-level names can't be chosen freely, anymore.

- We could introduce naming rules to separate base level and meta-level. For example, Python brackets meta-level names with two underscores: `__init__`, `__iter__`, `__hash__`, etc. However, the meta-level names of the language and the meta-level names of libraries would still exist in the same namespace and can clash.

These are two examples of where the latter was an issue for JavaScript:

- In May 2018, the Array method `.flatten()` had to be renamed to `.flat()` because the former name was already used by libraries (source).

- In November 2020, the Array method `.item()` had to be renamed to `.at()` because the former name was already used by library (source).

Symbols, used as property keys, help us here: Each symbol is unique and a symbol key never clashes with any other string or symbol key.

Example: a library with a meta-level method

As an example, let's assume we are writing a library that treats objects differently if they implement a special method. This is what defining a property key for such a method and implementing it for an object would look like:

```
const specialMethod = Symbol('specialMethod');
const obj = {
  _id: 'kf12oi',
  [specialMethod]() { // (A)
    return this._id;
  }
};
assert.equal(obj[specialMethod](), 'kf12oi');
```

The square brackets in line A enable us to specify that the method must have the key `specialMethod`. More details are explained in "Computed keys in object literals" (§30.7.2).

24.4 Publicly known symbols

Symbols that play special roles within ECMAScript are called *publicly known symbols*. Examples include:

- `Symbol.iterator`: makes an object *iterable*. It's the key of a method that returns an iterator. For more information on this topic, see "Synchronous iteration".

- `Symbol.hasInstance`: customizes how `instanceof` works. If an object implements a method with that key, it can be used at the right-hand side of that operator. For example:

```
const PrimitiveNull = {
  [Symbol.hasInstance](x) {
    return x === null;
  }
};
assert.equal(null instanceof PrimitiveNull, true);
```

- Symbol.toStringTag: influences the default .toString() method.

  ```
  > String({})
  '[object Object]'
  > String({ [Symbol.toStringTag]: 'is no money' })
  '[object is no money]'
  ```

  Note: It's usually better to override .toString().

🧩 **Exercises: Publicly known symbols**

- Symbol.toStringTag: exercises/symbols/to_string_tag_test.mjs
- Symbol.hasInstance: exercises/symbols/has_instance_test.mjs

---

24.5   Converting symbols

What happens if we convert a symbol sym to another primitive type? Table 24.1 has the answers. One key pitfall with symbols is how often exceptions are thrown when convert-

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

Table 24.1: The results of converting symbols to other primitive types.

ing them to something else. What is the thinking behind that? First, conversion to number never makes sense and should be warned about. Second, converting a symbol to a string is indeed useful for diagnostic output. But it also makes sense to warn about accidentally turning a symbol into a string (which is a different kind of property key):

```
const obj = {};
const sym = Symbol();
assert.throws(
  () => { obj['__'+sym+'__'] = true },
  { message: 'Cannot convert a Symbol value to a string' });
```

The downside is that the exceptions make working with symbols more complicated. You have to explicitly convert symbols when assembling strings via the plus operator:

```
> const mySymbol = Symbol('mySymbol');
> 'Symbol I used: ' + mySymbol
TypeError: Cannot convert a Symbol value to a string
> 'Symbol I used: ' + String(mySymbol)
'Symbol I used: Symbol(mySymbol)'
```

Part V

Control flow and data flow

Chapter 25

Control flow statements

This chapter covers the following control flow statements:

- `if` statement [ES1]
- `switch` statement [ES3]
- `while` loop [ES1]
- `do-while` loop [ES3]
- `for` loop [ES1]
- `for-of` loop [ES6]
- `for-await-of` loop [ES2018]
- `for-in` loop [ES1]

25.1 Controlling loops: `break` and `continue`

The two operators `break` and `continue` can be used to control loops and other statements while we are inside them.

25.1.1 `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:

```js
for (const x of ['a', 'b', 'c']) {
  console.log(x);
  if (x === 'b') break;
  console.log('---')
}
```

Output:

```
a
---
b
```

25.1.2 `break` plus label: leaving any labeled statement

`break` with an operand works everywhere. Its operand is a *label*. Labels can be put in front of any statement, including blocks. `break my_label` leaves the statement whose label is `my_label`:

```js
my_label: { // label
  if (condition) break my_label; // labeled break
  // ···
}
```

In the following example, the search can either:

- Fail: The loop finishes without finding a `result`. That is handled directly after the loop (line B).
- Succeed: While looping, we find a `result`. Then we use `break` plus label (line A) to skip the code that handles failure.

```
function findSuffix(stringArray, suffix) {
  let result;
  search_block: {
    for (const str of stringArray) {
      if (str.endsWith(suffix)) {
        // Success:
        result = str;
        break search_block; // (A)
      }
    } // for
    // Failure:
    result = '(Untitled)'; // (B)
  } // search_block

  return { suffix, result };
    // Same as: {suffix: suffix, result: result}
}
assert.deepEqual(
  findSuffix(['notes.txt', 'index.html'], '.html'),
  { suffix: '.html', result: 'index.html' }
);
assert.deepEqual(
  findSuffix(['notes.txt', 'index.html'], '.mjs'),
  { suffix: '.mjs', result: '(Untitled)' }
);
```

25.1.3 `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
```

25.2 Conditions of control flow statements

`if`, `while`, and `do-while` have conditions that are, in principle, boolean. However, a condition only has to be *truthy* (`true` if coerced to boolean) in order to be accepted. In other words, the following two control flow statements are equivalent:

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

This is a list of all *falsy* values:

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

All other values are truthy. For more information, see "Falsy and truthy values" (§17.2).

25.3 `if` statements [ES1]

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.

25.3.1   The syntax of `if` statements

The general syntax of `if` statements is:

```
if («cond») «then_statement»
else «else_statement»
```

So far, the `then_statement` has always been a block, but we can use any statement. That statement must be terminated with a semicolon:

```
if (true) console.log('Yes'); else console.log('No');
```

That means that `else if` is not its own construct; it's simply an `if` statement whose `else_-statement` is another `if` statement.

25.4   `switch` statements [ES3]

A `switch` statement looks as follows:

```
switch («switch_expression») {
  «switch_body»
}
```

The body of `switch` consists of zero or more case clauses:

```
case «case_expression»:
  «statements»
```

And, optionally, a default clause:

```
default:
  «statements»
```

A `switch` is executed as follows:

- It evaluates the switch expression.
- It jumps to the first case clause whose expression has the same result as the switch expression.
- Otherwise, if there is no such clause, it jumps to the default clause.
- Otherwise, if there is no default clause, it does nothing.

25.4.1   A first example of a `switch` statement

Let's look at an example: The following function converts a number from 1–7 to the name of a weekday.

```
function dayOfTheWeek(num) {
  switch (num) {
    case 1:
      return 'Monday';
    case 2:
      return 'Tuesday';
    case 3:
      return 'Wednesday';
```

```
    case 4:
      return 'Thursday';
    case 5:
      return 'Friday';
    case 6:
      return 'Saturday';
    case 7:
      return 'Sunday';
  }
}
assert.equal(dayOfTheWeek(5), 'Friday');
```

25.4.2 Don't forget to `return` or `break`!

At the end of a case clause, execution continues with the next case clause, unless we `return`
or `break` – for example:

```
function englishToFrench(english) {
  let french;
  switch (english) {
    case 'hello':
      french = 'bonjour';
    case 'goodbye':
      french = 'au revoir';
  }
  return french;
}
// The result should be 'bonjour'!
assert.equal(englishToFrench('hello'), 'au revoir');
```

That is, our implementation of `dayOfTheWeek()` only worked because we used `return`. We
can fix `englishToFrench()` by using `break`:

```
function englishToFrench(english) {
  let french;
  switch (english) {
    case 'hello':
      french = 'bonjour';
      break;
    case 'goodbye':
      french = 'au revoir';
      break;
  }
  return french;
}
assert.equal(englishToFrench('hello'), 'bonjour'); // ok
```

25.4.3   Empty case clauses

The statements of a case clause can be omitted, which effectively gives us multiple case
expressions per case clause:

```
function isWeekDay(name) {
  switch (name) {
    case 'Monday':
    case 'Tuesday':
    case 'Wednesday':
    case 'Thursday':
    case 'Friday':
      return true;
    case 'Saturday':
    case 'Sunday':
      return false;
  }
}
assert.equal(isWeekDay('Wednesday'), true);
assert.equal(isWeekDay('Sunday'), false);
```

25.4.4   Checking for illegal values via a `default` clause

A `default` clause is jumped to if the `switch` expression has no other match. That makes it
useful for error checking:

```
function isWeekDay(name) {
  switch (name) {
    case 'Monday':
    case 'Tuesday':
    case 'Wednesday':
    case 'Thursday':
    case 'Friday':
      return true;
    case 'Saturday':
    case 'Sunday':
      return false;
    default:
      throw new Error('Illegal value: '+name);
  }
}
assert.throws(
  () => isWeekDay('January'),
  {message: 'Illegal value: January'});
```

> ᗌᗌᗌ **Exercises: `switch`**
>
> - exercises/control-flow/number_to_month_test.mjs
> - Bonus: exercises/control-flow/is_object_via_switch_test.mjs

25.5 `while` loops [ES1]

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.

25.5.1 Examples of `while` loops

The following code uses a `while` loop. In each loop iteration, it removes the first element of `arr` via `.shift()` and logs it.

```
const arr = ['a', 'b', 'c'];
while (arr.length > 0) {
  const elem = arr.shift(); // remove first element
  console.log(elem);
}
```

Output:

```
a
b
c
```

If the condition always evaluates to `true`, then `while` is an infinite loop:

```
while (true) {
  if (Math.random() === 0) break;
}
```

25.6 `do-while` loops [ES3]

The `do-while` loop works much like `while`, but it checks its condition *after* each loop iteration, not before.

```
let input;
do {
  input = prompt('Enter text:');
```

```
    console.log(input);
  } while (input !== ':q');
```

do-while can also be viewed as a while loop that runs at least once.

prompt() is a global function that is available in web browsers. It prompts the user to input text and returns it.

25.7 for loops [ES1]

A for loop has the following syntax:

```
for («initialization»; «condition»; «post_iteration») {
  «statements»
}
```

The first line is the *head* of the loop and controls how often the *body* (the remainder of the loop) is executed. It has three parts and each of them is optional:

- initialization: sets up variables, etc. for the loop. Variables declared here via let or const only exist inside the loop.
- condition: This condition is checked before each loop iteration. If it is falsy, the loop stops.
- post_iteration: This code is executed after each loop iteration.

A for loop is therefore roughly equivalent to the following while loop:

```
«initialization»
while («condition») {
  «statements»
  «post_iteration»
}
```

25.7.1 Examples of **for** loops

As an example, this is how to count from zero to two via a for loop:

```
for (let i=0; i<3; i++) {
  console.log(i);
}
```

Output:

```
0
1
2
```

This is how to log the contents of an Array via a for loop:

```
const arr = ['a', 'b', 'c'];
for (let i=0; i<arr.length; i++) {
  console.log(arr[i]);
}
```

Output:

```
a
b
c
```

If we omit all three parts of the head, we get an infinite loop:

```javascript
for (;;) {
  if (Math.random() === 0) break;
}
```

25.8 `for-of` loops [ES6]

A `for-of` loop iterates over any *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:

```javascript
const iterable = ['hello', 'world'];
for (const elem of iterable) {
  console.log(elem);
}
```

Output:

```
hello
world
```

But we can also use a (mutable) variable that already exists:

```javascript
const iterable = ['hello', 'world'];
let elem;
for (elem of iterable) {
  console.log(elem);
}
```

25.8.1 `const`: `for-of` vs. `for`

Note that in `for-of` loops we 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 we must declare variables via `let` or `var` if their values change.

25.8.2 Iterating over iterables

As mentioned before, `for-of` works with any iterable object, not just with Arrays – for example, with Sets:

```
const set = new Set(['hello', 'world']);
for (const elem of set) {
  console.log(elem);
}
```

25.8.3   Iterating over [index, element] pairs of Arrays

Lastly, we can also use for-of to iterate over the [index, element] entries of Arrays:

```
const arr = ['a', 'b', 'c'];
for (const [index, element] of arr.entries()) {
  console.log(`${index} -> ${element}`);
}
```

Output:

```
0 -> a
1 -> b
2 -> c
```

With [index, element], we are using *destructuring* to access Array elements.

⛫ Exercise: `for-of`

exercises/control-flow/array_to_string_test.mjs

25.9   `for-await-of` loops [ES2018]

for-await-of is like for-of, but it works with asynchronous iterables instead of synchronous ones. And it can only be used inside async functions and async generators.

```
for await (const item of asyncIterable) {
  // ···
}
```

for-await-of is described in detail in the chapter on asynchronous iteration.

25.10   `for-in` loops (avoid) [ES1]

The for-in loop visits all (own and inherited) enumerable property keys of an object. When looping over an Array, it is rarely a good choice:

- It visits property keys, not values.
- As property keys, the indices of Array elements are strings, not numbers (more information on how Array elements work).
- It visits all enumerable property keys (both own and inherited ones), not just those of Array elements.

The following code demonstrates these points:

```
const arr = ['a', 'b', 'c'];
arr.propKey = 'property value';

for (const key in arr) {
  console.log(key);
}
```

Output:

```
0
1
2
propKey
```

25.11   Recomendations for looping

- If you want to loop over an [asynchronous iterable](#) (in ES2018+), you must use `for-await-of`.
- For looping over a synchronous iterable (in ES6+), you must use `for-of`. Note that Arrays are iterables.
- For looping over an Array in ES5+, you can use [the Array method `.forEach()`](#).
- Before ES5, you can use a plain `for` loop to loop over an Array.
- Don't use `for-in` to loop over an Array.

Chapter 26

Exception handling

This chapter covers how JavaScript handles exceptions.

> ⚙ **Why doesn't JavaScript throw exceptions more often?**
>
> JavaScript didn't support exceptions until ES3. That explains why they are used sparingly by the language and its standard library.

26.1   Motivation: throwing and catching exceptions

Consider the following code. It reads profiles stored in files into an Array with instances of class `Profile`:

```javascript
function readProfiles(filePaths) {
  const profiles = [];
  for (const filePath of filePaths) {
    try {
      const profile = readOneProfile(filePath);
      profiles.push(profile);
    } catch (err) { // (A)
      console.log('Error in: '+filePath, err);
    }
  }
}
function readOneProfile(filePath) {
  const profile = new Profile();
  const file = openFile(filePath);
  // ··· (Read the data in `file` into `profile`)
  return profile;
}
function openFile(filePath) {
  if (!fs.existsSync(filePath)) {
    throw new Error('Could not find file '+filePath); // (B)
  }
  // ··· (Open the file whose path is `filePath`)
}
```

Let's examine what happens in line B: An error occurred, but the best place to handle the problem is not the current location, it's line A. There, we can skip the current file and move on to the next one.

Therefore:

- In line B, we use a `throw` statement to indicate that there was a problem.
- In line A, we use a `try-catch` statement to handle the problem.

When we throw, the following constructs are active:

```
readProfiles(···)
  for (const filePath of filePaths)
    try
      readOneProfile(···)
        openFile(···)
          if (!fs.existsSync(filePath))
            throw
```

One by one, `throw` exits the nested constructs, until it encounters a `try` statement. Execution continues in the `catch` clause of that `try` statement.

26.2 `throw`

This is the syntax of the `throw` statement:

```
throw «value»;
```

26.2.1 What values should we throw?

Any value can be thrown in JavaScript. However, it's best to use instances of `Error` or a subclass because they support additional features such as stack traces and error chaining (see "Error and its subclasses" (§26.4)).

That leaves us with the following options:

- Using class `Error` directly. That is less limiting in JavaScript than in a more static language because we can add our own properties to instances:

  ```
  const err = new Error('Could not find the file');
  err.filePath = filePath;
  throw err;
  ```

- Using one of the subclasses of `Error`.

- Subclassing `Error` (more details are explained later):

  ```
  class MyError extends Error {
  }
  function func() {
    throw new MyError('Problem!');
  }
  assert.throws(
    () => func(),
    MyError);
  ```

26.3 The `try` statement

The maximal version of the `try` statement looks as follows:

```
try {
  «try_statements»
} catch (error) {
  «catch_statements»
} finally {
  «finally_statements»
}
```

We can combine these clauses as follows:

- try-catch
- try-finally
- try-catch-finally

26.3.1   The `try` block

The `try` block can be considered the body of the statement. This is where we execute the regular code.

26.3.2   The `catch` clause

If an exception reaches the `try` block, then it is assigned to the parameter of the `catch` clause and the code in that clause is executed. Next, execution normally continues after the `try` statement. That may change if:

- There is a `return`, `break`, or `throw` inside the `catch` block.
- There is a `finally` clause (which is always executed before the `try` statement ends).

The following code demonstrates that the value that is thrown in line A is indeed caught in line B.

```
const errorObject = new Error();
function func() {
  throw errorObject; // (A)
}

try {
  func();
} catch (err) { // (B)
  assert.equal(err, errorObject);
}
```

Omitting the `catch` binding [ES2019]

We can omit the `catch` parameter if we are not interested in the value that was thrown:

```
try {
  // ···
} catch {
  // ···
}
```

That may occasionally be useful. For example, Node.js has the API function `assert.throws(func)` that checks whether an error is thrown inside `func`. It could be implemented as follows.

```
function throws(func) {
  try {
    func();
  } catch {
    return; // everything OK
  }
  throw new Error('Function didn't throw an exception!');
}
```

However, a more complete implementation of this function would have a `catch` parameter and would, for example, check that its type is as expected.

26.3.3 The `finally` clause

The code inside the `finally` clause is always executed at the end of a `try` statement – no matter what happens in the `try` block or the `catch` clause.

Let's look at a common use case for `finally`: We have created a resource and want to always destroy it when we are done with it, no matter what happens while working with it. We would implement that as follows:

```
const resource = createResource();
try {
  // Work with `resource`. Errors may be thrown.
} finally {
  resource.destroy();
}
```

`finally` is always executed

The `finally` clause is always executed, even if an error is thrown (line A):

```
let finallyWasExecuted = false;
assert.throws(
  () => {
    try {
      throw new Error(); // (A)
    } finally {
      finallyWasExecuted = true;
    }
  },
  Error
);
assert.equal(finallyWasExecuted, true);
```

And even if there is a `return` statement (line A):

```
let finallyWasExecuted = false;
function func() {
  try {
    return; // (A)
  } finally {
    finallyWasExecuted = true;
  }
}
func();
assert.equal(finallyWasExecuted, true);
```

26.4 Error and its subclasses

`Error` is the common superclass of all built-in error classes.

26.4.1 Class `Error`

This is what `Error`'s instance properties and constructor look like:

```
class Error {
  // Instance properties
  message: string;
  cause?: any; // ES2022
  stack: string; // non-standard but widely supported

  constructor(
    message: string = '',
    options?: ErrorOptions // ES2022
  );
}
interface ErrorOptions {
  cause?: any; // ES2022
}
```

The constructor has two parameters:

- `message` specifies an error message.
- `options` was introduced in ECMAScript 2022. It contains an object where one property is currently supported:
    - `.cause` specifies which exception (if any) caused the current error.

The subsections after the next one explain the instance properties `.message`, `.cause` and `.stack` in more detail.

`Error.prototype.name`

Each built-in error class `E` has a property `E.prototype.name`:

```
> Error.prototype.name
'Error'
> RangeError.prototype.name
'RangeError'
```

Therefore, there are two ways to get the name of the class of a built-in error object:

```
> new RangeError().name
'RangeError'
> new RangeError().constructor.name
'RangeError'
```

Error instance property `.message`

`.message` contains just the error message:

```
const err = new Error('Hello!');
assert.equal(String(err), 'Error: Hello!');
assert.equal(err.message, 'Hello!');
```

If we omit the message then the empty string is used as a default value (inherited from `Error.prototype.message`):

If we omit the message, it is the empty string:

```
assert.equal(new Error().message, '');
```

Error instance property `.stack`

The instance property `.stack` is not an ECMAScript feature, but it is widely supported by JavaScript engines. It is usually a string, but its exact structure is not standardized and varies between engines.

This is what it looks like on the JavaScript engine V8:

```
const err = new Error('Hello!');
assert.equal(
err.stack,
`
Error: Hello!
    at main.mjs:1:13
`.trim());
```

Error instance property `.cause` [ES2022]

The instance property `.cause` is created via the options object in the second parameter of `new Error()`. It specifies which other error caused the current one.

```
const err = new Error('msg', {cause: 'the cause'});
assert.equal(err.cause, 'the cause');
```

For information on how to use this property see "Chaining errors" (§26.5).

26.4.2 The built-in subclasses of `Error`

`Error` has the following subclasses – quoting the ECMAScript specification:

- `AggregateError` [ES2021] represents multiple errors at once. In the standard library, only `Promise.any()` uses it.
- `RangeError` indicates a value that is not in the set or range of allowable values.
- `ReferenceError` indicates 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.

26.4.3 Subclassing `Error`

Since ECMAScript 2022, the `Error` constructor accepts two parameters (see previous subsection). Therefore, we have two choices when subclassing it: We can either omit the constructor in our subclass or we can invoke `super()` like this:

```
class MyCustomError extends Error {
  constructor(message, options) {
    super(message, options);
    // ···
  }
}
```

26.5   Chaining errors

26.5.1   Why would we want to chain errors?

Sometimes, we catch errors that are thrown during a more deeply nested function call and would like to attach more information to it:

```
function readFiles(filePaths) {
  return filePaths.map(
    (filePath) => {
      try {
        const text = readText(filePath);
        const json = JSON.parse(text);
        return processJson(json);
      } catch (error) {
        // (A)
      }
    });
}
```

The statements inside the try clause may throw all kinds of errors. In most cases, an error won't be aware of the path of the file that caused it. That's why we would like to attach that information in line A.

26.5.2   Chaining errors via `error.cause` [ES2022]

Since ECMAScript 2022, new Error() lets us specify what caused it:

```
function readFiles(filePaths) {
  return filePaths.map(
    (filePath) => {
      try {
        // ···
      } catch (error) {
        throw new Error(
          `While processing ${filePath}`,
          {cause: error}
        );
      }
    });
}
```

26.5.3 An alternative to `.cause`: a custom error class

The following custom error class supports chaining. It is forward compatible with `.cause`.

```
/**
 * An error class that supports error chaining.
 * If there is built-in support for .cause, it uses it.
 * Otherwise, it creates this property itself.
 */
class CausedError extends Error {
  constructor(message, options) {
    super(message, options);
    if (
      (isObject(options) && 'cause' in options)
      && !('cause' in this)
    ) {
      // .cause was specified but the superconstructor
      // did not create an instance property.
      const cause = options.cause;
      this.cause = cause;
      if ('stack' in cause) {
        this.stack = this.stack + '\nCAUSE: ' + cause.stack;
      }
    }
  }
}

function isObject(value) {
  return value !== null && typeof value === 'object';
}
```

> **Exercise: Exception handling**
>
> exercises/exception-handling/call_function_test.mjs

Chapter 27

Callable values

In this chapter, we look at JavaScript values that can be invoked: functions, methods, and classes.

27.1 Kinds of functions

JavaScript has two categories of functions:

- An *ordinary function* can play several roles:
    - Real function
    - Method
    - Constructor function

- A *specialized function* can only play one of those roles – for example:
    - An *arrow function* can only be a real function.
    - A *method* can only be a method.
    - A *class* can only be a constructor function.

    Specialized functions were added to the language in ECMAScript 6.

Read on to find out what all of those things mean.

27.2 Ordinary functions

The following code shows two ways of doing (roughly) the same thing: creating an ordinary function.

```
// Function declaration (a statement)
function ordinary1(a, b, c) {
  // ···
}

// const plus anonymous (nameless) function expression
const ordinary2 = function (a, b, c) {
  // ···
};
```

Inside a scope, function declarations are activated early (see "Declarations: scope and activation" (§13.8)) and can be called before they are declared. That is occasionally useful.

Variable declarations, such as the one for `ordinary2`, are not activated early.

27.2.1 Named function expressions (advanced)

So far, we have only seen anonymous function expressions – which don't have names:

```
const anonFuncExpr = function (a, b, c) {
  // ···
};
```

But there are also *named function expressions*:

```
const namedFuncExpr = function myName(a, b, c) {
  // `myName` is only accessible in here
};
```

myName is only accessible inside the body of the function. The function can use it to refer to itself (for self-recursion, etc.) – independently of which variable it is assigned to:

```
const func = function funcExpr() { return funcExpr };
assert.equal(func(), func);

// The name `funcExpr` only exists inside the function body:
assert.throws(() => funcExpr(), ReferenceError);
```

Even if they are not assigned to variables, named function expressions have names (line A):

```
function getNameOfCallback(callback) {
  return callback.name;
}

assert.equal(
  getNameOfCallback(function () {}), ''); // anonymous

assert.equal(
  getNameOfCallback(function named() {}), 'named'); // (A)
```

Note that functions created via function declarations or variable declarations always have names:

```
function funcDecl() {}
assert.equal(
  getNameOfCallback(funcDecl), 'funcDecl');

const funcExpr = function () {};
assert.equal(
  getNameOfCallback(funcExpr), 'funcExpr');
```

One benefit of functions having names is that those names show up in error stack traces.

27.2.2 Terminology: function definitions and function expressions

A *function definition* is syntax that creates functions:

- A function declaration (a statement)
- A function expression

Function declarations always produce ordinary functions. Function expressions produce either ordinary functions or specialized functions:

- Ordinary function expressions (which we have already encountered):
    - Anonymous function expressions
    - Named function expressions

- Specialized function expressions (which we'll look at later):
  - Arrow functions (which are always expressions)

While function declarations are still popular in JavaScript, function expressions are almost always arrow functions in modern code.

27.2.3   Parts of a function declaration

Let's examine the parts of a function declaration via the following example. Most of the terms also apply to function expressions.

```
function add(x, y) {
  return x + y;
}
```

- add is the *name* of the function declaration.
- add(x, y) is the *head* of the function declaration.
- x and y are the *parameters*.
- The curly braces ({ and }) and everything between them are the *body* of the function declaration.
- The return statement explicitly returns a value from the function.

Trailing commas in parameter lists [ES2017]

JavaScript has always allowed and ignored trailing commas in Array literals. Since ES5, they are also allowed in object literals. Since ES2017, we can add trailing commas to parameter lists (declarations and invocations):

```
// Declaration
function retrieveData(
  contentText,
  keyword,
  {unique, ignoreCase, pageSize}, // trailing comma
) {
  // ···
}

// Invocation
retrieveData(
  '',
  null,
  {ignoreCase: true, pageSize: 10}, // trailing comma
);
```

27.2.4   Roles played by ordinary functions

Consider the following function declaration from the previous section:

```
function add(x, y) {
  return x + y;
}
```

This function declaration creates an ordinary function whose name is `add`. As an ordinary function, `add()` can play three roles:

- Real function: invoked via a function call.

  ```
  assert.equal(add(2, 1), 3);
  ```

- Method: stored in a property, invoked via a method call.

  ```
  const obj = { addAsMethod: add };
  assert.equal(obj.addAsMethod(2, 4), 6); // (A)
  ```

  In line A, `obj` is called the *receiver* of the method call.

- Constructor function: invoked via `new`.

  ```
  const inst = new add();
  assert.equal(inst instanceof add, true);
  ```

  As an aside, the names of constructor functions (incl. classes) normally start with capital letters.

27.2.5 Terminology: entity vs. syntax vs. role (advanced)

The distinction between the concepts *syntax*, *entity*, and *role* is subtle and often doesn't matter. But I'd like to sharpen your eye for it:

- An *entity* is a JavaScript feature as it "lives" in RAM. An ordinary function is an entity.

  - Entities include: ordinary functions, arrow functions, methods, and classes.

- *Syntax* is the code that we use to create entities. Function declarations and anonymous function expressions are syntax. They both create entities that are called ordinary functions.

  - Syntax includes: function declarations and anonymous function expressions. The syntax that produces arrow functions is also called *arrow functions*. The same is true for methods and classes.

- A *role* describes how we use entities. The entity *ordinary function* can play the role *real function*, or the role *method*, or the role *class*. The entity *arrow function* can also play the role *real function*.

  - The roles of functions are: real function, method, and constructor function.

Many other programming languages only have a single entity that plays the role *real function*. Then they can use the name *function* for both role and entity.

27.3 Specialized functions [ES6]

Specialized functions are single-purpose versions of ordinary functions. Each one of them specializes in a single role:

- The purpose of an *arrow function* is to be a real function:

```
const arrow = () => {
  return 123;
};
assert.equal(arrow(), 123);
```

- The purpose of a *method* is to be a method:

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

- The purpose of a *class* is to be a constructor function:

```
class MyClass {
  /* ··· */
}
const inst = new MyClass();
```

Apart from nicer syntax, each kind of specialized function also supports new features, making them better at their jobs than ordinary functions.

- Arrow functions are explained soon.
- Methods are explained in the chapter on objects.
- Classes are explained in the chapter on classes.

Table 27.1 lists the capabilities of ordinary and specialized functions.

	Function call	**Method call**	**Constructor call**
Ordinary function	(this === undefined)	✔	✔
Arrow function	✔	(lexical this)	✘
Method	(this === undefined)	✔	✘
Class	✘	✘	✔

Table 27.1: Capabilities of four kinds of functions. If a cell value is in parentheses, that implies some kind of limitation. The special variable `this` is explained in "The special variable `this` in methods, ordinary functions and arrow functions" (§27.3.3).

27.3.1 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
```

27.3.2   Arrow functions

Arrow functions were added to JavaScript for two reasons:

1. To provide a more concise way for creating functions.
2. They work better as real functions inside methods: Methods can refer to the object that received a method call via the special variable `this`. Arrow functions can access the `this` of a surrounding method, ordinary functions can't (because they have their own `this`).

We'll first examine the syntax of arrow functions and then how `this` works in various functions.

The syntax of arrow functions

Let's review the syntax of an anonymous function expression:

```
const f = function (x, y, z) { return 123 };
```

The (roughly) equivalent arrow function looks as follows. Arrow functions are expressions.

```
const f = (x, y, z) => { return 123 };
```

Here, the body of the arrow function is a block. But it can also be an expression. The following arrow function works exactly like the previous one.

```
const f = (x, y, z) => 123;
```

If an arrow function has only a single parameter and that parameter is an identifier (not a destructuring pattern) then you can omit the parentheses around the parameter:

```
const id = x => x;
```

That is convenient when passing arrow functions as parameters to other functions or methods:

```
> [1,2,3].map(x => x+1)
[ 2, 3, 4 ]
```

This previous example demonstrates one benefit of arrow functions – conciseness. If we perform the same task with a function expression, our code is more verbose:

```
[1,2,3].map(function (x) { return x+1 });
```

Syntax pitfall: returning an object literal from an arrow function

If you want the expression body of an arrow function to be an object literal, you must put the literal in parentheses:

```
const func1 = () => ({a: 1});
assert.deepEqual(func1(), { a: 1 });
```

If you don't, JavaScript thinks, the arrow function has a block body (that doesn't return anything):

```
const func2 = () => {a: 1};
assert.deepEqual(func2(), undefined);
```

{a: 1} is interpreted as a block with the label a: and the expression statement 1. Without an explicit return statement, the block body returns undefined.

This pitfall is caused by syntactic ambiguity: object literals and code blocks have the same syntax. We use the parentheses to tell JavaScript that the body is an expression (an object literal) and not a statement (a block).

27.3.3   The special variable `this` in methods, ordinary functions and arrow functions

> 👁 **The special variable `this` is an object-oriented feature**
>
> We are taking a quick look at the special variable `this` here, in order to understand why arrow functions are better real functions than ordinary functions.
> But this feature only matters in object-oriented programming and is covered in more depth in "Methods and the special variable `this`" (§30.5). Therefore, don't worry if you don't fully understand it yet.

Inside methods, the special variable `this` lets us access the *receiver* – the object which received the method call:

```
const obj = {
  myMethod() {
    assert.equal(this, obj);
  }
};
obj.myMethod();
```

Ordinary functions can be methods and therefore also have the implicit parameter `this`:

```
const obj = {
  myMethod: function () {
    assert.equal(this, obj);
  }
};
obj.myMethod();
```

`this` is even an implicit parameter when we use an ordinary function as a real function. Then its value is undefined (if strict mode is active, which it almost always is):

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

That means that an ordinary function, used as a real function, can't access the `this` of a surrounding method (line A). In contrast, arrow functions don't have `this` as an implicit parameter. They treat it like any other variable and can therefore access the `this` of a surrounding method (line B):

```
const jill = {
  name: 'Jill',
  someMethod() {
    function ordinaryFunc() {
      assert.throws(
        () => this.name, // (A)
        /^TypeError: Cannot read properties of undefined \(reading 'name'\)$/);
    }
    ordinaryFunc();

    const arrowFunc = () => {
      assert.equal(this.name, 'Jill'); // (B)
    };
    arrowFunc();
  },
};
jill.someMethod();
```

In this code, we can observe two ways of handling `this`:

- Dynamic `this`: In line A, we try to access the `this` of `.someMethod()` from an ordinary function. There, it is *shadowed* by the function's own `this`, which is `undefined` (as filled in by the function call). Given that ordinary functions receive their `this` via (dynamic) function or method calls, their `this` is called *dynamic*.

- Lexical `this`: In line B, we again try to access the `this` of `.someMethod()`. This time, we succeed because the arrow function does not have its own `this`. `this` is resolved *lexically*, just like any other variable. That's why the `this` of arrow functions is called *lexical*.

27.3.4 Recommendation: prefer specialized functions over ordinary functions

Normally, you should prefer specialized functions over ordinary functions, especially classes and methods.

When it comes to real functions, the choice between an arrow function and an ordinary function is less clear-cut, though:

- For anonymous inline function expressions, arrow functions are clear winners, due to their compact syntax and them not having `this` as an implicit parameter:

  ```
  const twiceOrdinary = [1, 2, 3].map(function (x) {return x * 2});
  const twiceArrow = [1, 2, 3].map(x => x * 2);
  ```

- For stand-alone named function declarations, arrow functions still benefit from lexical `this`. But function declarations (which produce ordinary functions) have nice

syntax and early activation is also occasionally useful (see "Declarations: scope and activation" (§13.8)). If `this` doesn't appear in the body of an ordinary function, there is no downside to using it as a real function. The static checking tool ESLint can warn us during development when we do this wrong via a built-in rule.

```
function timesOrdinary(x, y) {
  return x * y;
}
const timesArrow = (x, y) => {
  return x * y;
};
```

27.4 Summary: kinds of callable values

> **This section refers to upcoming content**
>
> This section mainly serves as a reference for the current and upcoming chapters. Don't worry if you don't understand everything.

So far, all (real) functions and methods, that we have seen, were:

- Single-result
- Synchronous

Later chapters will cover other modes of programming:

- *Iteration* treats objects as containers of data (so-called *iterables*) and provides a standardized way for retrieving what is inside them. If a function or a method returns an iterable, it returns multiple values.
- *Asynchronous programming* deals with handling a long-running computation. You are notified when the computation is finished and can do something else in between. The standard pattern for asynchronously delivering single results is called *Promise*.

These modes can be combined – for example, there are synchronous iterables and asynchronous iterables.

Several new kinds of functions and methods help with some of the mode combinations:

- *Async functions* help implement functions that return Promises. There are also *async methods*.
- *Synchronous generator functions* help implement functions that return synchronous iterables. There are also *synchronous generator methods*.
- *Asynchronous generator functions* help implement functions that return asynchronous iterables. There are also *asynchronous generator methods*.

That leaves us with 4 kinds (2 × 2) of functions and methods:

- Synchronous vs. asynchronous
- Generator vs. single-result

Table 27.2 gives an overview of the syntax for creating these 4 kinds of functions and methods.

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

Table 27.2: Syntax for creating functions and methods. The last column specifies how many values are produced by an entity.

27.5 Returning values from functions and methods

(Everything mentioned in this section applies to both functions and methods.)

The `return` statement explicitly returns a value from a function:

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

Another example:

```
function boolToYesNo(bool) {
  if (bool) {
    return 'Yes';
  } else {
    return 'No';
  }
}
assert.equal(boolToYesNo(true), 'Yes');
assert.equal(boolToYesNo(false), 'No');
```

If, at the end of a function, you haven't returned anything explicitly, JavaScript returns `undefined` for you:

```
function noReturn() {
  // No explicit return
}
assert.equal(noReturn(), undefined);
```

27.6   Parameter handling

Once again, I am only mentioning functions in this section, but everything also applies to methods.

27.6.1   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*.

27.6.2   Terminology: callback

A *callback* or *callback function* is a function that is an argument of a function or method call.

The following is an example of a callback:

```
const myArray = ['a', 'b'];
const callback = (x) => console.log(x);
myArray.forEach(callback);
```

Output:

```
a
b
```

27.6.3   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]);
```

27.6.4   Parameter default values [ES6]

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]);
```

27.6.5   Rest parameters [ES6]

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, []]
);
```

There are two restrictions related to how we can use rest parameters:

- We cannot use more than one rest parameter per function definition.

  ```
  assert.throws(
    () => eval('function f(...x, ...y) {}'),
    /^SyntaxError: Rest parameter must be last formal parameter$/
  );
  ```

- A rest parameter must always come last. As a consequence, we can't access the last
  parameter like this:

```
assert.throws(
  () => eval('function f(...restParams, lastParam) {}'),
  /^SyntaxError: Rest parameter must be last formal parameter$/
);
```

Enforcing a certain number of arguments via a rest parameter

You can use a rest parameter to enforce a certain number of arguments. Take, for example, the following function:

```
function createPoint(x, y) {
  return {x, y};
    // same as {x: x, y: y}
}
```

This is how we force callers to always provide two arguments:

```
function createPoint(...args) {
  if (args.length !== 2) {
    throw new Error('Please provide exactly 2 arguments!');
  }
  const [x, y] = args; // (A)
  return {x, y};
}
```

In line A, we access the elements of args via *destructuring*.

27.6.6 Named parameters

When someone calls a function, the arguments provided by the caller are assigned to the parameters received by the callee. Two common ways of performing the mapping are:

1. Positional parameters: An argument is assigned to a parameter if they have the same position. A function call with only positional arguments looks as follows.

   ```
   selectEntries(3, 20, 2)
   ```

2. Named parameters: An argument is assigned to a parameter if they have the same name. JavaScript doesn't have named parameters, but you can simulate them. For example, this is a function call with only (simulated) named arguments:

   ```
   selectEntries({start: 3, end: 20, step: 2})
   ```

Named parameters have several benefits:

- They lead to more self-explanatory code because each argument has a descriptive label. Just compare the two versions of selectEntries(): with the second one, it is much easier to see what happens.

- The order of the arguments doesn't matter (as long as the names are correct).

- Handling more than one optional parameter is more convenient: callers can easily provide any subset of all optional parameters and don't have to be aware of the

ones they omit (with positional parameters, you have to fill in preceding optional parameters, with `undefined`).

27.6.7 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 read properties of undefined (reading 'start')
```

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 });
```

27.6.8 Spreading (...) into function calls [ES6]

If you put three dots (`...`) in front of the argument of a function call, then you *spread* it. That means that the argument must be an *iterable* object and the iterated values all become arguments. In other words, a single argument is expanded into multiple arguments – for example:

```
function func(x, y) {
  console.log(x);
  console.log(y);
}
const someIterable = ['a', 'b'];
func(...someIterable);
  // same as func('a', 'b')
```

Output:

```
a
b
```

Spreading and rest parameters use the same syntax (`...`), but they serve opposite pur-
poses:

- Rest parameters are used when defining functions or methods. They collect argu-
  ments into Arrays.
- Spread arguments are used when calling functions or methods. They turn iterable
  objects into arguments.

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
```

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

Similarly, the Array method `.push()` destructively adds its zero or more parameters to
the end of its Array. JavaScript has no method for destructively appending an Array to
another one. Once again, we are saved by spreading:

```
const arr1 = ['a', 'b'];
const arr2 = ['c', 'd'];

arr1.push(...arr2);
assert.deepEqual(arr1, ['a', 'b', 'c', 'd']);
```

> **Exercises: Parameter handling**
> - Positional parameters: `exercises/callables/positional_parameters_te
>   st.mjs`
> - Named parameters: `exercises/callables/named_parameters_test.mjs`

27.7 Methods of functions: `.call()`, `.apply()`, `.bind()`

Functions are objects and have methods. In this section, we look at three of those methods:
`.call()`, `.apply()`, and `.bind()`.

27.7.1 The function method `.call()`

Each function `someFunc` has the following method:

```
someFunc.call(thisValue, arg1, arg2, arg3);
```

This method invocation is loosely equivalent to the following function call:

```
someFunc(arg1, arg2, arg3);
```

However, with .call(), we can also specify a value for the implicit parameter this. In other words: .call() makes the implicit parameter this explicit.

The following code demonstrates the use of .call():

```
function func(x, y) {
  return [this, x, y];
}

assert.deepEqual(
  func.call('hello', 'a', 'b'),
  ['hello', 'a', 'b']);
```

As we have seen before, if we function-call an ordinary function, its this is undefined:

```
assert.deepEqual(
  func('a', 'b'),
  [undefined, 'a', 'b']);
```

Therefore, the previous function call is equivalent to:

```
assert.deepEqual(
  func.call(undefined, 'a', 'b'),
  [undefined, 'a', 'b']);
```

In arrow functions, the value for this provided via .call() (or other means) is ignored.

27.7.2 The function method `.apply()`

Each function someFunc has the following method:

```
someFunc.apply(thisValue, [arg1, arg2, arg3]);
```

This method invocation is loosely equivalent to the following function call (which uses spreading):

```
someFunc(...[arg1, arg2, arg3]);
```

However, with .apply(), we can also specify a value for the implicit parameter this.

The following code demonstrates the use of .apply():

```
function func(x, y) {
  return [this, x, y];
}

const args = ['a', 'b'];
assert.deepEqual(
  func.apply('hello', args),
  ['hello', 'a', 'b']);
```

27.7.3   The function method `.bind()`

`.bind()` is another method of function objects. This method is invoked as follows:

```
const boundFunc = someFunc.bind(thisValue, arg1, arg2);
```

`.bind()` returns a new function `boundFunc()`. Calling that function invokes `someFunc()` with `this` set to `thisValue` and these parameters: `arg1`, `arg2`, followed by the parameters of `boundFunc()`.

That is, the following two function calls are equivalent:

```
boundFunc('a', 'b')
someFunc.call(thisValue, arg1, arg2, 'a', 'b')
```

An alternative to `.bind()`

Another way of pre-filling `this` and parameters is via an arrow function:

```
const boundFunc2 = (...args) =>
  someFunc.call(thisValue, arg1, arg2, ...args);
```

An implementation of `.bind()`

Considering the previous section, `.bind()` can be implemented as a real function as follows:

```
function bind(func, thisValue, ...boundArgs) {
  return (...args) =>
    func.call(thisValue, ...boundArgs, ...args);
}
```

Example: binding a real function

Using `.bind()` for real functions is somewhat unintuitive because we have to provide a value for `this`. Given that it is `undefined` during function calls, it is usually set to `undefined` or `null`.

In the following example, we create `add8()`, a function that has one parameter, by binding the first parameter of `add()` to `8`.

```
function add(x, y) {
  return x + y;
}

const add8 = add.bind(undefined, 8);
assert.equal(add8(1), 9);
```

Chapter 28

Evaluating code dynamically: `eval()`, `new Function()` (advanced)

In this chapter, we'll look at two ways of evaluating code dynamically: `eval()` and `new Function()`.

28.1 `eval()`

Given a string `str` with JavaScript code, `eval(str)` evaluates that code and returns the result:

```
> eval('2 ** 4')
16
```

There are two ways of invoking `eval()`:

- *Directly*, via a function call. Then the code in its argument is evaluated inside the current scope.
- *Indirectly*, not via a function call. Then it evaluates its code in global scope.

"Not via a function call" means "anything that looks different than `eval(···)`":

- `eval.call(undefined, '···')` (uses method .call() of functions)
- `eval?.() ()` (uses optional chaining)
- `(0, eval)('···')` (uses the comma operator)
- `globalThis.eval('···')`
- `const e = eval; e('···')`

- Etc.

The following code illustrates the difference:

```
globalThis.myVariable = 'global';
function func() {
  const myVariable = 'local';

  // Direct eval
  assert.equal(eval('myVariable'), 'local');

  // Indirect eval
  assert.equal(eval.call(undefined, 'myVariable'), 'global');
}
```

Evaluating code in global context is safer because the code has access to fewer internals.

28.2 `new Function()`

`new Function()` creates a function object and is invoked as follows:

```
const func = new Function('«param_1»', ···, '«param_n»', '«func_body»');
```

The previous statement is equivalent to the next statement. Note that «param_1», etc., are not inside string literals, anymore.

```
const func = function («param_1», ···, «param_n») {
  «func_body»
};
```

In the next example, we create the same function twice, first via `new Function()`, then via a function expression:

```
const times1 = new Function('a', 'b', 'return a * b');
const times2 = function (a, b) { return a * b };
```

> ⚠️ **`new Function()` creates non-strict mode functions**
>
> By default, functions created via `new Function()` are sloppy. If we want the function body to be in strict mode, we have to switch it on manually.

28.3 Recommendations

Avoid dynamic evaluation of code as much as you can:

- It's a security risk because it may enable an attacker to execute arbitrary code with the privileges of your code.
- It may be switched off – for example, in browsers, via a Content Security Policy.

Very often, JavaScript is dynamic enough so that you don't need eval() or similar. In the following example, what we are doing with eval() (line A) can be achieved just as well without it (line B).

```
const obj = {a: 1, b: 2};
const propKey = 'b';

assert.equal(eval('obj.' + propKey), 2); // (A)
assert.equal(obj[propKey], 2); // (B)
```

If you have to dynamically evaluate code:

- Prefer new Function() over eval(): it always executes its code in global context and a function provides a clean interface to the evaluated code.
- Prefer indirect eval over direct eval: evaluating code in global context is safer.

Part VI

Modularity

Chapter 29

Modules [ES6]

29.1 Cheat sheet: modules

29.1.1 Named exports, named imports, namespace imports

If we put `export` in front of a named entity inside a module, it becomes a *named export* of that module. All other entities are private to the module.

```
//===== lib1.mjs =====
// Named exports
export const one = 1, two = 2;
export function myFunc() {
  return 3;
}

//===== main1a.mjs =====
// Named imports
import {one, myFunc as f} from './lib1.mjs';
assert.equal(one, 1);
assert.equal(f(), 3);


// Namespace import
import * as lib1 from './lib1.mjs';
assert.equal(lib1.one, 1);
assert.equal(lib1.myFunc(), 3);
```

The string after `from` is called a *module specifier*. It identifies from which module we want to import.

29.1.2 Dynamic imports

So far, all imports we have seen were *static*, with the following constraints:

- They have to appear at the top level of a module.
- The module specifier is fixed.

Dynamic imports via `import()` [ES2020] don't have those constraints:

```
//===== main1b.mjs =====
function importLibrary(moduleSpecifier) {
  return import(moduleSpecifier)
  .then((lib1) => {
    assert.equal(lib1.one, 1);
    assert.equal(lib1.myFunc(), 3);
  });
}
await importLibrary('./lib1.mjs');
```

29.1.3 Default exports and imports

A *default export* is mainly used when a module only contains a single entity (even though it can be combined with named exports).

```
//===== lib2a.mjs =====
export default function getHello() {
  return 'hello';
}
```

A default export is the exception to the rule that function declarations always have names: In the previous example, we can omit the name `getHello`.

```
//===== lib2b.mjs =====
export default 123; // (A) instead of `const`
```

There can be at most one default export. That's why `const` or `let` can't be default-exported (line A).

```
//===== main2.mjs =====
import lib2a from './lib2a.mjs';
assert.equal(lib2a(), 'hello');

import lib2b from './lib2b.mjs';
assert.equal(lib2b, 123);
```

29.1.4 Kinds of module specifiers

There are three kinds of module specifiers:

- *Absolute specifiers* are full URLs – for example:

```
'https://www.unpkg.com/browse/yargs@17.3.1/browser.mjs'
'file:///opt/nodejs/config.mjs'
```

Absolute specifiers are mostly used to access libraries that are directly hosted on the web.

- *Relative specifiers* are relative URLs (starting with `'/'`, `'./'` or `'../'`) – for example:

  ```
  './sibling-module.js'
  '../module-in-parent-dir.mjs'
  '../../dir/other-module.js'
  ```

  Relative specifiers are mostly used to access other modules within the same code base.

- *Bare specifiers* are paths (without protocol and domain) that start with neither slashes nor dots. They begin with the names of *packages* (as installed via a package manager such *npm*). Those names can optionally be followed by *subpaths*:

  ```
  'some-package'
  'some-package/sync'
  'some-package/util/files/path-tools.js'
  ```

  Bare specifiers can also refer to packages with scoped names:

  ```
  '@some-scope/scoped-name'
  '@some-scope/scoped-name/async'
  '@some-scope/scoped-name/dir/some-module.mjs'
  ```

  Each bare specifier refers to exactly one module inside a package; if it has no subpath, it refers to the designated "main" module of its package.

29.2   JavaScript source code formats

The current landscape of JavaScript modules is quite diverse: ES6 brought built-in modules, but the source code formats that came before them, are still around, too. Understanding the latter helps understand the former, so let's investigate. The next sections describe the following ways of delivering JavaScript source code:

- *Scripts* are code fragments that browsers run in global scope. They are precursors of modules.
- *CommonJS modules* are a module format that is mainly used on servers (e.g., via Node.js).
- *AMD modules* are a module format that is mainly used in browsers.
- *ECMAScript modules* are JavaScript's built-in module format. It supersedes all previous formats.

Table 29.1 gives an overview of these code formats. Note that for CommonJS modules and ECMAScript modules, two filename extensions are commonly used. Which one is appropriate depends on how we want to use a file. Details are given later in this chapter.

29.2.1   Code before built-in modules was written in ECMAScript 5

Before we get to built-in modules (which were introduced with ES6), all code that we'll see, will be written in ES5. Among other things:

	Runs on	**Loaded**	**Filename ext.**
Script	browsers	async	`.js`
CommonJS module	servers	sync	`.js .cjs`
AMD module	browsers	async	`.js`
ECMAScript module	browsers and servers	async	`.js .mjs`

Table 29.1: Ways of delivering JavaScript source code.

- ES5 did not have `const` and `let`, only `var`.
- ES5 did not have arrow functions, only function expressions.

29.3 Before we had modules, we had scripts

Initially, browsers only had *scripts* – pieces of code that were executed in global scope. As an example, consider an HTML file that loads script files via the following HTML:

```html
<script src="other-module1.js"></script>
<script src="other-module2.js"></script>
<script src="my-module.js"></script>
```

The main file is `my-module.js`, where we simulate a module:

```js
var myModule = (function () { // Open IIFE
  // Imports (via global variables)
  var importedFunc1 = otherModule1.importedFunc1;
  var importedFunc2 = otherModule2.importedFunc2;

  // Body
  function internalFunc() {
    // ···
  }
  function exportedFunc() {
    importedFunc1();
    importedFunc2();
    internalFunc();
  }

  // Exports (assigned to global variable `myModule`)
  return {
    exportedFunc: exportedFunc,
  };
})(); // Close IIFE
```

`myModule` is a global variable that is assigned the result of immediately invoking a function expression. The function expression starts in the first line. It is invoked in the last line.

This way of wrapping a code fragment is called *immediately invoked function expression* (IIFE, coined by Ben Alman). What do we gain from an IIFE? `var` is not block-scoped (like `const`

and `let`), it is function-scoped: the only way to create new scopes for `var`-declared variables is via functions or methods (with `const` and `let`, we can use either functions, methods, or blocks `{}`). Therefore, the IIFE in the example hides all of the following variables from global scope and minimizes name clashes: `importedFunc1`, `importedFunc2`, `internalFunc`, `exportedFunc`.

Note that we are using an IIFE in a particular manner: at the end, we pick what we want to export and return it via an object literal. That is called the *revealing module pattern* (coined by Christian Heilmann).

This way of simulating modules, has several issues:

- Libraries in script files export and import functionality via global variables, which risks name clashes.
- Dependencies are not stated explicitly, and there is no built-in way for a script to load the scripts it depends on. Therefore, the web page has to load not just the scripts that are needed by the page but also the dependencies of those scripts, the dependencies' dependencies, etc. And it has to do so in the right order!

29.4   Module systems created prior to ES6

Prior to ECMAScript 6, JavaScript did not have built-in modules. Therefore, the flexible syntax of the language was used to implement custom module systems *within* the language. Two popular ones are:

- CommonJS (targeting the server side)
- AMD (Asynchronous Module Definition, targeting the client side)

29.4.1   Server side: CommonJS modules

The original CommonJS standard for modules was created for server and desktop platforms. It was the foundation of the original Node.js module system, where it achieved enormous popularity. Contributing to that popularity were the npm package manager for Node and tools that enabled using Node modules on the client side (browserify, webpack, and others).

From now on, *CommonJS module* means the Node.js version of this standard (which has a few additional features). This is an example of a CommonJS module:

```
// Imports
var importedFunc1 = require('./other-module1.js').importedFunc1;
var importedFunc2 = require('./other-module2.js').importedFunc2;

// Body
function internalFunc() {
  // ···
}
function exportedFunc() {
  importedFunc1();
  importedFunc2();
  internalFunc();
```

```
}

// Exports
module.exports = {
  exportedFunc: exportedFunc,
};
```

CommonJS can be characterized as follows:

- Designed for servers.
- Modules are meant to be loaded *synchronously* (the importer waits while the imported module is loaded and executed).
- Compact syntax.

29.4.2   Client side: AMD (Asynchronous Module Definition) modules

The AMD module format was created to be easier to use in browsers than the CommonJS format. Its most popular implementation is RequireJS. The following is an example of an AMD module.

```
define(['./other-module1.js', './other-module2.js'],
  function (otherModule1, otherModule2) {
    var importedFunc1 = otherModule1.importedFunc1;
    var importedFunc2 = otherModule2.importedFunc2;

    function internalFunc() {
      // ···
    }
    function exportedFunc() {
      importedFunc1();
      importedFunc2();
      internalFunc();
    }

    return {
      exportedFunc: exportedFunc,
    };
  });
```

AMD can be characterized as follows:

- Designed for browsers.
- Modules are meant to be loaded *asynchronously*. That's a crucial requirement for browsers, where code can't wait until a module has finished downloading. It has to be notified once the module is available.
- The syntax is slightly more complicated.

On the plus side, AMD modules can be executed directly. In contrast, CommonJS modules must either be compiled before deployment or custom source code must be generated and evaluated dynamically (think eval()). That isn't always permitted on the web.

29.4.3   Characteristics of JavaScript modules

Looking at CommonJS and AMD, similarities between JavaScript module systems emerge:

- There is one module per file.
- Such a file is basically a piece of code that is executed:
  - Local scope: The code is executed in a local "module scope". Therefore, by default, all of the variables, functions, and classes declared in it are internal and not global.
  - Exports: If we want any declared entity to be exported, we must explicitly mark it as an export.
  - Imports: Each module can import exported entities from other modules. Those other modules are identified via *module specifiers* (usually paths, occasionally full URLs).
- Modules are *singletons*: Even if a module is imported multiple times, only a single "instance" of it exists.
- No global variables are used. Instead, module specifiers serve as global IDs.

29.5   ECMAScript modules

ECMAScript modules (*ES modules* or *ESM*) were introduced with ES6. They continue the tradition of JavaScript modules and have all of their aforementioned characteristics. Additionally:

- With CommonJS, ES modules share the compact syntax and support for cyclic dependencies.
- With AMD, ES modules share being designed for asynchronous loading.

ES modules also have new benefits:

- The syntax is even more compact than CommonJS's.
- Modules have *static* structures (which can't be changed at runtime). That helps with static checking, optimized access of imports, dead code elimination, and more.
- Support for cyclic imports is completely transparent.

This is an example of ES module syntax:

```
import {importedFunc1} from './other-module1.mjs';
import {importedFunc2} from './other-module2.mjs';

function internalFunc() {
  ...
}

export function exportedFunc() {
  importedFunc1();
  importedFunc2();
  internalFunc();
}
```

From now on, "module" means "ECMAScript module".

29.5.1 ES modules: syntax, semantics, loader API

The full standard of ES modules comprises the following parts:

1. Syntax (how code is written): What is a module? How are imports and exports declared? Etc.
2. Semantics (how code is executed): How are variable bindings exported? How are imports connected with exports? Etc.
3. A programmatic loader API for configuring module loading.

Parts 1 and 2 were introduced with ES6. Work on part 3 is ongoing.

29.6 Named exports and imports

29.6.1 Named exports

Each module can have zero or more *named exports*.

As an example, consider the following two files:

```
lib/my-math.mjs
main.mjs
```

Module `my-math.mjs` has two named exports: `square` and `LIGHTSPEED`.

```
// Not exported, private to module
function times(a, b) {
  return a * b;
}
export function square(x) {
  return times(x, x);
}
export const LIGHTSPEED = 299792458;
```

To export something, we put the keyword `export` in front of a declaration. Entities that are not exported are private to a module and can't be accessed from outside.

29.6.2 Named imports

Module `main.mjs` has a single named import, `square`:

```
import {square} from './lib/my-math.mjs';
assert.equal(square(3), 9);
```

It can also rename its import:

```
import {square as sq} from './lib/my-math.mjs';
assert.equal(sq(3), 9);
```

Syntactic pitfall: named importing is not destructuring

Both named importing and destructuring look similar:

```
import {foo} from './bar.mjs'; // import
const {foo} = require('./bar.mjs'); // destructuring
```

But they are quite different:

- Imports remain connected with their exports.

- We can destructure again inside a destructuring pattern, but the {} in an import statement can't be nested.

- The syntax for renaming is different:

  ```
  import {foo as f} from './bar.mjs'; // importing
  const {foo: f} = require('./bar.mjs'); // destructuring
  ```

  Rationale: Destructuring is reminiscent of an object literal (including nesting), while importing evokes the idea of renaming.

Exercise: Named exports

`exercises/modules/export_named_test.mjs`

29.6.3   Namespace imports

Namespace imports are an alternative to named imports. If we namespace-import a module, it becomes an object whose properties are the named exports. This is what `main.mjs` looks like if we use a namespace import:

```
import * as myMath from './lib/my-math.mjs';
assert.equal(myMath.square(3), 9);

assert.deepEqual(
  Object.keys(myMath), ['LIGHTSPEED', 'square']);
```

29.6.4   Named exporting styles: inline versus clause (advanced)

The named export style we have seen so far was *inline*: We exported entities by prefixing them with the keyword `export`.

But we can also use separate *export clauses*. For example, this is what `lib/my-math.mjs` looks like with an export clause:

```
function times(a, b) {
  return a * b;
}
function square(x) {
  return times(x, x);
}
const LIGHTSPEED = 299792458;

export { square, LIGHTSPEED }; // semicolon!
```

With an export clause, we can rename before exporting and use different names internally:

```
function times(a, b) {
  return a * b;
}
function sq(x) {
  return times(x, x);
}
const LS = 299792458;

export {
  sq as square,
  LS as LIGHTSPEED, // trailing comma is optional
};
```

29.7   Default exports and imports

Each module can have at most one *default export*. The idea is that the module *is* the default-exported value.

> 💡 **Avoid mixing named exports and default exports**
>
> A module can have both named exports and a default export, but it's usually better to stick to one export style per module.

As an example for default exports, consider the following two files:

```
my-func.mjs
main.mjs
```

Module `my-func.mjs` has a default export:

```
const GREETING = 'Hello!';
export default function () {
  return GREETING;
}
```

Module `main.mjs` default-imports the exported function:

```
import myFunc from './my-func.mjs';
assert.equal(myFunc(), 'Hello!');
```

Note the syntactic difference: the curly braces around named imports indicate that we are reaching *into* the module, while a default import *is* the module.

> **?** **What are use cases for default exports?**
>
> The most common use case for a default export is a module that contains a single function or a single class.

29.7.1   The two styles of default-exporting

There are two styles of doing default exports.

First, we can label existing declarations with `export default`:

```
export default function myFunc() {} // no semicolon!
export default class MyClass {} // no semicolon!
```

Second, we can directly default-export values. This style of `export default` is much like a declaration.

```
export default myFunc; // defined elsewhere
export default MyClass; // defined previously
export default Math.sqrt(2); // result of invocation is default-exported
export default 'abc' + 'def';
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. Consider the following hypothetical code:

```
// Not legal JavaScript!
export default const foo = 1, bar = 2, baz = 3;
```

With this code, we don't know which one of the three values is the default export.

> **⧉** **Exercise: Default exports**
>
> `exercises/modules/export_default_test.mjs`

29.7.2   The default export as a named export (advanced)

Internally, a default export is simply a named export whose name is `default`. As an example, consider the previous module `my-func.mjs` with a default export:

```
const GREETING = 'Hello!';
export default function () {
  return GREETING;
}
```

The following module `my-func2.mjs` is equivalent to that module:

```
const GREETING = 'Hello!';
function greet() {
  return GREETING;
}

export {
  greet as default,
};
```

For importing, we can use a normal default import:

```
import myFunc from './my-func2.mjs';
assert.equal(myFunc(), 'Hello!');
```

Or we can use a named import:

```
import {default as myFunc} from './my-func2.mjs';
assert.equal(myFunc(), 'Hello!');
```

The default export is also available via property `.default` of namespace imports:

```
import * as mf from './my-func2.mjs';
assert.equal(mf.default(), 'Hello!');
```

> (?) **Isn't default illegal as a variable name?**
>
> `default` can't be a variable name, but it can be an export name and it can be a property name:
> ```
> const obj = {
>   default: 123,
> };
> assert.equal(obj.default, 123);
> ```

29.8   Re-exporting

A module `library.mjs` can export one or more exports of another module `internal.mjs` as if it had made them itself. That is called *re-exporting*.

```
//===== internal.mjs =====
export function internalFunc() {}
export const INTERNAL_DEF = 'hello';
export default 123;

//===== library.mjs =====
// Named re-export [ES6]
export {internalFunc as func, INTERNAL_DEF as DEF} from './internal.mjs';
// Wildcard re-export [ES6]
export * from './internal.mjs';
// Namespace re-export [ES2020]
export * as ns from './internal.mjs';
```

- The wildcard re-export turns all exports of module `internal.mjs` into exports of `library.mjs`, except the default export.
- The namespace re-export turns all exports of module `internal.mjs` into an object that becomes the named export `ns` of `library.mjs`. Because `internal.mjs` has a default export, `ns` has a property `.default`.

The following code demonstrates the two bullet points above:

```
//===== main.mjs =====
import * as library from './library.mjs';

assert.deepEqual(
  Object.keys(library),
  ['DEF', 'INTERNAL_DEF', 'func', 'internalFunc', 'ns']
);
assert.deepEqual(
  Object.keys(library.ns),
  ['INTERNAL_DEF', 'default', 'internalFunc']
);
```

29.9   More details on exporting and importing

29.9.1   Imports are read-only views on exports

So far, we have used imports and exports intuitively, and everything seems to have worked as expected. But now it is time to take a closer look at how imports and exports are really related.

Consider the following two modules:

```
counter.mjs
main.mjs
```

`counter.mjs` exports a (mutable!) variable and a function:

```
export let counter = 3;
export function incCounter() {
  counter++;
}
```

`main.mjs` name-imports both exports. When we use `incCounter()`, we discover that the connection to `counter` is live – we can always access the live state of that variable:

```
import { counter, incCounter } from './counter.mjs';

// The imported value `counter` is live
assert.equal(counter, 3);
incCounter();
assert.equal(counter, 4);
```

Note that while the connection is live and we can read `counter`, we cannot change this variable (e.g., via `counter++`).

There are two benefits to handling imports this way:

- It is easier to split modules because previously shared variables can become exports.
- This behavior is crucial for supporting transparent cyclic imports. Read on for more information.

29.9.2 ESM's transparent support for cyclic imports (advanced)

ESM supports cyclic imports transparently. To understand how that is achieved, consider the following example: figure 29.1 shows a directed graph of modules importing other modules. P importing M is the cycle in this case. After parsing, these modules are set up
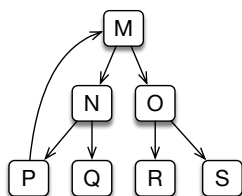
Figure 29.1: A directed graph of modules importing modules: M imports N and O, N imports P and Q, etc.

in two phases:

- Instantiation: Every module is visited and its imports are connected to its exports. Before a parent can be instantiated, all of its children must be instantiated.
- Evaluation: The bodies of the modules are executed. Once again, children are evaluated before parents.

This approach handles cyclic imports correctly, due to two features of ES modules:

- Due to the static structure of ES modules, the exports are already known after parsing. That makes it possible to instantiate P before its child M: P can already look up M's exports.

- When P is evaluated, M hasn't been evaluated, yet. However, entities in P can already mention imports from M. They just can't use them, yet, because the imported values are filled in later. For example, a function in P can access an import from M. The only limitation is that we must wait until after the evaluation of M, before calling that function.

  Imports being filled in later is enabled by them being "live immutable views" on exports.

29.10   npm packages

The *npm software registry* is the dominant way of distributing JavaScript libraries and apps for Node.js and web browsers. It is managed via the *npm package manager* (short: *npm*). Software is distributed as so-called *packages*. A package is a directory containing arbitrary files and a file `package.json` at the top level that describes the package. For example, when npm creates an empty package inside a directory `my-package/`, we get this `package.json`:

```
{
  "name": "my-package",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Some of these properties contain simple metadata:

- `name` specifies the name of this package. Once it is uploaded to the npm registry, it can be installed via `npm install my-package`.
- `version` is used for version management and follows semantic versioning, with three numbers:
  - Major version: is incremented when incompatible API changes are made.
  - Minor version: is incremented when functionality is added in a backward compatible manner.
  - Patch version: is incremented when backward compatible changes are made.
- `description`, `keywords`, `author` make it easier to find packages.
- `license` clarifies how we can use this package.

Other properties enable advanced configuration:

- `main`: specifies the module that "is" the package (explained later in this chapter).
- `scripts`: are commands that we can execute via `npm run`. For example, the script `test` can be executed via `npm run test`.

For more information on `package.json`, consult the npm documentation.

29.10.1 Packages are installed inside a directory `node_modules/`

npm always installs packages inside a directory `node_modules`. There are usually many of these directories. Which one npm uses, depends on the directory where one currently is. For example, if we are inside a directory `/tmp/a/b/`, npm tries to find a `node_modules` in the current directory, its parent directory, the parent directory of the parent, etc. In other words, it searches the following *chain* of locations:

- `/tmp/a/b/node_modules`
- `/tmp/a/node_modules`
- `/tmp/node_modules`

When installing a package `some-pkg`, npm uses the closest `node_modules`. If, for example, we are inside `/tmp/a/b/` and there is a `node_modules` in that directory, then npm puts the package inside the directory:

```
/tmp/a/b/node_modules/some-pkg/
```

When importing a module, we can use a special module specifier to tell Node.js that we want to import it from an installed package. How exactly that works, is explained later. For now, consider the following example:

```
// /home/jane/proj/main.mjs
import * as theModule from 'the-package/the-module.mjs';
```

To find `the-module.mjs` (Node.js prefers the filename extension `.mjs` for ES modules), Node.js walks up the `node_module` chain and searches the following locations:

- /home/jane/proj/node_modules/the-package/the-module.mjs
- /home/jane/node_modules/the-package/the-module.mjs
- /home/node_modules/the-package/the-module.mjs

29.10.2 Why can npm be used to install frontend libraries?

Finding installed modules in `node_modules` directories is only supported on Node.js. So why can we also use npm to install libraries for browsers?

That is enabled via bundling tools, such as webpack, that compile and optimize code before it is deployed online. During this compilation process, the code in npm packages is adapted so that it works in browsers.

29.11 Naming modules

There are no established best practices for naming module files and the variables they are imported into.

In this chapter, I'm using the following naming style:

- The names of module files are dash-cased and start with lowercase letters:

    ```
    ./my-module.mjs
    ./some-func.mjs
    ```

- The names of namespace imports are lowercased and camel-cased:

    ```
    import * as myModule from './my-module.mjs';
    ```

- The names of default imports are lowercased and camel-cased:

    ```
    import someFunc from './some-func.mjs';
    ```

What are the rationales behind this style?

- npm doesn't allow uppercase letters in package names (source). Thus, we avoid camel case, so that "local" files have names that are consistent with those of npm packages. Using only lowercase letters also minimizes conflicts between file systems that are case-sensitive and file systems that aren't: the former distinguish files whose names have the same letters, but with different cases; the latter don't.

- There are clear rules for translating dash-cased file names to camel-cased JavaScript variable names. Due to how we name namespace imports, these rules work for both namespace imports and default imports.

I also like underscore-cased module file names because we can directly use these names for namespace imports (without any translation):

```
import * as my_module from './my_module.mjs';
```

But that style does not work for default imports: I like underscore-casing for namespace objects, but it is not a good choice for functions, etc.

29.12   Module specifiers

Module specifiers are the strings that identify modules. They work slightly differently in browsers and Node.js. Before we can look at the differences, we need to learn about the different categories of module specifiers.

29.12.1   Categories of module specifiers

In ES modules, we distinguish the following categories of specifiers. These categories originated with CommonJS modules.

- Relative path: starts with a dot. Examples:

  ```
  './some/other/module.mjs'
  '../../lib/counter.mjs'
  ```

- Absolute path: starts with a slash. Example:

  ```
  '/home/jane/file-tools.mjs'
  ```

- URL: includes a protocol (technically, paths are URLs, too). Examples:

  ```
  'https://example.com/some-module.mjs'
  'file:///home/john/tmp/main.mjs'
  ```

- Bare path: does not start with a dot, a slash or a protocol, and consists of a single filename without an extension. Examples:

  ```
  'lodash'
  'the-package'
  ```

- Deep import path: starts with a bare path and has at least one slash. Example:

  ```
  'the-package/dist/the-module.mjs'
  ```

29.12.2   ES module specifiers in browsers

Browsers handle module specifiers as follows:

- Relative paths, absolute paths, and URLs work as expected. They all must point to real files (in contrast to CommonJS, which lets us omit filename extensions and more).
- The file name extensions of modules don't matter, as long as they are served with the content type `text/javascript`.
- How bare paths will end up being handled is not yet clear. We will probably eventually be able to map them to other specifiers via lookup tables.

Note that bundling tools such as webpack, which combine modules into fewer files, are often less strict with specifiers than browsers. That's because they operate at build/compile time (not at runtime) and can search for files by traversing the file system.

29.12.3 ES module specifiers on Node.js

Node.js handles module specifiers as follows:

- Relative paths are resolved as they are in web browsers – relative to the path of the current module.

- Absolute paths are currently not supported. As a workaround, we can use URLs that start with `file:///`. We can create such URLs via `url.pathToFileURL()`.

- Only `file:` is supported as a protocol for URL specifiers.

- A bare path is interpreted as a package name and resolved relative to the closest `node_modules` directory. What module should be loaded, is determined by looking at property `"main"` of the package's `package.json` (similarly to CommonJS).

- Deep import paths are also resolved relatively to the closest `node_modules` directory. They contain file names, so it is always clear which module is meant.

All specifiers, except bare paths, must refer to actual files. That is, ESM does not support the following CommonJS features:

- CommonJS automatically adds missing filename extensions.

- CommonJS can import a directory `dir` if there is a `dir/package.json` with a `"main"` property.

- CommonJS can import a directory `dir` if there is a module `dir/index.js`.

All built-in Node.js modules are available via bare paths and have named ESM exports – for example:

```
import assert from 'node:assert/strict';
import * as path from 'node:path';

assert.equal(
  path.join('a/b/c', '../d'), 'a/b/d');
```

Filename extensions on Node.js

Node.js supports the following default filename extensions:

- `.mjs` for ES modules
- `.cjs` for CommonJS modules

The filename extension `.js` stands for either ESM or CommonJS. Which one it is is configured via the "closest" `package.json` (in the current directory, the parent directory, etc.). Using `package.json` in this manner is independent of packages.

In that `package.json`, there is a property `"type"`, which has two settings:

- "commonjs" (the default): files with the extension .js or without an extension are interpreted as CommonJS modules.

- "module": files with the extension .js or without an extension are interpreted as ESM modules.

Interpreting non-file source code as either CommonJS or ESM

Not all source code executed by Node.js comes from files. We can also send it code via stdin, --eval, and --print. The command line option --input-type lets us specify how such code is interpreted:

- As CommonJS (the default): --input-type=commonjs
- As ESM: --input-type=module

29.13 `import.meta` – metadata for the current module [ES2020]

The object import.meta holds metadata for the current module.

29.13.1 `import.meta.url`

The most important property of import.meta is .url which contains a string with the URL of the current module's file – for example:

```
'https://example.com/code/main.mjs'
```

29.13.2 `import.meta.url` and class `URL`

Class URL is available via a global variable in browsers and on Node.js. We can look up its full functionality in the Node.js documentation. When working with import.meta.url, its constructor is especially useful:

```
new URL(input: string, base?: string|URL)
```

Parameter input contains the URL to be parsed. It can be relative if the second parameter, base, is provided.

In other words, this constructor lets us resolve a relative path against a base URL:

```
> new URL('other.mjs', 'https://example.com/code/main.mjs').href
'https://example.com/code/other.mjs'
> new URL('../other.mjs', 'https://example.com/code/main.mjs').href
'https://example.com/other.mjs'
```

This is how we get a URL instance that points to a file data.txt that sits next to the current module:

```
const urlOfData = new URL('data.txt', import.meta.url);
```

29.13.3 `import.meta.url` on Node.js

On Node.js, import.meta.url is always a string with a file: URL – for example:

```
'file:///Users/rauschma/my-module.mjs'
```

Example: reading a sibling file of a module

Many Node.js file system operations accept either strings with paths or instances of URL. That enables us to read a sibling file `data.txt` of the current module:

```
import * as fs from 'node:fs';
function readData() {
  // data.txt sits next to current module
  const urlOfData = new URL('data.txt', import.meta.url);
  return fs.readFileSync(urlOfData, {encoding: 'UTF-8'});
}
```

Module `fs` and URLs

For most functions of the module `fs`, we can refer to files via:

- Paths – in strings or instances of `Buffer`.
- URLs – in instances of URL (with the protocol `file:`)

For more information on this topic, see the Node.js API documentation.

Converting between `file:` URLs and paths

The Node.js module `url` has two functions for converting between `file:` URLs and paths:

- `fileURLToPath(url: URL|string): string`
  Converts a `file:` URL to a path.
- `pathToFileURL(path: string): URL`
  Converts a path to a `file:` URL.

If we need a path that can be used in the local file system, then property `.pathname` of URL instances does not always work:

```
assert.equal(
  new URL('file:///tmp/with%20space.txt').pathname,
  '/tmp/with%20space.txt');
```

Therefore, it is better to use `fileURLToPath()`:

```
import * as url from 'node:url';
assert.equal(
  url.fileURLToPath('file:///tmp/with%20space.txt'),
  '/tmp/with space.txt'); // result on Unix
```

Similarly, `pathToFileURL()` does more than just prepend `'file://'` to an absolute path.

29.14 Loading modules dynamically via `import()` [ES2020] (advanced)

> 👁 **The `import()` operator uses Promises**
>
> Promises are a technique for handling results that are computed asynchronously (i.e., not immediately). They are explained in "Promises for asynchronous programming". It may make sense to postpone reading this section until you understand them.

29.14.1 The limitations of static `import` statements

So far, the only way to import a module has been via an `import` statement. That statement has several limitations:

- We must use it at the top level of a module. That is, we can't, for example, import something when we are inside a function or inside an `if` statement.
- The module specifier is always fixed. That is, we can't change what we import depending on a condition. And we can't assemble a specifier dynamically.

29.14.2 Dynamic imports via the `import()` operator

The `import()` operator doesn't have the limitations of `import` statements. It looks like this:

```
import(moduleSpecifierStr)
.then((namespaceObject) => {
  console.log(namespaceObject.namedExport);
});
```

This operator is used like a function, receives a string with a module specifier and returns a Promise that resolves to a namespace object. The properties of that object are the exports of the imported module.

`import()` is even more convenient to use via `await`:

```
const namespaceObject = await import(moduleSpecifierStr);
console.log(namespaceObject.namedExport);
```

Note that `await` can be used at the top levels of modules (see next section).

Let's look at an example of using `import()`.

Example: loading a module dynamically

Consider the following files:

```
lib/my-math.mjs
main1.mjs
main2.mjs
```

We have already seen module `my-math.mjs`:

```
// Not exported, private to module
function times(a, b) {
  return a * b;
}
export function square(x) {
  return times(x, x);
}
export const LIGHTSPEED = 299792458;
```

We can use `import()` to load this module on demand:

```
// main1.mjs
const moduleSpecifier = './lib/my-math.mjs';

function mathOnDemand() {
  return import(moduleSpecifier)
  .then(myMath => {
    const result = myMath.LIGHTSPEED;
    assert.equal(result, 299792458);
    return result;
  });
}

await mathOnDemand()
.then((result) => {
  assert.equal(result, 299792458);
});
```

Two things in this code can't be done with `import` statements:

- We are importing inside a function (not at the top level).
- The module specifier comes from a variable.

Next, we'll implement the same functionality as in `main1.mjs` but via a feature called *async function* or *async/await* which provides nicer syntax for Promises.

```
// main2.mjs
const moduleSpecifier = './lib/my-math.mjs';

async function mathOnDemand() {
  const myMath = await import(moduleSpecifier);
  const result = myMath.LIGHTSPEED;
  assert.equal(result, 299792458);
  return result;
}
```

> ⑦ **Why is `import()` an operator and not a function?**
>
> `import()` looks like a function but couldn't be implemented as a function:
> - It needs to know the URL of the current module in order to resolve relative module specifiers.
> - If `import()` were a function, we'd have to explicitly pass this information to it (e.g. via an parameter).
> - In contrast, an operator is a core language construct and has implicit access to more data, including the URL of the current module.

29.14.3   Use cases for `import()`

Loading code on demand

Some functionality of web apps doesn't have to be present when they start, it can be loaded on demand. Then `import()` helps because we can put such functionality into modules – for example:

```
button.addEventListener('click', event => {
  import('./dialogBox.mjs')
    .then(dialogBox => {
      dialogBox.open();
    })
    .catch(error => {
      /* Error handling */
    })
});
```

Conditional loading of modules

We may want to load a module depending on whether a condition is true. For example, a module with a polyfill that makes a new feature available on legacy platforms:

```
if (isLegacyPlatform()) {
  import('./my-polyfill.mjs')
    .then(···);
}
```

Computed module specifiers

For applications such as internationalization, it helps if we can dynamically compute module specifiers:

```
import(`messages_${getLocale()}.mjs`)
  .then(···);
```

29.15   Top-level **await** in modules [ES2022] (advanced)

> 👁 **await is a feature of async functions**
>
> await is explained in "Async functions". It may make sense to postpone reading this section until you understand async functions.

We can use the await operator at the top level of a module. If we do that, the module becomes asynchronous and works differently. Thankfully, we don't usually see that as programmers because it is handled transparently by the language.

29.15.1   Use cases for top-level **await**

Why would we want to use the await operator at the top level of a module? It lets us initialize a module with asynchronously loaded data. The next three subsections show three examples of where that is useful.

Loading modules dynamically

```
const params = new URLSearchParams(location.search);
const language = params.get('lang');
const messages = await import(`./messages-${language}.mjs`); // (A)

console.log(messages.welcome);
```

In line A, we dynamically import a module. Thanks to top-level await, that is almost as convenient as using a normal, static import.

Using a fallback if module loading fails

```
let mylib;
try {
  mylib = await import('https://primary.example.com/mylib');
} catch {
  mylib = await import('https://secondary.example.com/mylib');
}
```

Using whichever resource loads fastest

```
const resource = await Promise.any([
  fetch('http://example.com/first.txt')
    .then(response => response.text()),
  fetch('http://example.com/second.txt')
    .then(response => response.text()),
]);
```

Due to Promise.any(), variable resource is initialized via whichever download finishes first.

29.15.2   How does top-level `await` work under the hood?

Consider the following two files.

`first.mjs`:

```
const response = await fetch('http://example.com/first.txt');
export const first = await response.text();
```

`main.mjs`:

```
import {first} from './first.mjs';
import {second} from './second.mjs';
assert.equal(first, 'First!');
assert.equal(second, 'Second!');
```

Both are roughly equivalent to the following code: `first.mjs`:

```
export let first;
export const promise = (async () => { // (A)
  const response = await fetch('http://example.com/first.txt');
  first = await response.text();
})();
```

`main.mjs`:

```
import {promise as firstPromise, first} from './first.mjs';
import {promise as secondPromise, second} from './second.mjs';
export const promise = (async () => { // (B)
  await Promise.all([firstPromise, secondPromise]); // (C)
  assert.equal(first, 'First!');
  assert.equal(second, 'Second!');
})();
```

A module becomes asynchronous if:

1. It directly uses top-level `await` (`first.mjs`).
2. It imports one or more asynchronous modules (`main.mjs`).

Each asynchronous module exports a Promise (line A and line B) that is fulfilled after its body was executed. At that point, it is safe to access the exports of that module.

In case (2), the importing module waits until the Promises of all imported asynchronous modules are fulfilled, before it enters its body (line C). Synchronous modules are handled as usually.

Awaited rejections and synchronous exceptions are managed as in async functions.

29.15.3   The pros and cons of top-level `await`

The two most important benefits of top-level `await` are:

- It ensures that modules don't access asynchronous imports before they are fully initialized.
- It handles asynchronicity transparently: Importers do not need to know if an imported module is asynchronous or not.

On the downside, top-level `await` delays the initialization of importing modules. Therefore, it's best used sparingly. Asynchronous tasks that take longer are better performed later, on demand.

However, even modules without top-level `await` can block importers (e.g. via an infinite loop at the top level), so blocking per se is not an argument against it.

29.16   Polyfills: emulating native web platform features (advanced)

> ⚙️ **Backends have polyfills, too**
>
> This section is about frontend development and web browsers, but similar ideas apply to backend development.

Polyfills help with a conflict that we are facing when developing a web application in JavaScript:

- On one hand, we want to use modern web platform features that make the app better and/or development easier.
- On the other hand, the app should run on as many browsers as possible.

Given a web platform feature X:

- A *polyfill* for X is a piece of code. If it is executed on a platform that already has built-in support for X, it does nothing. Otherwise, it makes the feature available on the platform. In the latter case, the polyfilled feature is (mostly) indistinguishable from a native implementation. In order to achieve that, the polyfill usually makes global changes. For example, it may modify global data or configure a global module loader. Polyfills are often packaged as modules.

    - The term *polyfill* was coined by Remy Sharp.

- A *speculative polyfill* is a polyfill for a proposed web platform feature (that is not standardized, yet).

    - Alternative term: *prollyfill*

- A *replica* of X is a library that reproduces the API and functionality of X locally. Such a library exists independently of a native (and global) implementation of X.

    - *Replica* is a new term introduced in this section. Alternative term: *ponyfill*

- There is also the term *shim*, but it doesn't have a universally agreed upon definition. It often means roughly the same as *polyfill*.

Every time our web applications starts, it must first execute all polyfills for features that may not be available everywhere. Afterwards, we can be sure that those features are available natively.

29.16.1 Sources of this section

- "What is a Polyfill?" by Remy Sharp
- Inspiration for the term *replica*: The Eiffel Tower in Las Vegas
- Useful clarification of "polyfill" and related terms: "Polyfills and the evolution of the Web". Edited by Andrew Betts.

Chapter 30

Objects

In this book, JavaScript's style of object-oriented programming (OOP) is introduced in four steps. This chapter covers step 1 and 2; the next chapter covers step 3 and 4. The steps are (figure 30.1):

1. **Single objects (this chapter):** How do *objects*, JavaScript's basic OOP building blocks, work in isolation?
2. **Prototype chains (this chapter):** Each object has a chain of zero or more *prototype objects*. Prototypes are JavaScript's core inheritance mechanism.
3. **Classes (next chapter):** JavaScript's *classes* are factories for objects. The relationship between a class and its instances is based on prototypal inheritance (step 2).
4. **Subclassing (next chapter):** The relationship between a *subclass* and its *superclass* is

also based on prototypal inheritance.
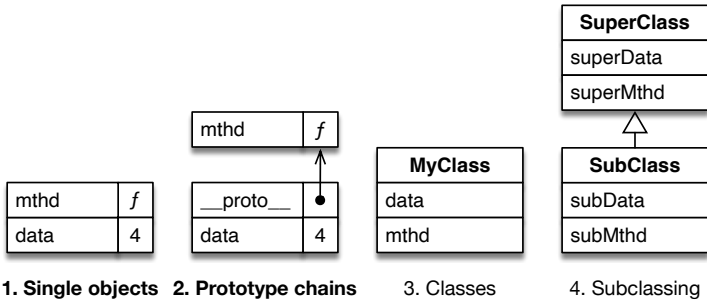
Figure 30.1: This book introduces object-oriented programming in JavaScript in four steps.

30.1 Cheat sheet: objects

30.1.1 Cheat sheet: single objects

Creating an object via an *object literal* (starts and ends with a curly brace):

```
const myObject = { // object literal
  myProperty: 1,
  myMethod() {
    return 2;
  }, // comma!
  get myAccessor() {
    return this.myProperty;
  }, // comma!
  set myAccessor(value) {
    this.myProperty = value;
  }, // last comma is optional
};

assert.equal(
  myObject.myProperty, 1
);
assert.equal(
  myObject.myMethod(), 2
);
assert.equal(
  myObject.myAccessor, 1
);
myObject.myAccessor = 3;
assert.equal(
  myObject.myProperty, 3
);
```

Being able to create objects directly (without classes) is one of the highlights of JavaScript.

Spreading into objects:

```
const original = {
  a: 1,
  b: {
    c: 3,
  },
};

// Spreading (...) copies one object "into" another one:
const modifiedCopy = {
  ...original, // spreading
  d: 4,
};

assert.deepEqual(
  modifiedCopy,
  {
    a: 1,
    b: {
      c: 3,
    },
    d: 4,
  }
);

// Caveat: spreading copies shallowly (property values are shared)
modifiedCopy.a = 5; // does not affect `original`
modifiedCopy.b.c = 6; // affects `original`
assert.deepEqual(
  original,
  {
    a: 1, // unchanged
    b: {
      c: 6, // changed
    },
  },
);
```

We can also use spreading to make an unmodified (shallow) copy of an object:

```
const exactCopy = {...obj};
```

30.1.2   Cheat sheet: prototype chains

Prototypes are JavaScript's fundamental inheritance mechanism. Even classes are based on it. Each object has `null` or an object as its prototype. The latter object can also have a prototype, etc. In general, we get *chains* of prototypes.

Prototypes are managed like this:

```
// `obj1` has no prototype (its prototype is `null`)
const obj1 = Object.create(null); // (A)
assert.equal(
  Object.getPrototypeOf(obj1), null // (B)
);

// `obj2` has the prototype `proto`
const proto = {
  protoProp: 'protoProp',
};
const obj2 = {
  __proto__: proto, // (C)
  objProp: 'objProp',
}
assert.equal(
  Object.getPrototypeOf(obj2), proto
);
```

Notes:

- Setting an object's prototype while creating the object: line A, line C
- Retrieving the prototype of an object: line B

Each object inherits all the properties of its prototype:

```
// `obj2` inherits .protoProp from `proto`
assert.equal(
  obj2.protoProp, 'protoProp'
);
assert.deepEqual(
  Reflect.ownKeys(obj2),
  ['objProp'] // own properties of `obj2`
);
```

The non-inherited properties of an object are called its *own* properties.

The most important use case for prototypes is that several objects can share methods by inheriting them from a common prototype.

30.2   What is an object?

Objects in JavaScript:

- An object is a set of *slots* (key-value entries).
- Public slots are called *properties*:
    - A property key can only be a string or a symbol.
- Private slots can only be created via classes and are explained in "Public slots (properties) vs. private slots" (§31.2.4).

30.2.1 The two ways of using objects

There are two ways of using objects in JavaScript:

- Fixed-layout objects: Used this way, objects work like records in databases. They have a fixed number of properties, whose keys are known at development time. Their values generally have different types.

  ```
  const fixedLayoutObject = {
    product: 'carrot',
    quantity: 4,
  };
  ```

- Dictionary objects: Used this way, objects work like lookup tables or maps. They have a variable number of properties, whose keys are not known at development time. All of their values have the same type.

  ```
  const dictionaryObject = {
    ['one']: 1,
    ['two']: 2,
  };
  ```

Note that the two ways can also be mixed: Some objects are both fixed-layout objects and dictionary objects.

The ways of using objects influence how they are explained in this chapter:

- First, we'll explore fixed-layout objects. Even though property keys are strings or symbols under the hood, they will appear as fixed identifiers to us.
- Later, we'll explore dictionary objects. Note that Maps are usually better dictionaries than objects. However, some of the operations that we'll encounter are also useful for fixed-layout objects.

30.3 Fixed-layout objects

Let's first explore *fixed-layout objects*.

30.3.1 Object literals: properties

Object literals are one way of creating fixed-layout objects. They are a stand-out feature of JavaScript: we can directly create objects – no need for classes! This is an example:

```
const jane = {
  first: 'Jane',
  last: 'Doe', // optional trailing comma
};
```

In the example, we created an object via an object literal, which starts and ends with curly braces {}. Inside it, we defined two *properties* (key-value entries):

- The first property has the key first and the value 'Jane'.
- The second property has the key last and the value 'Doe'.

Since ES5, trailing commas are allowed in object literals.

We will later see other ways of specifying property keys, but with this way of specifying them, they must follow the rules of JavaScript variable names. For example, we can use `first_name` as a property key, but not `first-name`). However, reserved words are allowed:

```
const obj = {
  if: true,
  const: true,
};
```

In order to check the effects of various operations on objects, we'll occasionally use `Object.keys()` in this part of the chapter. It lists property keys:

```
> Object.keys({a:1, b:2})
[ 'a', 'b' ]
```

30.3.2 Object literals: property value shorthands

Whenever the value of a property is defined via a variable that has the same name as the key, we can omit the key.

```
function createPoint(x, y) {
  return {x, y}; // Same as: {x: x, y: y}
}
assert.deepEqual(
  createPoint(9, 2),
  { x: 9, y: 2 }
);
```

30.3.3 Getting properties

This is how we *get* (read) a property (line A):

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

// Get property .first
assert.equal(jane.first, 'Jane'); // (A)
```

Getting an unknown property produces `undefined`:

```
assert.equal(jane.unknownProperty, undefined);
```

30.3.4 Setting properties

This is how we *set* (write to) a property (line A):

```
const obj = {
  prop: 1,
};
```

```
assert.equal(obj.prop, 1);
obj.prop = 2; // (A)
assert.equal(obj.prop, 2);
```

We just changed an existing property via setting. If we set an unknown property, we create a new entry:

```
const obj = {}; // empty object
assert.deepEqual(
  Object.keys(obj), []);

obj.unknownProperty = 'abc';
assert.deepEqual(
  Object.keys(obj), ['unknownProperty']);
```

30.3.5   Object literals: methods

The following code shows how to create the method `.says()` via an object literal:

```
const jane = {
  first: 'Jane', // value 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` (more on `this` in "Methods and the special variable `this`" (§30.5)). That enables method `.says()` to access the sibling property `.first` in line A.

30.3.6   Object literals: accessors

An *accessor* is defined via syntax inside an object literal that looks like methods: a *getter* and/or a *setter* (i.e., each accessor has one or both of them).

Invoking an accessor looks like accessing a value property:

- Reading the property invokes the getter.
- Writing to the property invokes the setter.

Getters

A getter is created by prefixing a method definition with the modifier `get`:

```
const jane = {
  first: 'Jane',
  last: 'Doe',
  get full() {
    return `${this.first} ${this.last}`;
  },
```

```
};

assert.equal(jane.full, 'Jane Doe');
jane.first = 'John';
assert.equal(jane.full, 'John Doe');
```

Setters

A setter is created by prefixing a method definition with the modifier `set`:

```
const jane = {
  first: 'Jane',
  last: 'Doe',
  set full(fullName) {
    const parts = fullName.split(' ');
    this.first = parts[0];
    this.last = parts[1];
  },
};

jane.full = 'Richard Roe';
assert.equal(jane.first, 'Richard');
assert.equal(jane.last, 'Roe');
```

Exercise: Creating an object via an object literal

`exercises/objects/color_point_object_test.mjs`

30.4 Spreading into object literals (. . .) [ES2018]

Inside an object literal, a *spread property* adds the properties of another object to the current one:

```
> const obj = {one: 1, two: 2};
> {...obj, three: 3}
{ one: 1, two: 2, three: 3 }

const obj1 = {one: 1, two: 2};
const obj2 = {three: 3};
assert.deepEqual(
  {...obj1, ...obj2, four: 4},
  {one: 1, two: 2, three: 3,  four: 4}
);
```

If property keys clash, the property that is mentioned last "wins":

```
> const obj = {one: 1, two: 2, three: 3};
> {...obj, one: true}
{ one: true, two: 2, three: 3 }
```

```
> {one: true, ...obj}
{ one: 1, two: 2, three: 3 }
```

All values are spreadable, even `undefined` and `null`:

```
> {...undefined}
{}
> {...null}
{}
> {...123}
{}
> {...'abc'}
{ '0': 'a', '1': 'b', '2': 'c' }
> {...['a', 'b']}
{ '0': 'a', '1': 'b' }
```

Property `.length` of strings and Arrays is hidden from this kind of operation (it is not *enumerable*; see "Property attributes and property descriptors" (§30.8) for more information).

Spreading includes properties whose keys are symbols (which are ignored by `Object.keys()`, `Object.values()` and `Object.entries()`):

```
const symbolKey = Symbol('symbolKey');
const obj = {
  stringKey: 1,
  [symbolKey]: 2,
};
assert.deepEqual(
  {...obj, anotherStringKey: 3},
  {
    stringKey: 1,
    [symbolKey]: 2,
    anotherStringKey: 3,
  }
);
```

30.4.1   Use case for spreading: copying objects

We can use spreading to create a copy of an object `original`:

```
const copy = {...original};
```

Caveat – copying is *shallow*: `copy` is a fresh object with duplicates of all properties (key-value entries) of `original`. But if property values are objects, then those are not copied themselves; they are shared between `original` and `copy`. Let's look at an example:

```
const original = { a: 1, b: {prop: true} };
const copy = {...original};
```

The first level of `copy` is really a copy: If we change any properties at that level, it does not affect the original:

```
copy.a = 2;
assert.deepEqual(
  original, { a: 1, b: {prop: true} }); // no change
```

However, deeper levels are not copied. For example, the value of `.b` is shared between original and copy. Changing `.b` in the copy also changes it in the original.

```
copy.b.prop = false;
assert.deepEqual(
  original, { a: 1, b: {prop: false} });
```

> ⚠ **JavaScript doesn't have built-in support for deep copying**
>
> JavaScript does not have a built-in operation for deeply copying objects. Options:
> - Implement it ourselves.
> - The global function `structuredClone()` is supported by most JavaScript platforms now – even though it is not part of ECMAScript. Alas, this function has a number of limitations – e.g., if we copy an instance of a class we created, the copy is not an instance of that class.
> - The Lodash library has the functions `_.cloneDeep()` and `_.cloneDeepWith()` that can help. They have fewer limitations than `structuredClone()`.

30.4.2 Use case for spreading: default values for missing properties

If one of the inputs of our code is an object with data, we can make properties optional by specifying default values that are used if those properties are missing. One technique for doing so is via an object whose properties contain the default values. In the following example, that object is `DEFAULTS`:

```
const DEFAULTS = {alpha: 'a', beta: 'b'};
const providedData = {alpha: 1};

const allData = {...DEFAULTS, ...providedData};
assert.deepEqual(allData, {alpha: 1, beta: 'b'});
```

The result, the object `allData`, is created by copying `DEFAULTS` and overriding its properties with those of `providedData`.

But we don't need an object to specify the default values; we can also specify them inside the object literal, individually:

```
const providedData = {alpha: 1};

const allData = {alpha: 'a', beta: 'b', ...providedData};
assert.deepEqual(allData, {alpha: 1, beta: 'b'});
```

30.4.3 Use case for spreading: non-destructively changing properties

So far, we have encountered one way of changing a property `.alpha` of an object: We *set* it (line A) and mutate the object. That is, this way of changing a property is destructive.

```
const obj = {alpha: 'a', beta: 'b'};
obj.alpha = 1; // (A)
assert.deepEqual(obj, {alpha: 1, beta: 'b'});
```

With spreading, we can change `.alpha` non-destructively – we make a copy of `obj` where `.alpha` has a different value:

```
const obj = {alpha: 'a', beta: 'b'};
const updatedObj = {...obj, alpha: 1};
assert.deepEqual(updatedObj, {alpha: 1, beta: 'b'});
```

🧩 **Exercise: Non-destructively updating a property via spreading (fixed key)**

`exercises/objects/update_name_test.mjs`

30.4.4   "Destructive spreading": `Object.assign()` [ES6]

`Object.assign()` is a tool method:

```
Object.assign(target, source_1, source_2, ···)
```

This expression assigns all properties of `source_1` to `target`, then all properties of `source_-2`, etc. At the end, it returns `target` – for example:

```
const target = { a: 1 };

const result = Object.assign(
  target,
  {b: 2},
  {c: 3, b: true});

assert.deepEqual(
  result, { a: 1, b: true, c: 3 });
// target was modified and returned:
assert.equal(result, target);
```

The use cases for `Object.assign()` are similar to those for spread properties. In a way, it spreads destructively.

30.5   Methods and the special variable `this`

30.5.1   Methods are properties whose values are functions

Let's revisit the example that was used to introduce methods:

```
const jane = {
  first: 'Jane',
  says(text) {
    return `${this.first} says "${text}"`;
```

```
  },
};
```

Somewhat surprisingly, methods are functions:

```
assert.equal(typeof jane.says, 'function');
```

Why is that? We learned in the chapter on callable values that ordinary functions play several roles. *Method* is one of those roles. Therefore, internally, `jane` roughly looks as follows.

```
const jane = {
  first: 'Jane',
  says: function (text) {
    return `${this.first} says "${text}"`;
  },
};
```

30.5.2 The special variable `this`

Consider the following code:

```
const obj = {
  someMethod(x, y) {
    assert.equal(this, obj); // (A)
    assert.equal(x, 'a');
    assert.equal(y, 'b');
  }
};
obj.someMethod('a', 'b'); // (B)
```

In line B, `obj` is the *receiver* of a method call. It is passed to the function stored in `obj.someMethod` via an implicit (hidden) parameter whose name is `this` (line A).

> 🔆 **How to understand `this`**
>
> The best way to understand `this` is as an implicit parameter of ordinary functions (and therefore methods, too).

30.5.3 Methods and `.call()`

Methods are functions and functions have methods themselves. One of those methods is `.call()`. Let's look at an example to understand how this method works.

In the previous section, there was this method invocation:

```
obj.someMethod('a', 'b')
```

This invocation is equivalent to:

```
obj.someMethod.call(obj, 'a', 'b');
```

Which is also equivalent to:

```
const func = obj.someMethod;
func.call(obj, 'a', 'b');
```

`.call()` makes the normally implicit parameter `this` explicit: When invoking a function via `.call()`, the first parameter is `this`, followed by the regular (explicit) function parameters.

As an aside, this means that there are actually two different dot operators:

1. One for accessing properties: `obj.prop`
2. Another one for calling methods: `obj.prop()`

They are different in that (2) is not just (1) followed by the function call operator (). Instead, (2) additionally provides a value for `this`.

30.5.4   Methods and `.bind()`

`.bind()` is another method of function objects. In the following code, we use `.bind()` to 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. In the next section, we'll explore why that is.

30.5.5   `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 we 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 properties of undefined (reading 'first')",
  });
```

In line A, we are making a normal function call. And in normal function calls, `this` is `undefined` (if strict mode is active, which it almost always is). Line A is therefore equivalent to:

```
assert.throws(
  () => jane.says.call(undefined, 'hello'), // `this` is undefined!
  {
    name: 'TypeError',
    message: "Cannot read properties of undefined (reading 'first')",
  }
);
```

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()`.

We can also use arrow functions to extract methods:

```
const func3 = text => jane.says(text);
assert.equal(func3('hello'), 'Jane says "hello"');
```

Example: extracting a method

The following is a simplified version of code that we may see in actual web development:

```
class ClickHandler {
  constructor(id, elem) {
    this.id = id;
    elem.addEventListener('click', this.handleClick); // (A)
  }
  handleClick(event) {
    alert('Clicked ' + this.id);
  }
}
```

In line A, we don't extract the method `.handleClick()` properly. Instead, we should do:

```
const listener = this.handleClick.bind(this);
elem.addEventListener('click', listener);

// Later, possibly:
elem.removeEventListener('click', listener);
```

Each invocation of `.bind()` creates a new function. That's why we need to store the result somewhere if we want to remove it later on.

How to avoid the pitfall of extracting methods

Alas, there is no simple way around the pitfall of extracting methods: Whenever we extract a method, we have to be careful and do it properly – for example, by binding `this` or by using an arrow function.

> 🧩 **Exercise: Extracting a method**
>
> `exercises/objects/method_extraction_exrc.mjs`

30.5.6 `this` pitfall: accidentally shadowing `this`

> 💡 **Accidentally shadowing `this` is only an issue with ordinary functions**
>
> Arrow functions don't shadow `this`.

Consider the following problem: when we are inside an ordinary function, we can't access the `this` of the surrounding scope because the ordinary function has its own `this`. In other words, a variable in an inner scope hides a variable in an outer scope. That is called *shadowing*. The following code is an example:

```
const prefixer = {
  prefix: '==> ',
  prefixStringArray(stringArray) {
    return stringArray.map(
      function (x) {
        return this.prefix + x; // (A)
      });
  },
};
assert.throws(
  () => prefixer.prefixStringArray(['a', 'b']),
  {
    name: 'TypeError',
    message: "Cannot read properties of undefined (reading 'prefix')",
  }
);
```

In line A, we want to access the `this` of `.prefixStringArray()`. But we can't since the surrounding ordinary function has its own `this` that *shadows* (and blocks access to) the `this` of the method. The value of the former `this` is `undefined` due to the callback being function-called. That explains the error message.

The simplest way to fix this problem is via an arrow function, which doesn't have its own `this` and therefore doesn't shadow anything:

```
const prefixer = {
  prefix: '==> ',
```

```
    prefixStringArray(stringArray) {
      return stringArray.map(
        (x) => {
          return this.prefix + x;
        });
    },
};
assert.deepEqual(
  prefixer.prefixStringArray(['a', 'b']),
  ['==> a', '==> b']);
```

We can also store `this` in a different variable (line A), so that it doesn't get shadowed:

```
  prefixStringArray(stringArray) {
    const that = this; // (A)
    return stringArray.map(
      function (x) {
        return that.prefix + x;
      });
  },
```

Another option is to specify a fixed `this` for the callback via `.bind()` (line A):

```
  prefixStringArray(stringArray) {
    return stringArray.map(
      function (x) {
        return this.prefix + x;
      }.bind(this)); // (A)
  },
```

Lastly, `.map()` lets us specify a value for `this` (line A) that it uses when invoking the callback:

```
  prefixStringArray(stringArray) {
    return stringArray.map(
      function (x) {
        return this.prefix + x;
      },
      this); // (A)
  },
```

Avoiding the pitfall of accidentally shadowing `this`

If we follow the advice in "Recommendation: prefer specialized functions over ordinary functions" (§27.3.4), we can avoid the pitfall of accidentally shadowing `this`. This is a summary:

- Use arrow functions as anonymous inline functions. They don't have `this` as an implicit parameter and don't shadow it.

- For named stand-alone function declarations we can either use arrow functions or function declarations. If we do the latter, we have to make sure `this` isn't mentioned

in their bodies.

30.5.7   The value of `this` in various contexts (advanced)

What is the value of `this` in various contexts?

Inside a callable entity, the value of `this` depends on how the callable entity is invoked and what kind of callable entity it is:

- Function call:
  - Ordinary functions: `this === undefined` (in strict mode)
  - Arrow functions: `this` is same as in surrounding scope (lexical `this`)
- Method call: `this` is receiver of call
- `new`: `this` refers to the newly created instance

We can also access `this` in all common top-level scopes:

- `<script>` element: `this === globalThis`
- ECMAScript modules: `this === undefined`
- CommonJS modules: `this === module.exports`

> ## 💡 Tip: pretend that `this` doesn't exist in top-level scopes
>
> I like to do that because top-level `this` is confusing and there are better alternatives for its (few) use cases.

30.6   Optional chaining for property getting and method calls [ES2020] (advanced)

The following kinds of optional chaining operations exist:

```
obj?.prop      // optional fixed property getting
obj?.[«expr»]  // optional dynamic property getting
func?.(«arg0», «arg1») // optional function or method call
```

The rough idea is:

- If the value before the question mark is neither `undefined` nor `null`, then perform the operation after the question mark.
- Otherwise, return `undefined`.

Each of the three syntaxes is covered in more detail later. These are a few first examples:

```
> null?.prop
undefined
> {prop: 1}?.prop
1

> null?.(123)
undefined
```

```
> String?.(123)
'123'
```

> 💡 **Mnemonic for the optional chaining operator (?.)**
>
> Are you occasionally unsure if the optional chaining operator starts with a dot (.?)
> or a question mark (?.)? Then this mnemonic may help you:
>   - **If** (?) the left-hand side is not nullish
>   - **then** (.) access a property.

30.6.1   Example: optional fixed property getting

Consider the following data:

```
const persons = [
  {
    surname: 'Zoe',
    address: {
      street: {
        name: 'Sesame Street',
        number: '123',
      },
    },
  },
  {
    surname: 'Mariner',
  },
  {
    surname: 'Carmen',
    address: {
    },
  },
];
```

We can use optional chaining to safely extract street names:

```
const streetNames = persons.map(
  p => p.address?.street?.name);
assert.deepEqual(
  streetNames, ['Sesame Street', undefined, undefined]
);
```

Handling defaults via nullish coalescing

The nullish coalescing operator allows us to use the default value '(no name)' instead of
undefined:

```
const streetNames = persons.map(
  p => p.address?.street?.name ?? '(no name)');
```

```
assert.deepEqual(
  streetNames, ['Sesame Street', '(no name)', '(no name)']
);
```

30.6.2   The operators in more detail (advanced)

Optional fixed property getting

The following two expressions are equivalent:

```
o?.prop
(o !== undefined && o !== null) ? o.prop : undefined
```

Examples:

```
assert.equal(undefined?.prop, undefined);
assert.equal(null?.prop,      undefined);
assert.equal({prop:1}?.prop,  1);
```

Optional dynamic property getting

The following two expressions are equivalent:

```
o?.[«expr»]
(o !== undefined && o !== null) ? o[«expr»] : undefined
```

Examples:

```
const key = 'prop';
assert.equal(undefined?.[key], undefined);
assert.equal(null?.[key], undefined);
assert.equal({prop:1}?.[key], 1);
```

Optional function or method call

The following two expressions are equivalent:

```
f?.(arg0, arg1)
(f !== undefined && f !== null) ? f(arg0, arg1) : undefined
```

Examples:

```
assert.equal(undefined?.(123), undefined);
assert.equal(null?.(123), undefined);
assert.equal(String?.(123), '123');
```

Note that this operator produces an error if its left-hand side is not callable:

```
assert.throws(
  () => true?.(123),
  TypeError);
```

Why? The idea is that the operator only tolerates deliberate omissions. An uncallable value (other than `undefined` and `null`) is probably an error and should be reported, rather than worked around.

30.6.3   Short-circuiting with optional property getting

In a chain of property gettings and method invocations, evaluation stops once the first optional operator encounters `undefined` or `null` at its left-hand side:

```js
function invokeM(value) {
  return value?.a.b.m(); // (A)
}

const obj = {
  a: {
    b: {
      m() { return 'result' }
    }
  }
};
assert.equal(
  invokeM(obj), 'result'
);
assert.equal(
  invokeM(undefined), undefined // (B)
);
```

Consider `invokeM(undefined)` in line B: `undefined?.a` is `undefined`. Therefore we'd expect `.b` to fail in line A. But it doesn't: The `?.` operator encounters the value `undefined` and the evaluation of the whole expression immediately returns `undefined`.

This behavior differs from a normal operator where JavaScript always evaluates all operands before evaluating the operator. It is called *short-circuiting*. Other short-circuiting operators are:

- `(a && b)`: `b` is only evaluated if `a` is truthy.
- `(a || b)`: `b` is only evaluated if `a` is falsy.
- `(c ? t : e)`: If `c` is truthy, `t` is evaluated. Otherwise, `e` is evaluated.

30.6.4   Optional chaining: downsides and alternatives

Optional chaining also has downsides:

- Deeply nested structures are more difficult to manage. For example, refactoring is harder if there are many sequences of property names: Each one enforces the structure of multiple objects.
- Being so forgiving when accessing data hides problems that will surface much later and are then harder to debug. For example, a typo early in a sequence of optional property names has more negative effects than a normal typo.

An alternative to optional chaining is to extract the information once, in a single location:

- We can either write a helper function that extracts the data.
- Or we can write a function whose input is deeply nested data and whose output is simpler, normalized data.

With either approach, it is possible to perform checks and to fail early if there are problems.

Further reading:

- "Overly defensive programming" by Carl Vitullo

30.6.5   Frequently asked questions

Why are there dots in `o?.[x]` and `f?.()`?

The syntaxes of the following two optional operator are not ideal:

```
obj?.[«expr»]           // better: obj?[«expr»]
func?.(«arg0», «arg1») // better: func?(«arg0», «arg1»)
```

Alas, the less elegant syntax is necessary because distinguishing the ideal syntax (first expression) from the conditional operator (second expression) is too complicated:

```
obj?['a', 'b', 'c'].map(x => x+x)
obj ? ['a', 'b', 'c'].map(x => x+x) : []
```

Why does `null?.prop` evaluate to `undefined` and not `null`?

The operator `?.` is mainly about its right-hand side: Does property `.prop` exist? If not, stop early. Therefore, keeping information about its left-hand side is rarely useful. However, only having a single "early termination" value does simplify things.

30.7   Dictionary objects (advanced)

Objects work best as fixed-layout objects. But before ES6, JavaScript did not have a data structure for dictionaries (ES6 brought Maps). Therefore, objects had to be used as dictionaries, which imposed a signficant constraint: Dictionary keys had to be strings (symbols were also introduced with ES6).

We first look at features of objects that are related to dictionaries but also useful for fixed-layout objects. This section concludes with tips for actually using objects as dictionaries. (Spoiler: If possible, it's better to use Maps.)

30.7.1   Quoted keys in object literals

So far, we have always used fixed-layout objects. Property keys were fixed tokens that had to be valid identifiers and internally became strings:

```
const obj = {
  mustBeAnIdentifier: 123,
};

// Get property
assert.equal(obj.mustBeAnIdentifier, 123);

// Set property
obj.mustBeAnIdentifier = 'abc';
assert.equal(obj.mustBeAnIdentifier, 'abc');
```

As a next step, we'll go beyond this limitation for property keys: In this subsection, we'll use arbitrary fixed strings as keys. In the next subsection, we'll dynamically compute keys.

Two syntaxes enable 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');
```

We can also use these syntaxes for methods:

```
const obj = {
  'A nice method'() {
    return 'Yes!';
  },
};

assert.equal(obj['A nice method'](), 'Yes!');
```

30.7.2 Computed keys in object literals

In the previous subsection, property keys were specified via fixed strings inside object literals. In this section we learn how to dynamically compute property keys. That enables us to use either arbitrary strings or symbols.

The syntax of dynamically computed property keys in object literals is inspired by dynamically accessing properties. That is, we can use square brackets to wrap expressions:

```
const obj = {
  ['Hello world!']: true,
  ['p'+'r'+'o'+'p']: 123,
  [Symbol.toStringTag]: 'Goodbye', // (A)
};

assert.equal(obj['Hello world!'], true);
assert.equal(obj.prop, 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['p'+'r'+'o'+'p'], 123);
assert.equal(obj['==> prop'.slice(4)], 123);
```

Methods can have computed property keys, too:

```
const methodKey = Symbol();
const obj = {
  [methodKey]() {
    return 'Yes!';
  },
};

assert.equal(obj[methodKey](), 'Yes!');
```

For the remainder of this chapter, we'll mostly use fixed property keys again (because they are syntactically more convenient). But all features are also available for arbitrary strings and symbols.

Exercise: Non-destructively updating a property via spreading (computed key)

```
exercises/objects/update_property_test.mjs
```

30.7.3   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 = {
  alpha: 'abc',
  beta: false,
};

assert.equal('alpha' in obj, true);
assert.equal('beta' in obj, true);
assert.equal('unknownKey' in obj, false);
```

Checking if a property exists via truthiness

We can also use a truthiness check to determine if a property exists:

```
assert.equal(
  obj.alpha ? 'exists' : 'does not exist',
  'exists');
assert.equal(
  obj.unknownKey ? 'exists' : 'does not exist',
  'does not exist');
```

The previous checks work because `obj.alpha` is truthy and because reading a missing property returns `undefined` (which is falsy).

There is, however, one important caveat: truthiness checks fail if the property exists, but has a falsy value (`undefined`, `null`, `false`, `0`, `""`, etc.):

```
assert.equal(
  obj.beta ? 'exists' : 'does not exist',
  'does not exist'); // should be: 'exists'
```

30.7.4 Deleting properties

We can delete properties via the `delete` operator:

```
const obj = {
  myProp: 123,
};

assert.deepEqual(Object.keys(obj), ['myProp']);
delete obj.myProp;
assert.deepEqual(Object.keys(obj), []);
```

30.7.5 Enumerability

Enumerability is an *attribute* of a property. Non-enumerable properties are ignored by some operations – for example, by `Object.keys()` and when spreading properties. By default, most properties are enumerable. The next example shows how to change that and how it affects spreading.

```
const enumerableSymbolKey = Symbol('enumerableSymbolKey');
const nonEnumSymbolKey = Symbol('nonEnumSymbolKey');

// We create enumerable properties via an object literal
const obj = {
  enumerableStringKey: 1,
  [enumerableSymbolKey]: 2,
}

// For non-enumerable properties, we need a more powerful tool
Object.defineProperties(obj, {
  nonEnumStringKey: {
    value: 3,
    enumerable: false,
  },
  [nonEnumSymbolKey]: {
    value: 4,
    enumerable: false,
  },
});
```

```
// Non-enumerable properties are ignored by spreading:
assert.deepEqual(
  {...obj},
  {
    enumerableStringKey: 1,
    [enumerableSymbolKey]: 2,
  }
);
```

`Object.defineProperties()` is explained later in this chapter. The next subsection shows how these operations are affected by enumerability:

30.7.6   Listing property keys via `Object.keys()` etc.

	enumerable	non-e.	string	symbol
`Object.keys()`	✔		✔	
`Object.getOwnPropertyNames()`	✔	✔	✔	
`Object.getOwnPropertySymbols()`	✔	✔		✔
`Reflect.ownKeys()`	✔	✔	✔	✔

Table 30.1: Standard library methods for listing *own* (non-inherited) property keys. All of them return Arrays with strings and/or symbols.

Each of the methods in table 30.1 returns an Array with the own property keys of the parameter. In the names of the methods, we can see that the following distinction is made:

- A *property key* can be either a string or a symbol. (`Object.keys()` is older and does not yet follow this convention.)
- A *property name* is a property key whose value is a string.
- A *property symbol* is a property key whose value is a symbol.

To demonstrate the four operations, we revisit the example from the previous subsection:

```
const enumerableSymbolKey = Symbol('enumerableSymbolKey');
const nonEnumSymbolKey = Symbol('nonEnumSymbolKey');

const obj = {
  enumerableStringKey: 1,
  [enumerableSymbolKey]: 2,
}
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,
  ]
);
```

30.7.7   Listing property values via `Object.values()`

`Object.values()` lists the values of all own enumerable string-keyed properties of an object:

```
const firstName = Symbol('firstName');
const obj = {
  [firstName]: 'Jane',
  lastName: 'Doe',
};
assert.deepEqual(
  Object.values(obj),
  ['Doe']);
```

30.7.8   Listing property entries via `Object.entries()` [ES2017]

`Object.entries(obj)` returns an Array with one key-value pair for each of its properties:

- Each pair is encoded as a two-element Array.
- Only own enumerable properties with string keys are included.

```
const firstName = Symbol('firstName');
const obj = {
  [firstName]: 'Jane',
  lastName: 'Doe',
};
Object.defineProperty(
  obj, 'city', {value: 'Metropolis', enumerable: false}
```

```
);
assert.deepEqual(
  Object.entries(obj),
  [
    ['lastName', 'Doe'],
  ]);
```

A simple implementation of `Object.entries()`

The following function is a simplified version of `Object.entries()`:

```
function entries(obj) {
  return Object.keys(obj)
  .map(key => [key, obj[key]]);
}
```

> 🧩 **Exercise: `Object.entries()`**
>
> `exercises/objects/find_key_test.mjs`

30.7.9   Properties are listed deterministically

Own (non-inherited) properties of objects are always listed in the following order:

1. Properties with string keys that contain integer indices (that includes Array indices):
   In ascending numeric order
2. Remaining properties with string keys:
   In the order in which they were added
3. Properties with symbol keys:
   In the order in which they were added

The following example demonstrates that property keys are sorted according to these rules:

```
> Object.keys({b:'',a:'', 10:'',2:''})
[ '2', '10', 'b', 'a' ]
```

> ⚙️ **The order of properties**
>
> The ECMAScript specification describes in more detail how properties are ordered.

30.7.10   Assembling objects via `Object.fromEntries()` [ES2019]

Given an iterable over [key, value] pairs, `Object.fromEntries()` creates an object:

```
const symbolKey = Symbol('symbolKey');
assert.deepEqual(
  Object.fromEntries(
```

```
    [
      ['stringKey', 1],
      [symbolKey, 2],
    ]
  ),
  {
    stringKey: 1,
    [symbolKey]: 2,
  }
);
```

`Object.fromEntries()` does the opposite of `Object.entries()`. However, while `Object.entries()` ignores symbol-keyed properties, `Object.fromEntries()` doesn't (see previous example).

To demonstrate both, we'll use them to implement two tool functions from the library Underscore in the next subsubsections.

Example: `pick()`

The Underscore function `pick()` has the following signature:

```
  pick(object, ...keys)
```

It returns a copy of `object` that has only those properties whose keys are mentioned in the trailing arguments:

```
  const address = {
    street: 'Evergreen Terrace',
    number: '742',
    city: 'Springfield',
    state: 'NT',
    zip: '49007',
  };
  assert.deepEqual(
    pick(address, 'street', 'number'),
    {
      street: 'Evergreen Terrace',
      number: '742',
    }
  );
```

We can implement `pick()` as follows:

```
  function pick(object, ...keys) {
    const filteredEntries = Object.entries(object)
      .filter(([key, _value]) => keys.includes(key));
    return Object.fromEntries(filteredEntries);
  }
```

Example: `invert()`

The Underscore function `invert()` has the following signature:

```
invert(object)
```

It returns a copy of `object` where the keys and values of all properties are swapped:

```
assert.deepEqual(
  invert({a: 1, b: 2, c: 3}),
  {1: 'a', 2: 'b', 3: 'c'}
);
```

We can implement `invert()` like this:

```
function invert(object) {
  const reversedEntries = Object.entries(object)
    .map(([key, value]) => [value, key]);
  return Object.fromEntries(reversedEntries);
}
```

A simple implementation of `Object.fromEntries()`

The following function is a simplified version of `Object.fromEntries()`:

```
function fromEntries(iterable) {
  const result = {};
  for (const [key, value] of iterable) {
    let coercedKey;
    if (typeof key === 'string' || typeof key === 'symbol') {
      coercedKey = key;
    } else {
      coercedKey = String(key);
    }
    result[coercedKey] = value;
  }
  return result;
}
```

🧩 **Exercise: Using `Object.entries()` and `Object.fromEntries()`**

`exercises/objects/omit_properties_test.mjs`

30.7.11   The pitfalls of using an object as a dictionary

If we use plain objects (created via object literals) as dictionaries, we 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 we 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), []);
```

Safely using objects as dictionaries

So how do we avoid the two pitfalls?

- If we can, we use Maps. They are the best solution for dictionaries.
- If we can't, we use a library for objects-as-dictionaries that protects us from making mistakes.
- If that's not possible or desired, we use an object without a prototype.

The following code demonstrates using prototype-less objects as dictionaries:

```
const dict = Object.create(null); // prototype is `null`

assert.equal('toString' in dict, false); // (A)

dict['__proto__'] = 123;
assert.deepEqual(Object.keys(dict), ['__proto__']);
```

We avoided both pitfalls:

- First, a property without a prototype does not inherit any properties (line A).
- Second, in modern JavaScript, `__proto__` is implemented via `Object.prototype`. That means that it is switched off if `Object.prototype` is not in the prototype chain.

Exercise: Using an object as a dictionary

`exercises/objects/simple_dict_test.mjs`

30.8 Property attributes and property descriptors [ES5] (advanced)

Just as objects are composed of properties, properties are composed of *attributes*. There are two kinds of properties and they are characterized by their attributes:

- A *data property* stores data. Its attribute `value` holds any JavaScript value.
  - Methods are data properties whose values are functions.
- An *accessor property* consists of a getter function and/or a setter function. The former is stored in the attribute `get`, the latter in the attribute `set`.

Additionally, there are attributes that both kinds of properties have. The following table lists all attributes and their default values.

Kind of property	Name and type of attribute	Default value
All properties	`configurable: boolean`	`false`
	`enumerable: boolean`	`false`
Data property	`value: any`	`undefined`
	`writable: boolean`	`false`
Accessor property	`get: (this: any) => any`	`undefined`
	`set: (this: any, v: any) => void`	`undefined`

We have already encountered the attributes `value`, `get`, and `set`. The other attributes work as follows:

- `writable` determines if the value of a data property can be changed.
- `configurable` determines if the attributes of a property can be changed. If it is `false`, then:
  - We cannot delete the property.
  - We cannot change a property from a data property to an accessor property or vice versa.
  - We cannot change any attribute other than `value`.
  - However, one more attribute change is allowed: We can change `writable` from `true` to `false`. The rationale behind this anomaly is historical: Property `.length` of Arrays has always been writable and non-configurable. Allowing its `writable` attribute to be changed enables us to freeze Arrays.
- `enumerable` influences some operations (such as `Object.keys()`). If it is `false`, then those operations ignore the property. Enumerability is covered in greater detail earlier in this chapter.

When we are using one of the operations for handling property attributes, attributes are specified via *property descriptors*: objects where each property represents one attribute. For example, this is how we read the attributes of a property `obj.myProp`:

```
const obj = { myProp: 123 };
assert.deepEqual(
  Object.getOwnPropertyDescriptor(obj, 'myProp'),
  {
    value: 123,
    writable: true,
    enumerable: true,
    configurable: true,
  });
```

And this is how we change the attributes of `obj.myProp`:

```
assert.deepEqual(Object.keys(obj), ['myProp']);

// Hide property `myProp` from Object.keys()
// by making it non-enumerable
Object.defineProperty(obj, 'myProp', {
  enumerable: false,
});
```

```
    assert.deepEqual(Object.keys(obj), []);
```

Lastly, let's see what methods and getters look like:

```
const obj = {
  myMethod() {},
  get myGetter() {},
};
const propDescs = Object.getOwnPropertyDescriptors(obj);
propDescs.myMethod.value = typeof propDescs.myMethod.value;
propDescs.myGetter.get = typeof propDescs.myGetter.get;
assert.deepEqual(
  propDescs,
  {
    myMethod: {
      value: 'function',
      writable: true,
      enumerable: true,
      configurable: true
    },
    myGetter: {
      get: 'function',
      set: undefined,
      enumerable: true,
      configurable: true
    }
  }
);
```

> **↗ Further reading**
>
> For more information on property attributes and property descriptors, see *Deep JavaScript*.

30.9 Protecting objects from being changed [ES5] (advanced)

JavaScript has three levels of protecting objects:

- *Preventing extensions* makes it impossible to add new properties to an object and to change its prototype. We can still delete and change properties, though.
  - Apply: `Object.preventExtensions(obj)`
  - Check: `Object.isExtensible(obj)`
- *Sealing* prevents extensions and makes all properties *unconfigurable* (roughly: we can't change how a property works anymore).
  - Apply: `Object.seal(obj)`
  - Check: `Object.isSealed(obj)`

- *Freezing* seals an object after making all of its properties non-writable. That is, the object is not extensible, all properties are read-only and there is no way to change that.
    - Apply: `Object.freeze(obj)`
    - Check: `Object.isFrozen(obj)`

⚠️ **Caveat: Objects are only protected shallowly**

All three of the aforementioned `Object.*` methods only affect the top level of an object, not objects nested inside it.

This is what using `Object.freeze()` looks like:

```
const frozen = Object.freeze({ x: 2, y: 5 });
assert.throws(
  () => frozen.x = 7,
  {
    name: 'TypeError',
    message: /^Cannot assign to read only property 'x'/,
  }
);
```

Changing frozen properties only causes an exception in strict mode. In sloppy mode, it fails silently.

↗️ **Further reading**

For more information on freezing and other ways of locking down objects, see *Deep JavaScript*.

30.10   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, we 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*. Inheritance gives us the impression that we are dealing with single objects, but we are actually dealing with chains of objects.

Figure 30.2 shows what the prototype chain of `obj` looks like. Non-inherited properties
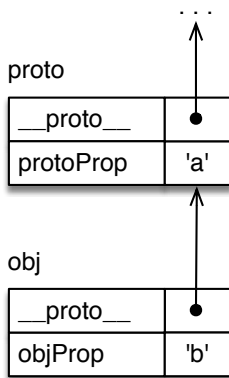
Figure 30.2: `obj` starts a chain of objects that continues with `proto` and other objects.

are called *own properties*. `obj` has one own property, `.objProp`.

30.10.1   JavaScript's operations: all properties vs. own properties

Some operations consider all properties (own and inherited) – for example, getting properties:

```
> const obj = { one: 1 };
> typeof obj.one // own
'number'
> typeof obj.toString // inherited
'function'
```

Other operations only consider own properties – for example, `Object.keys()`:

```
> Object.keys(obj)
[ 'one' ]
```

Read on for another operation that also only considers own properties: setting properties.

30.10.2   Pitfall: only the first member of a prototype chain is mutated

Given an object `obj` with a chain of prototype objects, it makes sense that setting an own property of `obj` only changes `obj`. However, setting an inherited property via `obj` also only changes `obj`. It creates a new own property in `obj` that overrides the inherited property. Let's explore how that works with the following object:

```
const proto = {
  protoProp: 'a',
};
const obj = {
  __proto__: proto,
  objProp: 'b',
};
```

In the next code snippet, we set the inherited property `obj.protoProp` (line A). That "changes" it by creating an own property: When reading `obj.protoProp`, the own property is found first and its value *overrides* the value of the inherited property.

```
// In the beginning, obj has one own property
assert.deepEqual(Object.keys(obj), ['objProp']);

obj.protoProp = 'x'; // (A)

// We created a new own property:
assert.deepEqual(Object.keys(obj), ['objProp', 'protoProp']);

// The inherited property itself is unchanged:
assert.equal(proto.protoProp, 'a');

// The own property overrides the inherited property:
assert.equal(obj.protoProp, 'x');
```

The prototype chain of `obj` is depicted in figure 30.3.
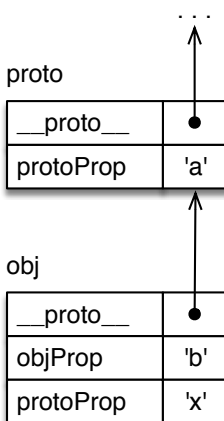
Figure 30.3: The own property `.protoProp` of `obj` overrides the property inherited from `proto`.

30.10.3   Tips for working with prototypes (advanced)

Getting and setting prototypes

Recommendations for `__proto__`:

- Don't use `__proto__` as a pseudo-property (a setter of all instances of `Object`):

  - It can't be used with all objects (e.g. objects that are not instances of `Object`).
  - The language specification has deprecated it.

  For more information on this feature see "`Object.prototype.__proto__` (accessor)" (§31.8.8).

- Using `__proto__` in object literals to set prototypes is different: It's a feature of object literals that has no pitfalls.

The recommended ways of getting and setting prototypes are:

- Getting the prototype of an object:

  ```
  Object.getPrototypeOf(obj: Object) : Object
  ```

- The best time to set the prototype of an object is when we are creating it. We can do so via `__proto__` in an object literal or via:

  ```
  Object.create(proto: Object) : Object
  ```

  If we have to, we can use `Object.setPrototypeOf()` to change the prototype of an existing object. But that may affect performance negatively.

This is how these features are used:

```
const proto1 = {};
const proto2a = {};
const proto2b = {};

const obj1 = {
  __proto__: proto1,
};
assert.equal(Object.getPrototypeOf(obj1), proto1);

const obj2 = Object.create(proto2a);
assert.equal(Object.getPrototypeOf(obj2), proto2a);

Object.setPrototypeOf(obj2, proto2b);
assert.equal(Object.getPrototypeOf(obj2), proto2b);
```

Checking if an object is in the prototype chain of another object

So far, "`proto` is a prototype of `obj`" always meant "`proto` is a *direct* prototype of `obj`". But it can also be used more loosely and mean that `proto` is in the prototype chain of `obj`. That looser relationship can be checked via `.isPrototypeOf()`:

For example:

```
const a = {};
const b = {__proto__: a};
const c = {__proto__: b};

assert.equal(a.isPrototypeOf(b), true);
assert.equal(a.isPrototypeOf(c), true);

assert.equal(c.isPrototypeOf(a), false);
assert.equal(a.isPrototypeOf(a), false);
```

For more information on this method see "Object.prototype.isPrototypeOf()" (§31.8.6).

30.10.4   `Object.hasOwn()`: Is a given property own (non-inherited)? [ES2022]

The `in` operator (line A) checks if an object has a given property. In contrast, `Object.hasOwn()` (lines B and C) checks if a property is own.

```
const proto = {
  protoProp: 'protoProp',
};
const obj = {
  __proto__: proto,
  objProp: 'objProp',
}
assert.equal('protoProp' in obj, true); // (A)
assert.equal(Object.hasOwn(obj, 'protoProp'), false); // (B)
assert.equal(Object.hasOwn(proto, 'protoProp'), true); // (C)
```

> 💡 **Alternative before ES2022: `.hasOwnProperty()`**
>
> Before ES2022, we can use another feature: "Object.prototype.hasOwnProperty()" (§31.8.9). This feature has pitfalls, but the referenced section explains how to work around them.

30.10.5   Sharing data via prototypes

Consider the following code:

```
const jane = {
  firstName: 'Jane',
  describe() {
    return 'Person named '+this.firstName;
  },
};
const tarzan = {
  firstName: 'Tarzan',
  describe() {
    return 'Person named '+this.firstName;
```

```
    },
  };

  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 .firstName and .describe. Additionally, method .describe() is the same. How can we avoid duplicating that method?

We can move it to an object PersonProto and make that object a prototype of both jane and tarzan:

```
  const PersonProto = {
    describe() {
      return 'Person named ' + this.firstName;
    },
  };
  const jane = {
    __proto__: PersonProto,
    firstName: 'Jane',
  };
  const tarzan = {
    __proto__: PersonProto,
    firstName: 'Tarzan',
  };
```

The name of the prototype reflects that both jane and tarzan are persons. Figure 30.4
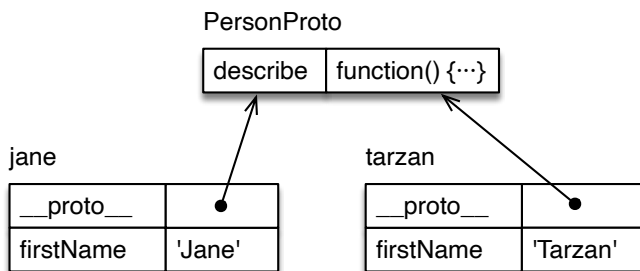
Figure 30.4: Objects jane and tarzan share method .describe(), via their common prototype PersonProto.

illustrates how the three objects are connected: The objects at the bottom now contain the properties that are specific to jane and tarzan. The object at the top contains the properties that are shared between them.

When we make the method call jane.describe(), this points to the receiver of that method call, jane (in the bottom-left corner of the diagram). That's why the method still works. tarzan.describe() works similarly.

```
  assert.equal(jane.describe(), 'Person named Jane');
  assert.equal(tarzan.describe(), 'Person named Tarzan');
```

Looking ahead to the next chapter on classes – this is how classes are organized internally:

- All instances share a common prototype with methods.
- Instance-specific data is stored in own properties in each instance.

"The internals of classes" (§31.3) explains this in more detail.

30.11 FAQ: objects

30.11.1 Why do objects preserve the insertion order of properties?

In principle, objects are unordered. The main reason for ordering properties is so that operations that list entries, keys, or values are deterministic. That helps, e.g., with testing.

30.12 Quick reference: `Object`

30.12.1 `Object.*`: creating objects, handling prototypes

- Object.create(proto, propDescObj?) [ES5]

    - Returns a new object whose prototype is `proto`.
    - The optional `propDescObj` is an object with property descriptors that is used to define properties in the new object.

    ```
    > const obj = Object.create(null);
    > Object.getPrototypeOf(obj)
    null
    ```

    In the following example, we define own properties via the second parameter:

    ```
    const obj = Object.create(
      null,
      {
        color: {
          value: 'green',
          writable: true,
          enumerable: true,
          configurable: true,
        },
      }
    );
    assert.deepEqual(
      obj,
      {
        __proto__: null,
        color: 'green',
      }
    );
    ```

- Object.getPrototypeOf(obj) [ES5]

Return the prototype of `obj` – which is either an object or `null`.

```
assert.equal(
  Object.getPrototypeOf({__proto__: null}), null
);
assert.equal(
  Object.getPrototypeOf({}), Object.prototype
);
assert.equal(
  Object.getPrototypeOf(Object.prototype), null
);
```

- `Object.setPrototypeOf(obj, proto)` [ES6]

Sets the prototype of `obj` to `proto` (which must be `null` or an object) and returns the former.

```
const obj = {};
assert.equal(
  Object.getPrototypeOf(obj), Object.prototype
);
Object.setPrototypeOf(obj, null);
assert.equal(
  Object.getPrototypeOf(obj), null
);
```

30.12.2 `Object.*`: property attributes

- `Object.defineProperty(obj, propKey, propDesc)` [ES5]

    - Defines one property in `obj`, as specified by the property key `propKey` and the property descriptor `propDesc`.
    - Returns `obj`.

```
const obj = {};
Object.defineProperty(
  obj, 'color',
  {
    value: 'green',
    writable: true,
    enumerable: true,
    configurable: true,
  }
);
assert.deepEqual(
  obj,
  {
    color: 'green',
  }
);
```

- `Object.defineProperties(obj, propDescObj)` [ES5]

  – Defines properties in `obj`, as specified by the object `propDescObj` with property descriptors.
  – Returns `obj`.

  ```
  const obj = {};
  Object.defineProperties(
    obj,
    {
      color: {
        value: 'green',
        writable: true,
        enumerable: true,
        configurable: true,
      },
    }
  );
  assert.deepEqual(
    obj,
    {
      color: 'green',
    }
  );
  ```

- `Object.getOwnPropertyDescriptor(obj, propKey)` [ES5]

  – Returns a property descriptor for the own property of `obj` whose key is `prop-Key`. If no such property exists, it returns `undefined`.
  – More information on property descriptors: "Property attributes and property descriptors" (§30.8)

  ```
  > Object.getOwnPropertyDescriptor({a: 1, b: 2}, 'a')
  { value: 1, writable: true, enumerable: true, configurable: true }
  > Object.getOwnPropertyDescriptor({a: 1, b: 2}, 'x')
  undefined
  ```

- `Object.getOwnPropertyDescriptors(obj)` [ES2017]

  – Returns an object with property descriptors, one for each own property of `obj`.
  – More information on property descriptors: "Property attributes and property descriptors" (§30.8)

  ```
  > Object.getOwnPropertyDescriptors({a: 1, b: 2})
  {
    a: { value: 1, writable: true, enumerable: true, configurable: true },
    b: { value: 2, writable: true, enumerable: true, configurable: true },
  }
  ```

30.12.3 `Object.*`: property keys, values, entries

- `Object.keys(obj)` [ES5]

  Returns an Array with all own enumerable property keys that are strings.

  ```js
  const enumSymbolKey = Symbol('enumSymbolKey');
  const nonEnumSymbolKey = Symbol('nonEnumSymbolKey');

  const obj = Object.defineProperties(
    {},
    {
      enumStringKey: {
        value: 1, enumerable: true,
      },
      [enumSymbolKey]: {
        value: 2, enumerable: true,
      },
      nonEnumStringKey: {
        value: 3, enumerable: false,
      },
      [nonEnumSymbolKey]: {
        value: 4, enumerable: false,
      },
    }
  );
  assert.deepEqual(
    Object.keys(obj),
    ['enumStringKey']
  );
  ```

- `Object.getOwnPropertyNames(obj)` [ES5]

  Returns an Array with all own property keys that are strings (enumerable and non-enumerable ones).

  ```js
  const enumSymbolKey = Symbol('enumSymbolKey');
  const nonEnumSymbolKey = Symbol('nonEnumSymbolKey');

  const obj = Object.defineProperties(
    {},
    {
      enumStringKey: {
        value: 1, enumerable: true,
      },
      [enumSymbolKey]: {
        value: 2, enumerable: true,
      },
      nonEnumStringKey: {
        value: 3, enumerable: false,
  ```

```
      },
      [nonEnumSymbolKey]: {
        value: 4, enumerable: false,
      },
    }
  );
  assert.deepEqual(
    Object.getOwnPropertyNames(obj),
    ['enumStringKey', 'nonEnumStringKey']
  );
```

- `Object.getOwnPropertySymbols(obj)` [ES6]

  Returns an Array with all own property keys that are symbols (enumerable and
  non-enumerable ones).

  ```
  const enumSymbolKey = Symbol('enumSymbolKey');
  const nonEnumSymbolKey = Symbol('nonEnumSymbolKey');

  const obj = Object.defineProperties(
    {},
    {
      enumStringKey: {
        value: 1, enumerable: true,
      },
      [enumSymbolKey]: {
        value: 2, enumerable: true,
      },
      nonEnumStringKey: {
        value: 3, enumerable: false,
      },
      [nonEnumSymbolKey]: {
        value: 4, enumerable: false,
      },
    }
  );
  assert.deepEqual(
    Object.getOwnPropertySymbols(obj),
    [enumSymbolKey, nonEnumSymbolKey]
  );
  ```

- `Object.values(obj)` [ES2017]

  Returns an Array with the values of all enumerable own string-keyed properties.

  ```
  > Object.values({a: 1, b: 2})
  [ 1, 2 ]
  ```

- `Object.entries(obj)` [ES2017]

    – Returns an Array with one key-value pair (encoded as a two-element Array)

per property of `obj`.

– Only own enumerable properties with string keys are included.
– Inverse operation: `Object.fromEntries()`

```
const obj = {
  a: 1,
  b: 2,
  [Symbol('myKey')]: 3,
};
assert.deepEqual(
  Object.entries(obj),
  [
    ['a', 1],
    ['b', 2],
    // Property with symbol key is ignored
  ]
);
```

- `Object.fromEntries(keyValueIterable)` [ES2019]

    – Creates an object whose own properties are specified by `keyValueIterable`.
    – Inverse operation: `Object.entries()`

    ```
    > Object.fromEntries([['a', 1], ['b', 2]])
    { a: 1, b: 2 }
    ```

- `Object.hasOwn(obj, key)` [ES2022]

    – Returns `true` if `obj` has an own property whose key is `key`. If not, it returns `false`.

    ```
    > Object.hasOwn({a: 1, b: 2}, 'a')
    true
    > Object.hasOwn({a: 1, b: 2}, 'x')
    false
    ```

30.12.4 `Object.*`: protecting objects

More information: "Protecting objects from being changed" (§30.9)

- `Object.preventExtensions(obj)` [ES5]

    – Makes `obj` non-extensible and returns it.
    – Effect:
        * `obj` is non-extensible: We can't add properties or change its prototype.
    – Only the top level of `obj` is changed (shallow change). Nested objects are not affected.
    – Related: `Object.isExtensible()`

- `Object.isExtensible(obj)` [ES5]

    – Returns `true` if `obj` is extensible and `false` if it isn't.

– Related: `Object.preventExtensions()`

- `Object.seal(obj)` [ES5]

    – Seals `obj` and returns it.
    – Effect:
        * `obj` is non-extensible: We can't add properties or change its prototype.
        * `obj` is sealed: Additionally, all of its properties are unconfigurable.
    – Only the top level of `obj` is changed (shallow change). Nested objects are not affected.
    – Related: `Object.isSealed()`

- `Object.isSealed(obj)` [ES5]

    – Returns `true` if `obj` is sealed and `false` if it isn't.
    – Related: `Object.seal()`

- `Object.freeze(obj)` [ES5]

    – Freezes `obj` and returns it.
    – Effect:
        * `obj` is non-extensible: We can't add properties or change its prototype.
        * `obj` is sealed: Additionally, all of its properties are unconfigurable.
        * `obj` is frozen: Additionally, all of its properties are non-writable.
    – Only the top level of `obj` is changed (shallow change). Nested objects are not affected.
    – Related: `Object.isFrozen()`

```js
const frozen = Object.freeze({ x: 2, y: 5 });
assert.equal(
  Object.isFrozen(frozen), true
);
assert.throws(
  () => frozen.x = 7,
  {
    name: 'TypeError',
    message: /^Cannot assign to read only property 'x'/,
  }
);
```

- `Object.isFrozen(obj)` [ES5]

    – Returns `true` if `obj` is frozen.
    – Related: `Object.freeze()`

30.12.5 `Object.*`: miscellaneous

- `Object.assign(target, ...sources)` [ES6]

    Assigns all enumerable own string-keyed properties of each of the `sources` to `target` and returns `target`.

```
> const obj = {a: 1, b: 1};
> Object.assign(obj, {b: 2, c: 2}, {d: 3})
{ a: 1, b: 2, c: 2, d: 3 }
> obj
{ a: 1, b: 2, c: 2, d: 3 }
```

- Object.groupBy(items, computeGroupKey) [ES2024]

  ```
  Object.groupBy<K extends PropertyKey, T>(
    items: Iterable<T>,
    computeGroupKey: (item: T, index: number) => K,
  ): {[key: K]: Array<T>}
  ```

  – The callback `computeGroupKey` returns a *group key* for each of the `items`.
  – The result of `Object.groupBy()` is an object where:
      * The key of each property is a group key and
      * its value is an Array with all items that have that group key.

  ```
  assert.deepEqual(
    Object.groupBy(
      ['orange', 'apricot', 'banana', 'apple', 'blueberry'],
      (str) => str[0] // compute group key
    ),
    {
      __proto__: null,
      'o': ['orange'],
      'a': ['apricot', 'apple'],
      'b': ['banana', 'blueberry'],
    }
  );
  ```

- Object.is(value1, value2) [ES6]

  Is mostly equivalent to value1 === value2 – with two exceptions:

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

  > -0 === 0
  true
  > Object.is(-0, 0)
  false
  ```

  – Considering all `NaN` values to be equal can be useful – e.g., when searching for a value in an Array.
  – The value `-0` is rare and it's usually best to pretend it is the same as `0`.

30.12.6 `Object.prototype.*`

`Object.prototype` has the following properties:

- `Object.prototype.__proto__` (getter and setter)
- `Object.prototype.hasOwnProperty()`
- `Object.prototype.isPrototypeOf()`
- `Object.prototype.propertyIsEnumerable()`
- `Object.prototype.toLocaleString()`
- `Object.prototype.toString()`
- `Object.prototype.valueOf()`

These methods are explained in detail in "Quick reference: `Object.prototype.*`" (§31.8.1).

30.13 Quick reference: `Reflect`

`Reflect` provides functionality for JavaScript proxies that is also occasionally useful elsewhere:

- `Reflect.apply(target, thisArgument, argumentsList)`

  - Invokes `target` with the arguments provided by `argumentsList` and `this` set to `thisArgument`.
  - Equivalent to `target.apply(thisArgument, argumentsList)`

- `Reflect.construct(target, argumentsList, newTarget=target)`

  - The `new` operator as a function.
  - `target` is the constructor to invoke.
  - The optional parameter `newTarget` points to the constructor that started the current chain of constructor calls.

- `Reflect.defineProperty(target, propertyKey, propDesc)`

  - Similar to `Object.defineProperty()`.
  - Returns a boolean indicating whether or not the operation succeeded.

- `Reflect.deleteProperty(target, propertyKey)`

  The `delete` operator as a function. It works slightly differently, though:

  - It returns `true` if it successfully deleted the property or if the property never existed.
  - It returns `false` if the property could not be deleted and still exists.

  In sloppy mode, the `delete` operator returns the same results as this method. But in strict mode, it throws a `TypeError` instead of returning `false`.

  The only way to protect properties from deletion is by making them non-configurable.

- `Reflect.get(target, propertyKey, receiver=target)`

  A function that gets properties. The optional parameter `receiver` is needed if `get` reaches a getter (somewhere in the prototype chain). Then it provides the value for `this`.

- `Reflect.getOwnPropertyDescriptor(target, propertyKey)`

Same as `Object.getOwnPropertyDescriptor()`.

- `Reflect.getPrototypeOf(target)`

  Same as `Object.getPrototypeOf()`.

- `Reflect.has(target, propertyKey)`

  The `in` operator as a function.

- `Reflect.isExtensible(target)`

  Same as `Object.isExtensible()`.

- `Reflect.ownKeys(target)`

  Returns all own property keys (strings and symbols) in an Array.

- `Reflect.preventExtensions(target)`

    – Similar to `Object.preventExtensions()`.
    – Returns a boolean indicating whether or not the operation succeeded.

- `Reflect.set(target, propertyKey, value, receiver=target)`

    – Sets properties.
    – Returns a boolean indicating whether or not the operation succeeded.

- `Reflect.setPrototypeOf(target, proto)`

    – Same as `Object.setPrototypeOf()`.
    – Returns a boolean indicating whether or not the operation succeeded.

30.13.1 `Reflect.*` vs. `Object.*`

General recommendations:

- Use `Object.*` whenever you can.
- Use `Reflect.*` when working with ECMAScript proxies. Its methods are well adapted to ECMAScript's meta-object protocol (MOP) which also return boolean error flags instead of exceptions.

What are use cases for `Reflect` beyond proxies?

- `Reflect.ownKeys()` lists all own property keys – functionality that isn't provided anywhere else.

- Same functionality as `Object` but different return values: `Reflect` duplicates the following methods of `Object`, but its methods return booleans indicating whether the operation succeeded (where the `Object` methods return the object that was modified).

    – `Object.defineProperty(obj, propKey, propDesc)`
    – `Object.preventExtensions(obj)`
    – `Object.setPrototypeOf(obj, proto)`

- Operators as functions: The following `Reflect` methods implement functionality that is otherwise only available via operators:

– Reflect.construct(target, argumentsList, newTarget=target)
– Reflect.deleteProperty(target, propertyKey)
– Reflect.get(target, propertyKey, receiver=target)
– Reflect.has(target, propertyKey)
– Reflect.set(target, propertyKey, value, receiver=target)

- Shorter version of apply(): If we want to be completely safe about invoking the method apply() on a function, we can't do so via dynamic dispatch, because the function may have an own property with the key 'apply':

```
func.apply(thisArg, argArray) // not safe
Function.prototype.apply.call(func, thisArg, argArray) // safe
```

Using Reflect.apply() is shorter:

```
Reflect.apply(func, thisArg, argArray)
```

- No exceptions when deleting properties: the delete operator throws in strict mode if we try to delete a non-configurable own property. Reflect.deleteProperty() returns false in that case.

Chapter 31

Classes [ES6]

In this book, JavaScript's style of object-oriented programming (OOP) is introduced in four steps. This chapter covers step 3 and 4, the previous chapter covers step 1 and 2. The steps are (figure 31.1):

1. **Single objects (previous chapter):** How do *objects*, JavaScript's basic OOP building blocks, work in isolation?
2. **Prototype chains (previous chapter):** Each object has a chain of zero or more *prototype objects*. Prototypes are JavaScript's core inheritance mechanism.
3. **Classes (this chapter):** JavaScript's *classes* are factories for objects. The relationship between a class and its instances is based on prototypal inheritance (step 2).
4. **Subclassing (this chapter):** The relationship between a *subclass* and its *superclass* is also based on prototypal inheritance.

31.1 Cheat sheet: classes

A JavaScript class:

```
class Person {
  constructor(firstName) { // (A)
    this.firstName = firstName; // (B)
```
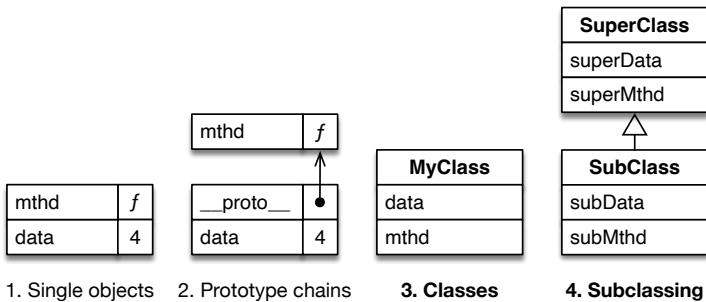
Figure 31.1: This book introduces object-oriented programming in JavaScript in four steps.

```
  }
  describe() { // (C)
    return 'Person named ' + this.firstName;
  }
}
const tarzan = new Person('Tarzan');
assert.equal(
  tarzan.firstName, 'Tarzan'
);
assert.equal(
  tarzan.describe(),
  'Person named Tarzan'
);
// One property (public slot)
assert.deepEqual(
  Reflect.ownKeys(tarzan), ['firstName']
);
```

Explanations:

- Inside a class, `this` refers to the current instance
- Line A: constructor of the class
- Line B: Property `.firstName` (a public slot) is created (no prior declaration necessary).
- Line C: method `.describe()`

Public instance data such as `.firstName` is relatively common in JavaScript.

The same class `Person`, but with private instance data:

```
class Person {
  #firstName; // (A)
  constructor(firstName) {
    this.#firstName = firstName; // (B)
  }
  describe() {
    return 'Person named ' + this.#firstName;
```

```
  }
}
const tarzan = new Person('Tarzan');
assert.equal(
  tarzan.describe(),
  'Person named Tarzan'
);
// No properties, only a private field
assert.deepEqual(
  Reflect.ownKeys(tarzan), []
);
```

Explanations:

- Line A: private field `.#firstName`. In contrast to properties, private fields must be declared (line A) before they can be used (line B). A private field can only be accessed inside the class that declares it. It can't even be accessed by subclasses.

Class `Employee` is a subclass of `Person`:

```
class Employee extends Person {
  #title;

  constructor(firstName, title) {
    super(firstName); // (A)
    this.#title = title;
  }
  describe() {
    return `${super.describe()} (${this.#title})`; // (B)
  }
}

const jane = new Employee('Jane', 'CTO');
assert.equal(
  jane.describe(),
  'Person named Jane (CTO)'
);
```

- Line A: In subclasses, we can omit the constructor. If we don't, we have to call `super()`.
- Line B: We can refer to overridden methods via `super`.

The next class demonstrates how to create properties via *public fields* (line A):

```
class StringBuilderClass {
  string = ''; // (A)
  add(str) {
    this.string += str;
    return this;
  }
}
```

```
const sb = new StringBuilderClass();
sb.add('Hello').add(' everyone').add('!');
assert.equal(
  sb.string, 'Hello everyone!'
);
```

JavaScript also supports `static` members, but external functions and variables are often preferred.

31.2 The essentials of classes

Classes are basically a compact syntax for setting up prototype chains (which are explained in the previous chapter). Under the hood, JavaScript's classes are unconventional. But that is something we rarely see when working with them. They should normally feel familiar to people who have used other object-oriented programming languages.

Note that we don't need classes to create objects. We can also do so via object literals. That's why the singleton pattern isn't needed in JavaScript and classes are used less than in many other languages that have them.

31.2.1 A class for persons

We have previously worked with `jane` and `tarzan`, single objects representing persons. Let's use a *class declaration* to implement a factory for such objects:

```
class Person {
  #firstName; // (A)
  constructor(firstName) {
    this.#firstName = firstName; // (B)
  }
  describe() {
    return `Person named ${this.#firstName}`;
  }
  static extractNames(persons) {
    return persons.map(person => person.#firstName);
  }
}
```

jane and tarzan can now be created via `new Person()`:

```
const jane = new Person('Jane');
const tarzan = new Person('Tarzan');
```

Let's examine what's inside the body of class `Person`.

- `.constructor()` is a special method that is called after the creation of a new instance. Inside it, `this` refers to that instance.

- `.#firstName` [ES2022] is an *instance private field*: Such fields are stored in instances. They are accessed similarly to properties, but their names are separate – they always

start with hash symbols (#). And they are invisible to the world outside the class:

```
assert.deepEqual(
  Reflect.ownKeys(jane),
  []
);
```

Before we can initialize `.#firstName` in the constructor (line B), we need to declare it by mentioning it in the class body (line A).

- `.describe()` is a method. If we invoke it via `obj.describe()` then `this` refers to `obj` inside the body of `.describe()`.

  ```
  assert.equal(
    jane.describe(), 'Person named Jane'
  );
  assert.equal(
    tarzan.describe(), 'Person named Tarzan'
  );
  ```

- `.extractName()` is a *static* method. "Static" means that it belongs to the class, not to instances:

  ```
  assert.deepEqual(
    Person.extractNames([jane, tarzan]),
    ['Jane', 'Tarzan']
  );
  ```

We can also create instance properties (public fields) in constructors:

```
class Container {
  constructor(value) {
    this.value = value;
  }
}
const abcContainer = new Container('abc');
assert.equal(
  abcContainer.value, 'abc'
);
```

In contrast to instance private fields, instance properties don't have to be declared in class bodies.

31.2.2 Class expressions

There are two kinds of *class definitions* (ways of defining classes):

- *Class declarations*, which we have seen in the previous section.
- *Class expressions*, which we'll see next.

Class expressions can be anonymous and named:

```
// Anonymous class expression
const Person = class { ··· };
```

```
// Named class expression
const Person = class MyClass { ··· };
```

The name of a named class expression works similarly to the name of a named function expression: It can only be accessed inside the body of a class and stays the same, regardless of what the class is assigned to.

31.2.3   The `instanceof` operator

The `instanceof` operator tells us 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.

31.2.4   Public slots (properties) vs. private slots

In the JavaScript language, objects can have two kinds of "slots".

- *Public slots* (which are are also called *properties*). For example, methods are public slots.
- *Private slots* [ES2022]. For example, private fields are private slots.

These are the most important rules we need to know about properties and private slots:

- In classes, we can use public and private versions of fields, methods, getters and setters. All of them are slots in objects. Which objects they are placed in depends on whether the keyword `static` is used and other factors.
- A getter and a setter that have the same key create a single *accessor* slot. An Accessor can also have only a getter or only a setter.
- Properties and private slots are very different – for example:
  - They are stored separately.
  - Their keys are different. The keys of private slots can't even be accessed directly (see "Each private slot has a unique key (a *private name*)" (§31.2.5.2) later in this chapter).
  - Properties are inherited from prototypes, private slots aren't.
  - Private slots can only be created via classes.

The following class demonstrates the two kinds of slots. Each of its instances has one private field and one property:

```
class MyClass {
  #instancePrivateField = 1;
```

```
  instanceProperty = 2;
  getInstanceValues() {
    return [
      this.#instancePrivateField,
      this.instanceProperty,
    ];
  }
}
const inst = new MyClass();
assert.deepEqual(
  inst.getInstanceValues(), [1, 2]
);
```

As expected, outside `MyClass`, we can only see the property:

```
assert.deepEqual(
  Reflect.ownKeys(inst),
  ['instanceProperty']
);
```

> ⚙️ **More information on properties**
>
> This chapter doesn't cover all details of properties (just the essentials). If you want to dig deeper, you can do so in "Property attributes and property descriptors" (§30.8)

Next, we'll look at some of the details of private slots.

31.2.5   Private slots in more detail [ES2022] (advanced)

Private slots can't be accessed in subclasses

A private slot really can only be accessed inside the class that declares it. We can't even access it from a subclass:

```
class SuperClass {
  #superProp = 'superProp';
}
class SubClass extends SuperClass {
  getSuperProp() {
    return this.#superProp;
  }
}
// SyntaxError: Private field '#superProp'
// must be declared in an enclosing class
```

Subclassing via `extends` is explained later in this chapter. How to work around this limitation is explained in "Simulating protected visibility and friend visibility via WeakMaps" (§31.5.4).

Each private slot has a unique key (a *private name*)

Private slots have unique keys that are similar to symbols. Consider the following class from earlier:

```
class MyClass {
  #instancePrivateField = 1;
  instanceProperty = 2;
  getInstanceValues() {
    return [
      this.#instancePrivateField,
      this.instanceProperty,
    ];
  }
}
```

Internally, the private field of `MyClass` is handled roughly like this:

```
let MyClass;
{ // Scope of the body of the class
  const instancePrivateFieldKey = Symbol();
  MyClass = class {
    __PrivateElements__ = new Map([
      [instancePrivateFieldKey, 1],
    ]);
    instanceProperty = 2;
    getInstanceValues() {
      return [
        this.__PrivateElements__.get(instancePrivateFieldKey),
        this.instanceProperty,
      ];
    }
  }
}
```

The value of `instancePrivateFieldKey` is called a *private name*. We can't use private names directly in JavaScript, we can only use them indirectly, via the fixed identifiers of private fields, private methods, and private accessors. Where the fixed identifiers of public slots (such as `getInstanceValues`) are interpreted as string keys, the fixed identifiers of private slots (such as `#instancePrivateField`) refer to private names (similarly to how variable names refer to values).

> ⚙️ **Private slots in the ECMAScript language specification**
>
> Section "Object Internal Methods and Internal Slots" in the ECMAScript language specification explains how private slots work. Search for "[[PrivateElements]]".

Private names are statically scoped (like variables)

A callable entity can only access the name of a private slot if it was born inside the scope where the name was declared. However, it doesn't lose this ability if it moves somewhere else later on:

```
class MyClass {
  #privateData = 'hello';
  static createGetter() {
    return (obj) => obj.#privateData; // (A)
  }
}

const myInstance = new MyClass();
const getter = MyClass.createGetter();
assert.equal(
  getter(myInstance), 'hello' // (B)
);
```

The arrow function `getter` was born inside `MyClass` (line A), but it can still access the private name `#privateData` after it left its birth scope (line B).

The same private identifier refers to different private names in different classes

Because the identifiers of private slots aren't used as keys, using the same identifier in different classes produces different slots (line A and line C):

```
class Color {
  #name; // (A)
  constructor(name) {
    this.#name = name; // (B)
  }
  static getName(obj) {
    return obj.#name;
  }
}
class Person {
  #name; // (C)
  constructor(name) {
    this.#name = name;
  }
}

assert.equal(
  Color.getName(new Color('green')), 'green'
);

// We can't access the private slot #name of a Person in line B:
assert.throws(
  () => Color.getName(new Person('Jane')),
```

```
  {
    name: 'TypeError',
    message: 'Cannot read private member #name from'
      + ' an object whose class did not declare it',
  }
);
```

The names of private fields never clash

Even if a subclass uses the same name for a private field, the two names never clash because they refer to private names (which are always unique). In the following example, `.#privateField` in `SuperClass` does not clash with `.#privateField` in `SubClass`, even though both slots are stored directly in `inst`:

```
class SuperClass {
  #privateField = 'super';
  getSuperPrivateField() {
    return this.#privateField;
  }
}
class SubClass extends SuperClass {
  #privateField = 'sub';
  getSubPrivateField() {
    return this.#privateField;
  }
}
const inst = new SubClass();
assert.equal(
  inst.getSuperPrivateField(), 'super'
);
assert.equal(
  inst.getSubPrivateField(), 'sub'
);
```

Subclassing via `extends` is explained later in this chapter.

Using `in` to check if an object has a given private slot

The `in` operator can be used to check if a private slot exists (line A):

```
class Color {
  #name;
  constructor(name) {
    this.#name = name;
  }
  static check(obj) {
    return #name in obj; // (A)
  }
}
```

Let's look at more examples of `in` applied to private slots.

Private methods. The following code shows that private methods create private slots in instances:

```
class C1 {
  #priv() {}
  static check(obj) {
    return #priv in obj;
  }
}
assert.equal(C1.check(new C1()), true);
```

Static private fields. We can also use in for a static private field:

```
class C2 {
  static #priv = 1;
  static check(obj) {
    return #priv in obj;
  }
}
assert.equal(C2.check(C2), true);
assert.equal(C2.check(new C2()), false);
```

Static private methods. And we can check for the slot of a static private method:

```
class C3 {
  static #priv() {}
  static check(obj) {
    return #priv in obj;
  }
}
assert.equal(C3.check(C3), true);
```

Using the same private identifier in different classes. In the next example, the two classes Color and Person both have a slot whose identifier is #name. The in operator distinguishes them correctly:

```
class Color {
  #name;
  constructor(name) {
    this.#name = name;
  }
  static check(obj) {
    return #name in obj;
  }
}
class Person {
  #name;
  constructor(name) {
    this.#name = name;
  }
  static check(obj) {
```

```
    return #name in obj;
  }
}

// Detecting Color's #name
assert.equal(
  Color.check(new Color()), true
);
assert.equal(
  Color.check(new Person()), false
);

// Detecting Person's #name
assert.equal(
  Person.check(new Person()), true
);
assert.equal(
  Person.check(new Color()), false
);
```

31.2.6 The pros and cons of classes in JavaScript

I recommend using classes for the following reasons:

- Classes are a common standard for object creation and inheritance that is now widely supported across libraries and frameworks. This is an improvement compared to how things were before, when almost every framework had its own inheritance library.

- They help tools such as IDEs and type checkers with their work and enable new features there.

- If you come from another language to JavaScript and are used to classes, then you can get started more quickly.

- JavaScript engines optimize them. That is, code that uses classes is almost always faster than code that uses a custom inheritance library.

- We can subclass built-in constructor functions such as `Error`.

That doesn't mean that classes are perfect:

- There is a risk of overdoing inheritance.

- There is a risk of putting too much functionality in classes (when some of it is often better put in functions).

- Classes look familiar to programmers coming from other languages, but they work differently and are used differently (see next subsection). Therefore, there is a risk of those programmers writing code that doesn't feel like JavaScript.

- How classes seem to work superficially is quite different from how they actually work. In other words, there is a disconnect between syntax and semantics. Two

examples are:

- A method definition inside a class C creates a method in the object C.prototype.
- Classes are functions.

The motivation for the disconnect is backward compatibility. Thankfully, the disconnect causes few problems in practice; we are usually OK if we go along with what classes pretend to be.

This was a first look at classes. We'll explore more features soon.

Exercise: Writing a class

exercises/classes/point_class_test.mjs

31.2.7   Tips for using classes

- Use inheritance sparingly – it tends to make code more complicated and spread out related functionality across multiple locations.
- Instead of static members, it is often better to use external functions and variables. We can even make those private to a module, simply by not exporting them. Two important exceptions to this rule are:
  - Operations that need access to private slots
  - Static factory methods
- Only put core functionality in prototype methods. Other functionality is better implemented via functions – especially algorithms that involve instances of multiple classes.

31.3   The internals of classes

31.3.1   A class is actually two connected objects

Under the hood, a class becomes two connected objects. Let's revisit class Person to see how that works:

```
class Person {
  #firstName;
  constructor(firstName) {
    this.#firstName = firstName;
  }
  describe() {
    return `Person named ${this.#firstName}`;
  }
  static extractNames(persons) {
    return persons.map(person => person.#firstName);
  }
}
```

The first object created by the class is stored in Person. It has four properties:

```
assert.deepEqual(
  Reflect.ownKeys(Person),
  ['length', 'name', 'prototype', 'extractNames']
);

// The number of parameters of the constructor
assert.equal(
  Person.length, 1
);

// The name of the class
assert.equal(
  Person.name, 'Person'
);
```

The two remaining properties are:

- `Person.extractNames` is the static method that we have already seen in action.
- `Person.prototype` points to the second object that is created by a class definition.

These are the contents of `Person.prototype`:

```
assert.deepEqual(
  Reflect.ownKeys(Person.prototype),
  ['constructor', 'describe']
);
```

There are two properties:

- `Person.prototype.constructor` points to the constructor.
- `Person.prototype.describe` is the method that we have already used.

31.3.2 Classes set up the prototype chains of their instances

The object `Person.prototype` is the prototype of all instances:

```
const jane = new Person('Jane');
assert.equal(
  Object.getPrototypeOf(jane), Person.prototype
);

const tarzan = new Person('Tarzan');
assert.equal(
  Object.getPrototypeOf(tarzan), Person.prototype
);
```

That explains how the instances get their methods: They inherit them from the object `Person.prototype`.

Figure 31.2 visualizes how everything is connected.
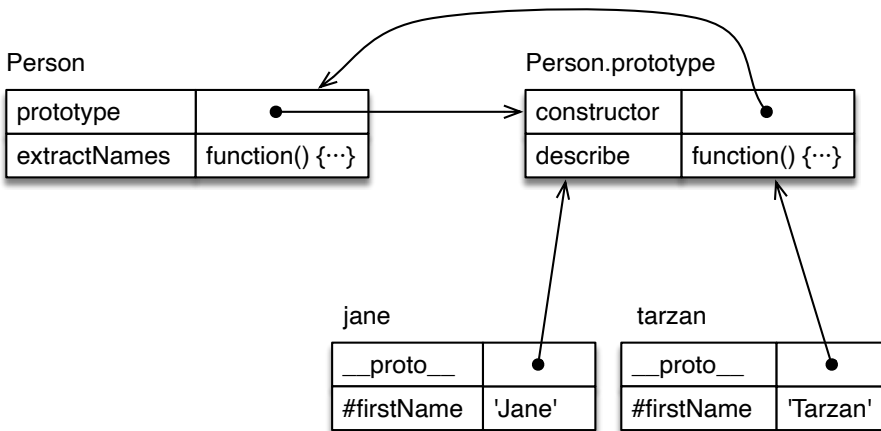
Figure 31.2: The class `Person` has the property `.prototype` that points to an object that is the prototype of all instances of `Person`. The objects `jane` and `tarzan` are two such instances.

31.3.3 `.__proto__` vs. `.prototype`

It is easy to confuse `.__proto__` and `.prototype`. Hopefully, figure 31.2 makes it clear how they differ:

- `Object.prototype.__proto__` is an accessor that most objects inherit that gets and sets the prototype of the receiver. Therefore the following two expressions are equivalent:

  ```
  someObj.__proto__
  Object.getPrototypeOf(someObj)
  ```

  As are the following two expressions:

  ```
  someObj.__proto__ = anotherObj
  Object.setPrototypeOf(someObj, anotherObj)
  ```

- `SomeClass.prototype` holds the object that becomes the prototype of all instances of `SomeClass`. A better name for `.prototype` would be `.instancePrototype`. This property is only special because the `new` operator uses it to set up instances of `SomeClass`.

  ```
  class SomeClass {}
  const inst = new SomeClass();
  assert.equal(
    Object.getPrototypeOf(inst), SomeClass.prototype
  );
  ```

31.3.4 `Person.prototype.constructor` (advanced)

There is one detail in figure 31.2 that we haven't looked at, yet: `Person.prototype.constructor` points back to `Person`:

```
> Person.prototype.constructor === Person
true
```

This setup exists due to backward compatibility. But it has two additional benefits.

First, each instance of a class inherits property `.constructor`. Therefore, given an instance, we can make "similar" objects via it:

```
const jane = new Person('Jane');

const cheeta = new jane.constructor('Cheeta');
// cheeta is also an instance of Person
assert.equal(cheeta instanceof Person, true);
```

Second, we can get the name of the class that created a given instance:

```
const tarzan = new Person('Tarzan');
assert.equal(tarzan.constructor.name, 'Person');
```

31.3.5 Dispatched vs. direct method calls (advanced)

In this subsection, we learn about two different ways of invoking methods:

- Dispatched method calls
- Direct method calls

Understanding both of them will give us important insights into how methods work.

We'll also need the second way later in this chapter: It will allow us to borrow useful methods from `Object.prototype`.

Dispatched method calls

Let's examine how method calls work with classes. We are revisiting `jane` from earlier:

```
class Person {
  #firstName;
  constructor(firstName) {
    this.#firstName = firstName;
  }
  describe() {
    return 'Person named '+this.#firstName;
  }
}
const jane = new Person('Jane');
```

Figure 31.3 has a diagram with `jane`'s prototype chain. Normal method calls are *dispatched* – the method call

```
jane.describe()
```

happens in two steps:

- Dispatch: JavaScript traverses the prototype chain starting with `jane` to find the first object that has an own property with the key `'describe'`: It first looks at `jane` and doesn't find an own property `.describe`. It continues with `jane`'s prototype, `Person.prototype` and finds an own property `describe` whose value it returns.
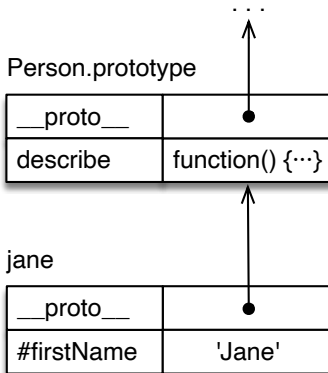
Figure 31.3: The prototype chain of `jane` starts with `jane` and continues with `Person.prototype`.

```
const func = jane.describe;
```

- Invocation: Method-invoking a value is different from function-invoking a value in that it not only calls what comes before the parentheses with the arguments inside the parentheses but also sets `this` to the receiver of the method call (in this case, `jane`):

```
func.call(jane);
```

This way of dynamically looking for a method and invoking it is called *dynamic dispatch*.

Direct method calls

We can also make method calls *directly*, without dispatching:

```
Person.prototype.describe.call(jane)
```

This time, we directly point to the method via `Person.prototype.describe` and don't search for it in the prototype chain. We also specify `this` differently – via `.call()`.

> ⚙️ **`this` always points to the instance**
>
> No matter where in the prototype chain of an instance a method is located, `this` always points to the instance (the beginning of the prototype chain). That enables `.describe()` to access `.#firstName` in the example.

When are direct method calls useful? Whenever we want to borrow a method from elsewhere that a given object doesn't have – for example:

```
const obj = Object.create(null);

// `obj` is not an instance of Object and doesn't inherit
// its prototype method .toString()
```

```
assert.throws(
  () => obj.toString(),
  /^TypeError: obj.toString is not a function$/
);
assert.equal(
  Object.prototype.toString.call(obj),
  '[object Object]'
);
```

31.3.6 Classes evolved from ordinary functions (advanced)

Before ECMAScript 6, JavaScript didn't have classes. Instead, ordinary functions were used as *constructor functions*:

```
function StringBuilderConstr(initialString) {
  this.string = initialString;
}
StringBuilderConstr.prototype.add = function (str) {
  this.string += str;
  return this;
};

const sb = new StringBuilderConstr('¡');
sb.add('Hola').add('!');
assert.equal(
  sb.string, '¡Hola!'
);
```

Classes provide better syntax for this approach:

```
class StringBuilderClass {
  constructor(initialString) {
    this.string = initialString;
  }
  add(str) {
    this.string += str;
    return this;
  }
}
const sb = new StringBuilderClass('¡');
sb.add('Hola').add('!');
assert.equal(
  sb.string, '¡Hola!'
);
```

Subclassing is especially tricky with constructor functions. Classes also offer benefits that go beyond more convenient syntax:

- Built-in constructor functions such as Error can be subclassed.
- We can access overridden properties via super.
- Classes can't be function-called.

- Methods can't be `new`-called and don't have the property `.prototype`.
- Support for private instance data.
- And more.

Classes are so compatible with constructor functions that they can even extend them:

```
function SuperConstructor() {}
class SubClass extends SuperConstructor {}

assert.equal(
  new SubClass() instanceof SuperConstructor, true
);
```

`extends` and subclassing are explained .

A class is the constructor

This brings us to an interesting insight. On one hand, `StringBuilderClass` refers to its constructor via `StringBuilderClass.prototype.constructor`.

On the other hand, the class *is* the constructor (a function):

```
> StringBuilderClass.prototype.constructor === StringBuilderClass
true
> typeof StringBuilderClass
'function'
```

> ⚙ **Constructor (functions) vs. classes**
>
> Due to how similar they are, I use the terms *constructor (function)* and *class* interchangeably.

31.4   Prototype members of classes

31.4.1   Public prototype methods and accessors

All members in the body of the following class declaration create properties of `PublicProtoClass.prototype`.

```
class PublicProtoClass {
  constructor(args) {
    // (Do something with `args` here.)
  }
  publicProtoMethod() {
    return 'publicProtoMethod';
  }
  get publicProtoAccessor() {
    return 'publicProtoGetter';
  }
  set publicProtoAccessor(value) {
```

```
      assert.equal(value, 'publicProtoSetter');
    }
  }

  assert.deepEqual(
    Reflect.ownKeys(PublicProtoClass.prototype),
    ['constructor', 'publicProtoMethod', 'publicProtoAccessor']
  );

  const inst = new PublicProtoClass('arg1', 'arg2');
  assert.equal(
    inst.publicProtoMethod(), 'publicProtoMethod'
  );
  assert.equal(
    inst.publicProtoAccessor, 'publicProtoGetter'
  );
  inst.publicProtoAccessor = 'publicProtoSetter';
```

All kinds of public prototype methods and accessors (advanced)

```
  const accessorKey = Symbol('accessorKey');
  const syncMethodKey = Symbol('syncMethodKey');
  const syncGenMethodKey = Symbol('syncGenMethodKey');
  const asyncMethodKey = Symbol('asyncMethodKey');
  const asyncGenMethodKey = Symbol('asyncGenMethodKey');

  class PublicProtoClass2 {
    // Identifier keys
    get accessor() {}
    set accessor(value) {}
    syncMethod() {}
    * syncGeneratorMethod() {}
    async asyncMethod() {}
    async * asyncGeneratorMethod() {}

    // Quoted keys
    get 'an accessor'() {}
    set 'an accessor'(value) {}
    'sync method'() {}
    * 'sync generator method'() {}
    async 'async method'() {}
    async * 'async generator method'() {}

    // Computed keys
    get [accessorKey]() {}
    set [accessorKey](value) {}
    [syncMethodKey]() {}
    * [syncGenMethodKey]() {}
```

```
  async [asyncMethodKey]() {}
  async * [asyncGenMethodKey]() {}
}

// Quoted and computed keys are accessed via square brackets:
const inst = new PublicProtoClass2();
inst['sync method']();
inst[syncMethodKey]();
```

Quoted and computed keys can also be used in object literals:

- "Quoted keys in object literals" (§30.7.1)
- "Computed keys in object literals" (§30.7.2)

More information on accessors (defined via getters and/or setters), generators, async methods, and async generator methods:

- "Object literals: accessors" (§30.3.6)
- "Synchronous generators"
- "Async functions"
- "Asynchronous generators"

31.4.2 Private methods and accessors [ES2022]

Private methods (and accessors) are an interesting mix of prototype members and instance members.

On one hand, private methods are stored in slots in instances (line A):

```
class MyClass {
  #privateMethod() {}
  static check() {
    const inst = new MyClass();
    assert.equal(
      #privateMethod in inst, true // (A)
    );
    assert.equal(
      #privateMethod in MyClass.prototype, false
    );
    assert.equal(
      #privateMethod in MyClass, false
    );
  }
}
MyClass.check();
```

Why are they not stored in `.prototype` objects? Private slots are not inherited, only properties are.

On the other hand, private methods are shared between instances – like prototype public methods:

```
class MyClass {
  #privateMethod() {}
  static check() {
    const inst1 = new MyClass();
    const inst2 = new MyClass();
    assert.equal(
      inst1.#privateMethod,
      inst2.#privateMethod
    );
  }
}
```

Due to that and due to their syntax being similar to prototype public methods, they are covered here.

The following code demonstrates how private methods and accessors work:

```
class PrivateMethodClass {
  #privateMethod() {
    return 'privateMethod';
  }
  get #privateAccessor() {
    return 'privateGetter';
  }
  set #privateAccessor(value) {
    assert.equal(value, 'privateSetter');
  }
  callPrivateMembers() {
    assert.equal(this.#privateMethod(), 'privateMethod');
    assert.equal(this.#privateAccessor, 'privateGetter');
    this.#privateAccessor = 'privateSetter';
  }
}
assert.deepEqual(
  Reflect.ownKeys(new PrivateMethodClass()), []
);
```

All kinds of private methods and accessors (advanced)

With private slots, the keys are always identifiers:

```
class PrivateMethodClass2 {
  get #accessor() {}
  set #accessor(value) {}
  #syncMethod() {}
  * #syncGeneratorMethod() {}
  async #asyncMethod() {}
  async * #asyncGeneratorMethod() {}
}
```

More information on accessors (defined via getters and / or setters), generators, async methods, and async generator methods:

- "Object literals: accessors" (§30.3.6)
- "Synchronous generators"
- "Async functions"
- "Asynchronous generators"

31.5   Instance members of classes [ES2022]

31.5.1   Instance public fields

Instances of the following class have two instance properties (created in line A and line B):

```
class InstPublicClass {
  // Instance public field
  instancePublicField = 0; // (A)

  constructor(value) {
    // We don't need to mention .property elsewhere!
    this.property = value; // (B)
  }
}

const inst = new InstPublicClass('constrArg');
assert.deepEqual(
  Reflect.ownKeys(inst),
  ['instancePublicField', 'property']
);
assert.equal(
  inst.instancePublicField, 0
);
assert.equal(
  inst.property, 'constrArg'
);
```

If we create an instance property inside the constructor (line B), we don't need to "declare" it elsewhere. As we have already seen, that is different for instance private fields.

Note that instance properties are relatively common in JavaScript; much more so than in, e.g., Java, where most instance state is private.

Instance public fields with quoted and computed keys (advanced)

```
const computedFieldKey = Symbol('computedFieldKey');
class InstPublicClass2 {
  'quoted field key' = 1;
  [computedFieldKey] = 2;
}
const inst = new InstPublicClass2();
```

```
assert.equal(inst['quoted field key'], 1);
assert.equal(inst[computedFieldKey], 2);
```

What is the value of `this` in instance public fields? (advanced)

In the initializer of a instance public field, `this` refers to the newly created instance:

```
class MyClass {
  instancePublicField = this;
}
const inst = new MyClass();
assert.equal(
  inst.instancePublicField, inst
);
```

When are instance public fields executed? (advanced)

The execution of instance public fields roughly follows these two rules:

- In base classes (classes without superclasses), instance public fields are executed immediately before the constructor.
- In derived classes (classes with superclasses):
    - The superclass sets up its instance slots when `super()` is called.
    - Instance public fields are executed immediately after `super()`.

The following example demonstrates these rules:

```
class SuperClass {
  superProp = console.log('superProp');
  constructor() {
    console.log('super-constructor');
  }
}
class SubClass extends SuperClass {
  subProp = console.log('subProp');
  constructor() {
    console.log('BEFORE super()');
    super();
    console.log('AFTER super()');
  }
}
new SubClass();
```

Output:

```
BEFORE super()
superProp
super-constructor
subProp
AFTER super()
```

extends and subclassing are explained later in this chapter.

31.5.2   Instance private fields

The following class contains two instance private fields (line A and line B):

```
class InstPrivateClass {
  #privateField1 = 'private field 1'; // (A)
  #privateField2; // (B) required!
  constructor(value) {
    this.#privateField2 = value; // (C)
  }
  /**
   * Private fields are not accessible outside the class body.
   */
  checkPrivateValues() {
    assert.equal(
      this.#privateField1, 'private field 1'
    );
    assert.equal(
      this.#privateField2, 'constructor argument'
    );
  }
}

const inst = new InstPrivateClass('constructor argument');
  inst.checkPrivateValues();

// No instance properties were created
assert.deepEqual(
  Reflect.ownKeys(inst),
  []
);
```

Note that we can only use `.#privateField2` in line C if we declare it in the class body.

31.5.3   Private instance data before ES2022 (advanced)

In this section, we look at two techniques for keeping instance data private. Because they don't rely on classes, we can also use them for objects that were created in other ways – e.g., via object literals.

Before ES6: private members via naming conventions

The first technique makes a property private by prefixing its name with an underscore. This doesn't protect the property in any way; it merely signals to the outside: "You don't need to know about this property."

In the following code, the properties `._counter` and `._action` are private.

```
class Countdown {
  constructor(counter, action) {
    this._counter = counter;
```

```
      this._action = action;
    }
    dec() {
      this._counter--;
      if (this._counter === 0) {
        this._action();
      }
    }
  }
}

// The two properties aren't really private:
assert.deepEqual(
  Object.keys(new Countdown()),
  ['_counter', '_action']);
```

With this technique, we don't get any protection and private names can clash. On the plus side, it is easy to use.

Private methods work similarly: They are normal methods whose names start with underscores.

ES6 and later: private instance data via WeakMaps

We can also manage private instance data via WeakMaps:

```
const _counter = new WeakMap();
const _action = new WeakMap();

class Countdown {
  constructor(counter, action) {
    _counter.set(this, counter);
    _action.set(this, action);
  }
  dec() {
    let counter = _counter.get(this);
    counter--;
    _counter.set(this, counter);
    if (counter === 0) {
      _action.get(this)();
    }
  }
}

// The two pseudo-properties are truly private:
assert.deepEqual(
  Object.keys(new Countdown()),
  []);
```

How exactly that works is explained in the chapter on WeakMaps.

This technique offers us considerable protection from outside access and there can't be any

name clashes. But it is also more complicated to use.

We control the visibility of the pseudo-property `_superProp` by controlling who has access to it – for example: If the variable exists inside a module and isn't exported, everyone inside the module and no one outside the module can access it. In other words: The scope of privacy isn't the class in this case, it's the module. We could narrow the scope, though:

```
let Countdown;
{ // class scope
  const _counter = new WeakMap();
  const _action = new WeakMap();

  Countdown = class {
    // ···
  }
}
```

This technique doesn't really support private methods. But module-local functions that have access to `_superProp` are the next best thing:

```
const _counter = new WeakMap();
const _action = new WeakMap();

class Countdown {
  constructor(counter, action) {
    _counter.set(this, counter);
    _action.set(this, action);
  }
  dec() {
    privateDec(this);
  }
}

function privateDec(_this) { // (A)
  let counter = _counter.get(_this);
  counter--;
  _counter.set(_this, counter);
  if (counter === 0) {
    _action.get(_this)();
  }
}
```

Note that `this` becomes the explicit function parameter `_this` (line A).

31.5.4 Simulating protected visibility and friend visibility via WeakMaps (advanced)

As previously discussed, instance private fields are only visible inside their classes and not even in subclasses. Thus, there is no built-in way to get:

- Protected visibility: A class and all of its subclasses can access a piece instance data.

- Friend visibility: A class and its "friends" (designated functions, objects, or classes) can access a piece of instance data.

In the previous subsection, we simulated "module visibility" (everyone inside a module has access to a piece of instance data) via WeakMaps. Therefore:

- If we put a class and its subclasses into the same module, we get protected visibility.
- If we put a class and its friends into the same module, we get friend visibility.

The next example demonstrates protected visibility:

```
const _superProp = new WeakMap();
class SuperClass {
  constructor() {
    _superProp.set(this, 'superProp');
  }
}
class SubClass extends SuperClass {
  getSuperProp() {
    return _superProp.get(this);
  }
}
assert.equal(
  new SubClass().getSuperProp(),
  'superProp'
);
```

Subclassing via `extends` is explained later in this chapter.

31.6   Static members of classes

31.6.1   Static public methods and accessors

All members in the body of the following class declaration create so-called *static* properties – properties of `StaticClass` itself.

```
class StaticPublicMethodsClass {
  static staticMethod() {
    return 'staticMethod';
  }
  static get staticAccessor() {
    return 'staticGetter';
  }
  static set staticAccessor(value) {
    assert.equal(value, 'staticSetter');
  }
}
assert.equal(
  StaticPublicMethodsClass.staticMethod(), 'staticMethod'
);
assert.equal(
```

```
  StaticPublicMethodsClass.staticAccessor, 'staticGetter'
);
StaticPublicMethodsClass.staticAccessor = 'staticSetter';
```

All kinds of static public methods and accessors (advanced)

```
const accessorKey = Symbol('accessorKey');
const syncMethodKey = Symbol('syncMethodKey');
const syncGenMethodKey = Symbol('syncGenMethodKey');
const asyncMethodKey = Symbol('asyncMethodKey');
const asyncGenMethodKey = Symbol('asyncGenMethodKey');

class StaticPublicMethodsClass2 {
  // Identifier keys
  static get accessor() {}
  static set accessor(value) {}
  static syncMethod() {}
  static * syncGeneratorMethod() {}
  static async asyncMethod() {}
  static async * asyncGeneratorMethod() {}

  // Quoted keys
  static get 'an accessor'() {}
  static set 'an accessor'(value) {}
  static 'sync method'() {}
  static * 'sync generator method'() {}
  static async 'async method'() {}
  static async * 'async generator method'() {}

  // Computed keys
  static get [accessorKey]() {}
  static set [accessorKey](value) {}
  static [syncMethodKey]() {}
  static * [syncGenMethodKey]() {}
  static async [asyncMethodKey]() {}
  static async * [asyncGenMethodKey]() {}
}

// Quoted and computed keys are accessed via square brackets:
StaticPublicMethodsClass2['sync method']();
StaticPublicMethodsClass2[syncMethodKey]();
```

Quoted and computed keys can also be used in object literals:

- "Quoted keys in object literals" (§30.7.1)
- "Computed keys in object literals" (§30.7.2)

More information on accessors (defined via getters and/or setters), generators, async methods, and async generator methods:

- "Object literals: accessors" (§30.3.6)
- "Synchronous generators"
- "Async functions"
- "Asynchronous generators"

31.6.2 Static public fields [ES2022]

The following code demonstrates static public fields. `StaticPublicFieldClass` has three of them:

```
const computedFieldKey = Symbol('computedFieldKey');
class StaticPublicFieldClass {
  static identifierFieldKey = 1;
  static 'quoted field key' = 2;
  static [computedFieldKey] = 3;
}

assert.deepEqual(
  Reflect.ownKeys(StaticPublicFieldClass),
  [
    'length', // number of constructor parameters
    'name', // 'StaticPublicFieldClass'
    'prototype',
    'identifierFieldKey',
    'quoted field key',
    computedFieldKey,
  ],
);

assert.equal(StaticPublicFieldClass.identifierFieldKey, 1);
assert.equal(StaticPublicFieldClass['quoted field key'], 2);
assert.equal(StaticPublicFieldClass[computedFieldKey], 3);
```

31.6.3 Static private methods, accessors, and fields [ES2022]

The following class has two static private slots (line A and line B):

```
class StaticPrivateClass {
  // Declare and initialize
  static #staticPrivateField = 'hello'; // (A)
  static #twice() { // (B)
    const str = StaticPrivateClass.#staticPrivateField;
    return str + ' ' + str;
  }
  static getResultOfTwice() {
    return StaticPrivateClass.#twice();
  }
}
```

```
assert.deepEqual(
  Reflect.ownKeys(StaticPrivateClass),
  [
    'length', // number of constructor parameters
    'name', // 'StaticPublicFieldClass'
    'prototype',
    'getResultOfTwice',
  ],
);

assert.equal(
  StaticPrivateClass.getResultOfTwice(),
  'hello hello'
);
```

This is a complete list of all kinds of static private slots:

```
class MyClass {
  static #staticPrivateMethod() {}
  static * #staticPrivateGeneratorMethod() {}

  static async #staticPrivateAsyncMethod() {}
  static async * #staticPrivateAsyncGeneratorMethod() {}

  static get #staticPrivateAccessor() {}
  static set #staticPrivateAccessor(value) {}
}
```

31.6.4   Static initialization blocks in classes [ES2022]

To set up instance data via classes, we have two constructs:

- *Fields*, to create and optionally initialize instance data
- *Constructors*, blocks of code that are executed every time a new instance is created

For static data, we have:

- *Static fields*
- *Static blocks* that are executed when a class is created

The following code demonstrates static blocks (line A):

```
class Translator {
  static translations = {
    yes: 'ja',
    no: 'nein',
    maybe: 'vielleicht',
  };
  static englishWords = [];
  static germanWords = [];
  static { // (A)
    for (const [english, german] of Object.entries(this.translations)) {
```

```
      this.englishWords.push(english);
      this.germanWords.push(german);
    }
  }
}
```

We could also execute the code inside the static block after the class (at the top level). However, using a static block has two benefits:

- All class-related code is inside the class.
- The code in a static block has access to private slots.

Rules for static initialization blocks

The rules for how static initialization blocks work, are relatively simple:

- There can be more than one static block per class.
- The execution of static blocks is interleaved with the execution of static field initializers.
- The static members of a superclass are executed before the static members of a subclass.

The following code demonstrates these rules:

```
class SuperClass {
  static superField1 = console.log('superField1');
  static {
    assert.equal(this, SuperClass);
    console.log('static block 1 SuperClass');
  }
  static superField2 = console.log('superField2');
  static {
    console.log('static block 2 SuperClass');
  }
}

class SubClass extends SuperClass {
  static subField1 = console.log('subField1');
  static {
    assert.equal(this, SubClass);
    console.log('static block 1 SubClass');
  }
  static subField2 = console.log('subField2');
  static {
    console.log('static block 2 SubClass');
  }
}
```

Output:

```
superField1
static block 1 SuperClass
```

```
superField2
static block 2 SuperClass
subField1
static block 1 SubClass
subField2
static block 2 SubClass
```

Subclassing via extends is explained later in this chapter.

31.6.5 Pitfall: Using **this** to access static private fields

In static public members, we can access static public slots via `this`. Alas, we should not use it to access static private slots.

this and static public fields

Consider the following code:

```
class SuperClass {
  static publicData = 1;

  static getPublicViaThis() {
    return this.publicData;
  }
}
class SubClass extends SuperClass {
}
```

Subclassing via extends is explained later in this chapter.

Static public fields are properties. If we make the method call

```
assert.equal(SuperClass.getPublicViaThis(), 1);
```

then `this` points to `SuperClass` and everything works as expected. We can also invoke `.getPublicViaThis()` via the subclass:

```
assert.equal(SubClass.getPublicViaThis(), 1);
```

SubClass inherits `.getPublicViaThis()` from its prototype `SuperClass`. `this` points to Sub-Class and things continue to work, because `SubClass` also inherits the property `.public-Data`.

As an aside, if we assigned to `this.publicData` in `getPublicViaThis()` and invoked it via `SubClass.getPublicViaThis()`, then we would create a new own property of `SubClass` that (non-destructively) overrides the property inherited from `SuperClass`.

this and static private fields

Consider the following code:

```
class SuperClass {
  static #privateData = 2;
  static getPrivateDataViaThis() {
```

```
    return this.#privateData;
  }
  static getPrivateDataViaClassName() {
    return SuperClass.#privateData;
  }
}
class SubClass extends SuperClass {
}
```

Invoking `.getPrivateDataViaThis()` via `SuperClass` works, because `this` points to `Super-Class`:

```
  assert.equal(SuperClass.getPrivateDataViaThis(), 2);
```

However, invoking `.getPrivateDataViaThis()` via `SubClass` does not work, because `this` now points to `SubClass` and `SubClass` has no static private field `.#privateData` (private slots in prototype chains are not inherited):

```
  assert.throws(
    () => SubClass.getPrivateDataViaThis(),
    {
      name: 'TypeError',
      message: 'Cannot read private member #privateData from'
        + ' an object whose class did not declare it',
    }
  );
```

The workaround is to accesss `.#privateData` directly, via `SuperClass`:

```
  assert.equal(SubClass.getPrivateDataViaClassName(), 2);
```

With static private methods, we are facing the same issue.

31.6.6 All members (static, prototype, instance) can access all private members

Every member inside a class can access all other members inside that class – both public and private ones:

```
class DemoClass {
  static #staticPrivateField = 1;
  #instPrivField = 2;

  static staticMethod(inst) {
    // A static method can access static private fields
    // and instance private fields
    assert.equal(DemoClass.#staticPrivateField, 1);
    assert.equal(inst.#instPrivField, 2);
  }

  protoMethod() {
    // A prototype method can access instance private fields
```

```
      // and static private fields
      assert.equal(this.#instPrivField, 2);
      assert.equal(DemoClass.#staticPrivateField, 1);
    }
  }
```

In contrast, no one outside can access the private members:

```
  // Accessing private fields outside their classes triggers
  // syntax errors (before the code is even executed).
  assert.throws(
    () => eval('DemoClass.#staticPrivateField'),
    {
      name: 'SyntaxError',
      message: "Private field '#staticPrivateField' must"
        + " be declared in an enclosing class",
    }
  );
  // Accessing private fields outside their classes triggers
  // syntax errors (before the code is even executed).
  assert.throws(
    () => eval('new DemoClass().#instPrivField'),
    {
      name: 'SyntaxError',
      message: "Private field '#instPrivField' must"
        + " be declared in an enclosing class",
    }
  );
```

31.6.7   Static private methods and data before ES2022

The following code only works in ES2022 – due to every line that has a hash symbol (#) in it:

```
  class StaticClass {
    static #secret = 'Rumpelstiltskin';
    static #getSecretInParens() {
      return `(${StaticClass.#secret})`;
    }
    static callStaticPrivateMethod() {
      return StaticClass.#getSecretInParens();
    }
  }
```

Since private slots only exist once per class, we can move #secret and #getSecretInParens to the scope surrounding the class and use a module to hide them from the world outside the module.

```
  const secret = 'Rumpelstiltskin';
  function getSecretInParens() {
    return `(${secret})`;
```

```
  }

  // Only the class is accessible outside the module
  export class StaticClass {
    static callStaticPrivateMethod() {
      return getSecretInParens();
    }
  }
```

31.6.8 Static factory methods

Sometimes there are multiple ways in which a class can be instantiated. Then we can implement *static factory methods* such as `Point.fromPolar()`:

```
  class Point {
    static fromPolar(radius, angle) {
      const x = radius * Math.cos(angle);
      const y = radius * Math.sin(angle);
      return new Point(x, y);
    }
    constructor(x=0, y=0) {
      this.x = x;
      this.y = y;
    }
  }

  assert.deepEqual(
    Point.fromPolar(13, 0.39479111969976155),
    new Point(12, 5)
  );
```

I like how descriptive static factory methods are: `fromPolar` describes how an instance is created. JavaScript's standard library also has such factory methods – for example:

- `Array.from()`
- `Object.create()`

I prefer to either have no static factory methods or *only* static factory methods. Things to consider in the latter case:

- One factory method will probably directly call the constructor (but have a descriptive name).
- We need to find a way to prevent the constructor being called from outside.

In the following code, we use a secret token (line A) to prevent the constructor being called from outside the current module.

```
  // Only accessible inside the current module
  const secretToken = Symbol('secretToken'); // (A)

  export class Point {
    static create(x=0, y=0) {
```

```
      return new Point(secretToken, x, y);
    }
    static fromPolar(radius, angle) {
      const x = radius * Math.cos(angle);
      const y = radius * Math.sin(angle);
      return new Point(secretToken, x, y);
    }
    constructor(token, x, y) {
      if (token !== secretToken) {
        throw new TypeError('Must use static factory method');
      }
      this.x = x;
      this.y = y;
    }
  }
Point.create(3, 4); // OK
assert.throws(
  () => new Point(3, 4),
  TypeError
);
```

31.7   Subclassing

Classes can also extend existing classes. For example, the following class `Employee` extends
`Person`:

```
  class Person {
    #firstName;
    constructor(firstName) {
      this.#firstName = firstName;
    }
    describe() {
      return `Person named ${this.#firstName}`;
    }
    static extractNames(persons) {
      return persons.map(person => person.#firstName);
    }
  }

  class Employee extends Person {
    constructor(firstName, title) {
      super(firstName);
      this.title = title;
    }
    describe() {
      return super.describe() +
        ` (${this.title})`;
    }
```

```
  }

  const jane = new Employee('Jane', 'CTO');
  assert.equal(
    jane.title,
    'CTO'
  );
  assert.equal(
    jane.describe(),
    'Person named Jane (CTO)'
  );
```

Terminology related to extending:

- Another word for *extending* is *subclassing*.
- `Person` is the superclass of `Employee`.
- `Employee` is the subclass of `Person`.
- A *base class* is a class that has no superclasses.
- A *derived class* is a class that has a superclass.

Inside the `.constructor()` of a derived class, we must call the super-constructor via `super()` before we can access `this`. Why is that?

Let's consider a chain of classes:

- Base class `A`
- Class `B` extends `A`.
- Class `C` extends `B`.

If we invoke `new C()`, `C`'s constructor super-calls `B`'s constructor which super-calls `A`'s constructor. Instances are always created in base classes, before the constructors of subclasses add their slots. Therefore, the instance doesn't exist before we call `super()` and we can't access it via `this`, yet.

Note that static public slots are inherited. For example, `Employee` inherits the static method `.extractNames()`:

```
  > 'extractNames' in Employee
  true
```

> **Exercise: Subclassing**
>
> `exercises/classes/color_point_class_test.mjs`

31.7.1 The internals of subclassing (advanced)

The classes `Person` and `Employee` from the previous section are made up of several objects (figure 31.4). One key insight for understanding how these objects are related is that there are two prototype chains:

- The instance prototype chain, on the right.
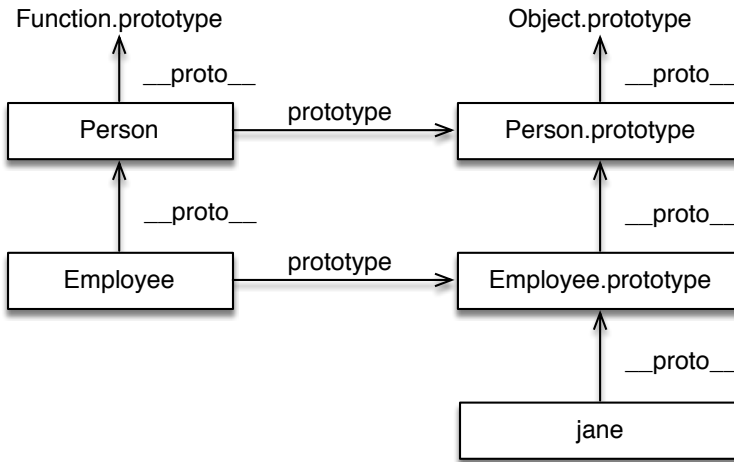
Figure 31.4: These are the objects that make up class `Person` and its subclass, `Employee`. The left column is about classes. The right column is about the `Employee` instance `jane` and its prototype chain.

- The class prototype chain, on the left.

The instance prototype chain (right column)

The instance prototype chain starts with `jane` and continues with `Employee.prototype` and `Person.prototype`. In principle, the prototype chain ends at this point, but we get one more object: `Object.prototype`. This prototype provides services to virtually all objects, which is why it is included here, too:

```
> Object.getPrototypeOf(Person.prototype) === Object.prototype
true
```

The class prototype chain (left column)

In the class prototype chain, `Employee` comes first, `Person` next. Afterward, the chain continues with `Function.prototype`, which is only there because `Person` is a function and functions need the services of `Function.prototype`.

```
> Object.getPrototypeOf(Person) === Function.prototype
true
```

31.7.2   `instanceof` and subclassing (advanced)

We have not yet learned how `instanceof` really works. How does `instanceof` determine if a value `x` is an instance of a class `C` (it can be a direct instance of `C` or a direct instance of a subclass of `C`)? It checks 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 figure 31.4, we can confirm that the prototype chain does lead us to the following correct answers:

```
> jane instanceof Employee
true
> jane instanceof Person
true
> jane instanceof Object
true
```

Note that `instanceof` always returns `false` if its self-hand side is a primitive value:

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

31.7.3   Not all objects are instances of `Object` (advanced)

An object (a non-primitive value) is only an instance of `Object` if `Object.prototype` is in its prototype chain (see previous subsection). Virtually all objects are instances of `Object` – for example:

```
assert.equal(
  {a: 1} instanceof Object, true
);
assert.equal(
  ['a'] instanceof Object, true
);
assert.equal(
  /abc/g instanceof Object, true
);
assert.equal(
  new Map() instanceof Object, true
);

class C {}
assert.equal(
  new C() instanceof Object, true
);
```

In the next example, `obj1` and `obj2` are both objects (line A and line C), but they are not instances of `Object` (line B and line D): `Object.prototype` is not in their prototype chains because they don't have any prototypes.

```
const obj1 = {__proto__: null};
assert.equal(
  typeof obj1, 'object' // (A)
);
assert.equal(
  obj1 instanceof Object, false // (B)
```

```
);

const obj2 = Object.create(null);
assert.equal(
  typeof obj2, 'object' // (C)
);
assert.equal(
  obj2 instanceof Object, false // (D)
);
```

Object.prototype is the object that ends most prototype chains. Its prototype is null, which means it isn't an instance of Object either:

```
> typeof Object.prototype
'object'
> Object.getPrototypeOf(Object.prototype)
null
> Object.prototype instanceof Object
false
```

31.7.4   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.

The prototype chain of {}

Let's start by examining plain objects:

```
> p({}) === Object.prototype
true
> p(p({})) === null
true
```
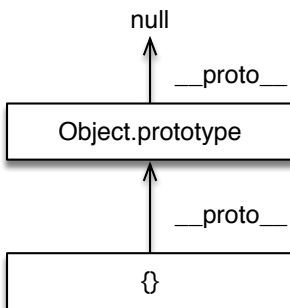

Figure 31.5: The prototype chain of an object created via an object literal starts with that object, continues with Object.prototype, and ends with null.

Figure 31.5 shows a diagram for this prototype chain. We can see that {} really is an instance of `Object` – `Object.prototype` is in its prototype chain.

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
```
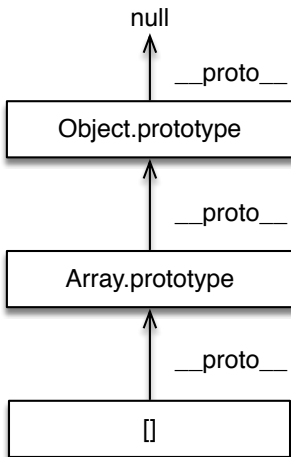

Figure 31.6: The prototype chain of an Array has these members: the Array instance, `Array.prototype`, `Object.prototype`, `null`.

This prototype chain (visualized in figure 31.6) tells us that an Array object is an instance of `Array` and of `Object`.

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
```

The prototype chains of built-in classes

The prototype of a base class is `Function.prototype` which means that it is a function (an instance of `Function`):

```
class A {}
assert.equal(
  Object.getPrototypeOf(A),
  Function.prototype
);

assert.equal(
  Object.getPrototypeOf(class {}),
  Function.prototype
);
```

The prototype of a derived class is its superclass:

```
class B extends A {}
assert.equal(
  Object.getPrototypeOf(B),
  A
);

assert.equal(
  Object.getPrototypeOf(class extends Object {}),
  Object
);
```

Interestingly, `Object`, `Array`, and `Function` are all base classes:

```
> Object.getPrototypeOf(Object) === Function.prototype
true
> Object.getPrototypeOf(Array) === Function.prototype
true
> Object.getPrototypeOf(Function) === Function.prototype
true
```

However, as we have seen, even the instances of base classes have `Object.prototype` in their prototype chains because it provides services that all objects need.

⚙️ **Why are `Array` and `Function` base classes?**

Base classes are where instances are actually created. Both `Array` and `Function` need to create their own instances because they have so-called "internal slots" which can't be added later to instances created by `Object`.

31.7.5   Mixin classes (advanced)

JavaScript's class system only supports *single inheritance*. That is, each class can have at most one superclass. One way around this limitation is via a technique called *mixin classes* (short: *mixins*).

The idea is as follows: Let's say we want a class `C` to inherit from two superclasses `S1` and `S2`. That would be *multiple inheritance*, which JavaScript doesn't support.

Our workaround is to turn `S1` and `S2` into *mixins*, factories for subclasses:

```
const S1 = (Sup) => class extends Sup { /*···*/ };
const S2 = (Sup) => class extends Sup { /*···*/ };
```

Each of these two functions returns a class that extends a given superclass `Sup`. We create class `C` as follows:

```
class C extends S2(S1(Object)) {
  /*...*/
}
```

We now have a class `C` that extends the class returned by `S2()` which extends the class returned by `S1()` which extends `Object`.

Example: a mixin for name management

We implement a mixin `Named` adds a property `.name` and a method `.toString()` to its superclass:

```
const Named = (Sup) => class extends Sup {
  name = '(Unnamed)';
  toString() {
    const className = this.constructor.name;
    return `${className} named ${this.name}`;
  }
};
```

We use this mixin to implement a class `City` that has a name:

```
class City extends Named(Object) {
  constructor(name) {
    super();
    this.name = name;
  }
}
```

The following code confirms that the mixin works:

```
const paris = new City('Paris');
assert.equal(
  paris.name, 'Paris'
);
assert.equal(
  paris.toString(), 'City named Paris'
);
```

The benefits of mixins

Mixins free us from the constraints of single inheritance:

- The same class can extend a single superclass and zero or more mixins.
- The same mixin can be used by multiple classes.

31.8 The methods and accessors of `Object.prototype` (advanced)

31.8.1 Quick reference: `Object.prototype.*`

As we have seen in "Not all objects are instances of `Object`" (§31.7.3), almost all objects are instances of `Object`. This class provides useful functionality to its instances:

- Configuring how objects are converted to primitive values (e.g. by the + operator): The following methods have default implementations but are often overridden in subclasses or instances.
  - `.toString()`: Configures how an object is converted to a string.
  - `.toLocaleString()`: A version of `.toString()` that can be configured in various ways via arguments (language, region, etc.).
  - `.valueOf()`: Configures how an object is converted to a non-string primitive value (often a number).
- Useful methods (with pitfalls – see next subsection):
  - `.isPrototypeOf()`: Is the receiver in the prototype chain of a given object?
  - `.propertyIsEnumerable()`: Does the receiver have an enumerable own property with the given key?
- Avoid these features (there are better alternatives):
  - `.__proto__`: Get and set the prototype of the receiver.
    * Using this accessor is not recommended. Alternatives:
      · `Object.getPrototypeOf()`
      · `Object.setPrototypeOf()`
  - `.hasOwnProperty()`: Does the receiver have an own property with a given key?
    * Using this method is not recommended. Alternative in ES2022 and later: `Object.hasOwn()`.

Before we take a closer look at each of these features, we'll learn about an important pitfall (and how to work around it): We can't use the features of `Object.prototype` with all objects.

31.8.2 Using `Object.prototype` methods safely

Invoking one of the methods of `Object.prototype` on an arbitrary object doesn't always work. To illustrate why, we use method `Object.prototype.hasOwnProperty`, which returns `true` if an object has an own property with a given key:

```
> {ownProp: true}.hasOwnProperty('ownProp')
true
> {ownProp: true}.hasOwnProperty('abc')
false
```

Invoking `.hasOwnProperty()` on an arbitrary object can fail in two ways. On one hand, this method isn't available if an object is not an instance of `Object` (see "Not all objects are instances of `Object`" (§31.7.3)):

```
const obj = Object.create(null);
assert.equal(obj instanceof Object, false);
assert.throws(
```

```
  () => obj.hasOwnProperty('prop'),
  {
    name: 'TypeError',
    message: 'obj.hasOwnProperty is not a function',
  }
);
```

On the other hand, we can't use `.hasOwnProperty()` if an object overrides it with an own property (line A):

```
const obj = {
  hasOwnProperty: 'yes' // (A)
};
assert.throws(
  () => obj.hasOwnProperty('prop'),
  {
    name: 'TypeError',
    message: 'obj.hasOwnProperty is not a function',
  }
);
```

There is, however, a safe way to use `.hasOwnProperty()`:

```
function hasOwnProp(obj, propName) {
  return Object.prototype.hasOwnProperty.call(obj, propName); // (A)
}
assert.equal(
  hasOwnProp(Object.create(null), 'prop'), false
);
assert.equal(
  hasOwnProp({hasOwnProperty: 'yes'}, 'prop'), false
);
assert.equal(
  hasOwnProp({hasOwnProperty: 'yes'}, 'hasOwnProperty'), true
);
```

The method invocation in line A is explained in "Dispatched vs. direct method calls" (§31.3.5).

We can also use `.bind()` to implement `hasOwnProp()`:

```
const hasOwnProp = Function.prototype.call
  .bind(Object.prototype.hasOwnProperty);
```

How does this code work? In line A in the example before the code above, we used the function method `.call()` to turn the function `hasOwnProperty` with one implicit parameter (`this`) and one explicit parameter (`propName`) into a function that has two explicit parameters (`obj` and `propName`).

In other words – method `.call()` invokes the function `f` referred to by its receiver (`this`):

- The first (explicit) parameter of `.call()` becomes the `this` of `f`.
- The second (explicit) parameter of `.call()` becomes the first argument of `f`.

- Etc.

We use `.bind()` to create a version `.call()` whose `this` always refers to `Object.prototype.hasOwnProperty`. That new version invokes `.hasOwnProperty()` in the same manner as we did in line A – which is what we want.

> ⑦ **Is it never OK to use `Object.prototype` methods via dynamic dispatch?**
>
> In some cases we can be lazy and call `Object.prototype` methods like normal methods: If we know the receivers and they are fixed-layout objects.
> If, on the other hand, we don't know their receivers and/or they are dictionary objects, then we need to take precautions.

31.8.3 `Object.prototype.toString()`

By overriding `.toString()` (in a subclass or an instance), we can configure how objects are converted to strings:

```
> String({toString() { return 'Hello!' }})
'Hello!'
> String({})
'[object Object]'
```

For converting objects to strings it's better to use `String()` because that also works with `undefined` and `null`:

```
> undefined.toString()
TypeError: Cannot read properties of undefined (reading 'toString')
> null.toString()
TypeError: Cannot read properties of null (reading 'toString')
> String(undefined)
'undefined'
> String(null)
'null'
```

31.8.4 `Object.prototype.toLocaleString()`

`.toLocaleString()` is a version of `.toString()` that can be configured via a locale and often additional options. Any class or instance can implement this method. In the standard library, the following classes do:

- `Array.prototype.toLocaleString()`
- `Number.prototype.toLocaleString()`
- `Date.prototype.toLocaleString()`
- `TypedArray.prototype.toLocaleString()`
- `BigInt.prototype.toLocaleString()`

As an example, this is how numbers with decimal fractions are converted to string differently, depending on locale (`'fr'` is French, `'en'` is English):

```
> 123.45.toLocaleString('fr')
'123,45'
> 123.45.toLocaleString('en')
'123.45'
```

31.8.5 `Object.prototype.valueOf()`

By overriding `.valueOf()` (in a subclass or an instance), we can configure how objects are converted to non-string values (often numbers):

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

31.8.6 `Object.prototype.isPrototypeOf()`

`proto.isPrototypeOf(obj)` returns `true` if `proto` is in the prototype chain of `obj` and `false` otherwise.

```
const a = {};
const b = {__proto__: a};
const c = {__proto__: b};

assert.equal(a.isPrototypeOf(b), true);
assert.equal(a.isPrototypeOf(c), true);

assert.equal(a.isPrototypeOf(a), false);
assert.equal(c.isPrototypeOf(a), false);
```

This is how to use this method safely (for details see "Using `Object.prototype` methods safely" (§31.8.2)):

```
const obj = {
  // Overrides Object.prototype.isPrototypeOf
  isPrototypeOf: true,
};
// Doesn't work in this case:
assert.throws(
  () => obj.isPrototypeOf(Object.prototype),
  {
    name: 'TypeError',
    message: 'obj.isPrototypeOf is not a function',
  }
);
// Safe way of using .isPrototypeOf():
assert.equal(
  Object.prototype.isPrototypeOf.call(obj, Object.prototype), false
);
```

31.8.7 `Object.prototype.propertyIsEnumerable()`

`obj.propertyIsEnumerable(propKey)` returns `true` if `obj` has an own enumerable property whose key is `propKey` and `false` otherwise.

```
const proto = {
  enumerableProtoProp: true,
};
const obj = {
  __proto__: proto,
  enumerableObjProp: true,
  nonEnumObjProp: true,
};
Object.defineProperty(
  obj, 'nonEnumObjProp',
  {
    enumerable: false,
  }
);

assert.equal(
  obj.propertyIsEnumerable('enumerableProtoProp'),
  false // not an own property
);
assert.equal(
  obj.propertyIsEnumerable('enumerableObjProp'),
  true
);
assert.equal(
  obj.propertyIsEnumerable('nonEnumObjProp'),
  false // not enumerable
);
assert.equal(
  obj.propertyIsEnumerable('unknownProp'),
  false // not a property
);
```

This is how to use this method safely (for details see "Using `Object.prototype` methods safely" (§31.8.2)):

```
const obj = {
  // Overrides Object.prototype.propertyIsEnumerable
  propertyIsEnumerable: true,
  enumerableProp: 'yes',
};
// Doesn't work in this case:
assert.throws(
  () => obj.propertyIsEnumerable('enumerableProp'),
  {
    name: 'TypeError',
```

```
      message: 'obj.propertyIsEnumerable is not a function',
  }
);
// Safe way of using .propertyIsEnumerable():
assert.equal(
  Object.prototype.propertyIsEnumerable.call(obj, 'enumerableProp'),
  true
);
```

Another safe alternative is to use property descriptors:

```
assert.deepEqual(
  Object.getOwnPropertyDescriptor(obj, 'enumerableProp'),
  {
    value: 'yes',
    writable: true,
    enumerable: true,
    configurable: true,
  }
);
```

31.8.8 `Object.prototype.__proto__` (accessor)

Property __proto__ exists in two versions:

- An accessor that all instances of `Object` have.
- A property of object literals that sets the prototypes of the objects created by them.

I recommend to avoid the former feature:

- As explained in "Using `Object.prototype` methods safely" (§31.8.2), it doesn't work with all objects.
- The ECMAScript specification has deprecated it and calls it "optional" and "legacy".

In contrast, __proto__ in object literals always works and is not deprecated.

Read on if you are interested in how the accessor __proto__ works.

__proto__ is an accessor of `Object.prototype` that is inherited by all instances of `Object`. Implementing it via a class would look like this:

```
class Object {
  get __proto__() {
    return Object.getPrototypeOf(this);
  }
  set __proto__(other) {
    Object.setPrototypeOf(this, other);
  }
  // ···
}
```

Since __proto__ is inherited from `Object.prototype`, we can remove this feature by creating an object that doesn't have `Object.prototype` in its prototype chain (see "Not all

objects are instances of `Object`" (§31.7.3)):

```
> '__proto__' in {}
true
> '__proto__' in Object.create(null)
false
```

31.8.9 `Object.prototype.hasOwnProperty()`

> ⚠️ **Better alternative to `.hasOwnProperty()`: `Object.hasOwn()`** [ES2022]
>
> See "`Object.hasOwn()`: Is a given property own (non-inherited)?" (§30.10.4).

`obj.hasOwnProperty(propKey)` returns `true` if `obj` has an own (non-inherited) property whose key is `propKey` and `false` otherwise.

```
const obj = { ownProp: true };
assert.equal(
  obj.hasOwnProperty('ownProp'), true // own
);
assert.equal(
  'toString' in obj, true // inherited
);
assert.equal(
  obj.hasOwnProperty('toString'), false
);
```

This is how to use this method safely (for details see "Using `Object.prototype` methods safely" (§31.8.2)):

```
const obj = {
  // Overrides Object.prototype.hasOwnProperty
  hasOwnProperty: true,
};
// Doesn't work in this case:
assert.throws(
  () => obj.hasOwnProperty('anyPropKey'),
  {
    name: 'TypeError',
    message: 'obj.hasOwnProperty is not a function',
  }
);
// Safe way of using .hasOwnProperty():
assert.equal(
  Object.prototype.hasOwnProperty.call(obj, 'anyPropKey'), false
);
```

31.9 FAQ: classes

31.9.1 Why are they called "instance private fields" in this book and not "private instance fields"?

That is done to highlight how different properties (public slots) and private slots are: By changing the order of the adjectives, the words "public" and "field" and the words "private" and "field" are always mentioned together.

31.9.2 Why the identifier prefix #? Why not declare private fields via `private`?

Could private fields be declared via `private` and use normal identifiers? Let's examine what would happen if that were possible:

```
class MyClass {
  private value; // (A)
  compare(other) {
    return this.value === other.value;
  }
}
```

Whenever an expression such as `other.value` appears in the body of `MyClass`, JavaScript has to decide:

- Is `.value` a property?
- Is `.value` a private field?

At compile time, JavaScript doesn't know if the declaration in line A applies to `other` (due to it being an instance of `MyClass`) or not. That leaves two options for making the decision:

1. `.value` is always interpreted as a private field.
2. JavaScript decides at runtime:
   - If `other` is an instance of `MyClass`, then `.value` is interpreted as a private field.
   - Otherwise `.value` is interpreted as a property.

Both options have downsides:

- With option (1), we can't use `.value` as a property, anymore – for any object.
- With option (2), performance is affected negatively.

That's why the name prefix `#` was introduced. The decision is now easy: If we use `#`, we want to access a private field. If we don't, we want to access a property.

`private` works for statically typed languages (such as TypeScript) because they know at compile time if `other` is an instance of `MyClass` and can then treat `.value` as private or public.

Chapter 32

Where are the remaining chapters?

You are reading a preview version of this book. You can either read all chapters online or you can buy the full version.