

What you need to know about this book

This book is about ECMAScript 6 (whose official name is ECMAScript 2015), a new version of JavaScript.

Audience: JavaScript programmers

In order to understand this book, you should already know JavaScript. If you don't: my other book "[Speaking JavaScript](#)" is free online and teaches programmers all of JavaScript (up to and including ECMAScript 5).

Why should I read this book?

- **You decide how deep to go:** This book covers ECMAScript 6 in depth, but is structured so that you can also quickly get an overview if you want to.
- **Not just "what", also "why":** This book not only tells you how ES6 works, it also tells you why it works the way it does.
- **Thoroughly researched:** In order to make sense of ES6, I have consulted many sources:
 - The language specification (to which you'll occasionally find pointers in this book)
 - The es-discuss mailing list
 - The TC39 meeting notes
 - Scientific papers
 - Documentation on features in other languages that inspired ES6 features
 - And more

How to read this book

This book covers ES6 with three levels of detail:

- **Quick start:** Begin with the chapter "[Core ES6 features](#)". Additionally, almost every chapter starts with a section giving an overview of what's in the chapter. [The last chapter](#) collects all of these overview sections in a single location.
- **Solid foundation:** Each chapter always starts with the essentials and then increasingly goes into details. The headings should give you a good idea of when to stop reading, but I also occasionally give tips in sidebars w.r.t. how important it is to know something.
- **In-depth knowledge:** Read all of a chapter, including the in-depth parts.

Other things to know:

- **Recommendations:** I occasionally recommend simple rules. Those are meant as guidelines, to keep you safe without you having to know (or remember) all of the details. I tend to favor mainstream over elegance, because most code doesn't exist in a vacuum. However, I'll always give you enough information so that you can make up your own mind.
- **Forum:** The "Exploring ES6" homepage [links to a forum](#) where you can discuss questions and ideas related to this book.
- **Errata (typos, errors, etc.):** On [the "Exploring ES6" homepage](#), there are links to a form for submitting errata and to a list with submitted errata.

Sources of this book

I started writing this book long before there were implementations of ES6 features, which required quite a bit of research. Essential sources were:

- [The es-discuss mailing list](#)
- [TC39 meeting notes](#)
- [The ECMAScript language specification](#)
- [The old ECMAScript Harmony wiki](#)
- Scientific papers (e.g. the ones written about ES6 proxies) and other material on the web.
- Asking around to fill remaining holes (the people who answered are acknowledged throughout the book)

Glossary

Strict mode versus sloppy mode

ECMAScript 5 introduced language modes: *Strict mode* makes JavaScript a cleaner language by changing its semantics, performing more checks and throwing more exceptions. Consult Sect. "[Strict Mode](#)" in "[Speaking JavaScript](#)" for more information. The legacy/default mode is called *non-strict mode* or *sloppy mode*.

Strict mode is switched on via the following line (which does nothing in ECMAScript versions before ES5):

```
'use strict';
```

If you put this line at the beginning of a file, all code in it is in strict mode. If you make this line the first line of a function, only that function is in strict mode.

Using a directive to switch on strict mode is not very user friendly and was one of the reasons why strict mode was not nearly as popular in ES5 as it should be. However, ES6 modules and classes are implicitly in strict mode. Given that most ES6 code will live in modules, strict mode becomes the de-facto default for ES6.

Protocol

The term *protocol* has various meanings in computing. In the context of programming languages and API design, I'm using it as follows:

A protocol defines interfaces (signatures for methods and/or functions) and rules for using them.

The idea is to specify how a service is to be performed. Then anyone can perform the service and anyone can request it and they are guaranteed to work together well.

Note that the definition given here is different from viewing a protocol as an interface (as, for example, Objective C does), because this definition includes rules.

Receiver (of a method call)

Given a method call `obj.m(...)`, `obj` is the *receiver* of the method call and accessible via `this` inside the method.

Signature of a function (or a method)

The (type) signature of a function describes how the function is to be called, what its inputs and its output are. I'm using the syntax established by Microsoft TypeScript and Facebook Flow in this book. An example of a signature:

```
parseInt(string : string, radix? : number) : number
```

You can see that `parseInt()` expects a string and a number and returns a number. If the type of a parameter is clear, I often omit the type annotation.

Internal properties

The ES6 specification uses *internal properties* to describe how JavaScript works. These are only known to the spec and not accessible from JavaScript. They may or may not exist in an actual implementation of the language. The names of internal properties are written in double square brackets.

For example: the link between an object and its prototype is the internal property `[[Prototype]]`. The value of that property cannot be read directly via JavaScript, but you can use `Object.getPrototypeOf()` to do so.

Bindings and environments

The ECMAScript spec uses a data structure called *environment* to store the variables of a scope. An environment is basically a dictionary that maps variable names to values. A *binding* is an entry in an environment, storage space for a variable.

Destructive operations

Destructive operations (methods, functions) modify their parameters or their receivers. For example, `push()` modifies its receiver `arr`:

```
> const arr = ['a', 'b'];
> arr.push('c')
3
> arr
[ 'a', 'b', 'c' ]
```

In contrast, `concat()` creates a new Array and does not change its receiver `arr`:

```
> const arr = ['a', 'b'];
> arr.concat(['c'])
[ 'a', 'b', 'c' ]
> arr
[ 'a', 'b' ]
```

Conventions

Documenting classes

The API of a class `c` is usually documented as follows:

- `c` constructor
- Static `c` methods
- `C.prototype` methods

Capitalization

In English, I capitalize JavaScript terms as follows:

- The names of primitive entities are not capitalized: a boolean value, a number value, a symbol, a string. One reason why I'm doing this is because TypeScript and Flow distinguish:
 - The type `String`: its members are objects, instances of `String`.
 - The type `string`: its members are primitive values, strings.
- The data structure `Map` is capitalized. Rationale: distinguish from the `Array` method `map()`.
- The data structure `Set` is capitalized. Rationale: distinguish from the verb `set`.
- `Array` and `Promise` are capitalized. Rationale: easy to confuse with English words.
- Not capitalized (for now): `object`, `generator`, `proxy`.

Demo code on GitHub

Several repositories on GitHub contain code shown in this book:

- [async-examples](#)
- [babel-on-node](#)
- [demo_promise](#)
- [generator-examples](#)
- [node-es6-demo](#)
- [promise-examples](#)
- [webpack-es6-demo](#)

Sidebars

Sidebars are boxes of text marked with icons. They complement the normal content.



Tips for reading

Gives you tips for reading (what content to skip etc.).



Code on GitHub

Tells you where you can download demo code shown in this book.



Information

General information.



Question

Asks and answers a question, in FAQ style.



Warning

Things you need to be careful about.



External material

Points to related material hosted somewhere on the web.



Related parts of the spec

Explains where in the ES6 spec you can find the feature that is currently being explained.

Footnotes

Occasionally, I refer to (publicly available) external material via footnotes. Two sources are marked with a prefix in square brackets:

- [Spec] refers to content in the HTML version of the ES6 spec.
- [Speaking JS] refers to content in the HTML version of “Speaking JavaScript”.

Foreword

Edge cases! My life as the project editor of the ES6 specification has been all about edge cases. Like most software, the design of a programming language feature is typically driven by specific use cases. But programmers can and often do use language features in novel ways that are well outside the scope of those original use cases. In addition, no language feature stands alone. Every feature potentially interacts with every other feature. Those unexpected uses and feature interactions are the realm of edge cases.

For example, consider a function that has a parameter default value initialization expression that uses the `eval` function to first declare a local variable that has the same name as a local variable declared in the function body and then returns, as the parameter value, an arrow function that references that name. What happens if code in the function body accesses the parameter value and calls the arrow function? Which variable is accessed? Is there an error that should be detected and reported? It's edge cases like this that kept me up at night while ES6 was being designed.

A good language design must at least consider such edge cases. The specification of a massively popular language that will have multiple implementations must pin down what happens for all the edge cases. Otherwise, different implementations of the language will handle edge cases differently and programs won't work the same everywhere.

If you really want to understand ES6, you have to understand how each feature works, even when you're dealing with unusual situations and edge cases. What sets Axel Rauschmayer's *Exploring ES6* apart from other books is that it really cares about the inner workings of ECMAScript. It doesn't just describe the common use cases that you probably already understand. It digs deep into the semantics and, where necessary, wallows in the edge cases. It explains why features work the way that they work and how they are used in realistic code. Assimilate the material in this book and you will be an ES6 expert.

Allen Wirfs-Brock
ECMAScript 2015 (ES6) Specification Editor

Preface

You are reading a book about ECMAScript 6 (ES6), a new version of JavaScript. It's great that we can finally use that version, which had a long and eventful past: It was first conceived as ECMAScript 4, a successor to ECMAScript 3 (whose release was in December 1999). In July 2008, plans changed and the next versions of JavaScript were to be first a small incremental release (which became ES5) and then a larger, more powerful release. The latter had the code name *Harmony* and part of it became ES6.

ECMAScript 5 was standardized in December 2009. I first heard and [blogged](#) about ECMAScript 6 in January 2011, when it was still called *Harmony*. The original plan was to finish ES6 in 2013, but things took longer and it was standardized in June 2015. (A more detailed account of ES6's history is given in [the next chapter](#).)

With a few minor exceptions, I am happy how ECMAScript 6 turned out. This book describes my experiences with, and my research of, its features. Similarly to ES6, it took a long time to finish – in a way, I started writing it in early 2011. Like my previous book [“Speaking JavaScript”](#), I wrote most of it as a series of blog posts. I like the discussion and feedback that this open process enables, which is why this book is available for free online.

This book can be read online for free. If you find it useful, please support it by [buying a copy](#). You'll get DRM-free PDF, EPUB, MOBI files.

I hope that reading the book conveys some of the fun I had investigating and playing with ES6.

Axel Rauschmayer

Acknowledgements

I owe thanks to the many people who have – directly or indirectly – contributed to this book; by answering questions, pointing out bugs in blog posts, etc.:

Jake Archibald, André Bargull, Guy Bedford, James Burke, Mathias Bynens, Raymond Camden, Domenic Denicola, Brendan Eich, Eric Elliott, Michael Ficarra, Aaron Frost, Andrea Giammarchi, Jaydson Gomes, Jordan Harband, David Herman, James Kyle, Russell Leggett, Dmitri Lomov, Sebastian McKenzie, Calvin Metcalf, Mark S. Miller, Alan Norbauer, Mariusz Novak, Addy Osmani, Claude Pache, John K. Paul, Philip

Roberts, Mike Samuel, Tom Schuster, Kyle Simpson (getify), Kevin Smith, Dmitry Soshnikov, Ingvar Stepanyan, Tom Van Cutsem, Šime Vidas, Rick Waldron, Allen Wirfs-Brock, Nicholas C. Zakas, Ondřej Žára, Juriy Zaytsev (kangax). And many more!

Special thanks go to Benjamin Gruenbaum for his thorough review of the book.

About the author

Dr. Axel Rauschmayer has been programming since 1985 and 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.

Axel specializes in JavaScript, as blogger, book author and trainer. He has done extensive research into programming language design and has followed the state of JavaScript since its creation. He started blogging about ECMAScript 6 in early 2011.

I Background

1. About ECMAScript 6 (ES6)

It took a long time to finish it, but ECMAScript 6, the next version of JavaScript, is finally a reality:

- [It became a standard on 17 June 2015.](#)
- Most of its features are already widely available (as documented in kangax' [ES6 compatibility table](#)).
- Transpilers (such as [Babel](#)) let you compile ES6 to ES5.

The next sections explain concepts that are important in the world of ES6.

1.1 TC39 (Ecma Technical Committee 39)

[TC39 \(Ecma Technical Committee 39\)](#) is the committee that evolves JavaScript. Its members are companies (among others, all major browser vendors). [TC39 meets regularly](#), its meetings are attended by delegates that members send and by invited experts. Minutes of the meetings are [available online](#) and give you a good idea of how TC39 works.

1.2 How ECMAScript 6 was designed

The ECMAScript 6 design process centers on *proposals* for features. Proposals are often triggered by suggestions from the developer community. To avoid design by committee, proposals are maintained by *champions* (1–2 committee delegates).

A proposal goes through the following steps before it becomes a standard:

- Sketch (informally: “strawman proposal”): A first description of the proposed feature.
- Proposal: If TC39 agrees that a feature is important, it gets promoted to official proposal status. That does not guarantee it will become a standard, but it considerably increases its chances. The deadline for ES6 proposals was May 2011. No major new proposals were considered after that.
- Implementations: Proposed features must be implemented. Ideally in two JavaScript engines. Implementations and feedback from the community shape the proposal as it evolves.
- Standard: If the proposal continues to prove itself and is accepted by TC39, it will eventually be included in an edition of the ECMAScript standard. At this point, it is a standard feature.

[Source of this section: “[The Harmony Process](#)” by David Herman.]

1.2.1 The design process after ES6

Starting with ECMAScript 2016 (ES7), TC39 will time-box releases. A new version of ECMAScript will be released every year, with whatever features are ready at that time. That means that from now on, ECMAScript versions will be relatively small upgrades. For more information on the new process, including finished and upcoming feature proposals, consult [the GitHub repository](#) [ecma262](#).

1.3 JavaScript versus ECMAScript

JavaScript is what everyone calls the language, but that name is trademarked (by Oracle, which inherited the trademark from Sun). Therefore, the official name of JavaScript is *ECMAScript*. That name comes from the standards organization Ecma, which manages the language standard. Since ECMAScript's inception, the name of the organization has changed from the acronym "ECMA" to the proper name "Ecma".

Versions of JavaScript are defined by specifications that carry the official name of the language. Hence, the first standard version of JavaScript is ECMAScript 1 which is short for "ECMAScript Language Specification, Edition 1". ECMAScript x is often abbreviated ESx.

1.4 Upgrading to ES6

The stake holders on the web are:

- Implementors of JavaScript engines
- Developers of web applications
- Users

These groups have remarkably little control over each other. That's why upgrading a web language is so challenging.

On one hand, upgrading engines is challenging, because they are confronted with all kinds of code on the web, some of which is very old. You also want engine upgrades to be automatic and unnoticeable for users. Therefore, ES6 is a superset of ES5, nothing is removed¹. ES6 upgrades the language without introducing versions or modes. It even manages to make strict mode the de-facto default (via modules), without increasing the rift between it and sloppy mode. The approach that was taken is called "One JavaScript" and explained in [a separate chapter](#).

On the other hand, upgrading code is challenging, because your code must run on all JavaScript engines that are used by your target audience. Therefore, if you want to use ES6 in your code, you only have two choices: You can either wait until no one in your target audience uses a non-ES6 engine, anymore. That will take years; mainstream audiences were at that point w.r.t. ES5 when ES6 became a standard in June 2015. And ES5 was standardized in December 2009! Or you can compile ES6 to ES5 and use it now. More information on how to do that is given in the book "[Setting up ES6](#)", which is free to read online.

Goals and requirements clash in the design of ES6:

- Goals are fixing JavaScript's pitfalls and adding new features.
- Requirements are that both need to be done without breaking existing code and without changing the lightweight nature of the language.

1.5 Goals for ES6

[The original project page for Harmony/ES6](#) mentions several goals. In the following subsections, I'm taking a look at some of them.

1.5.1 Goal: Be a better language

The goal is: Be a better language for writing:

- i. complex applications;
- ii. libraries (possibly including the DOM) shared by those applications;
- iii. code generators targeting the new edition.

Sub-goal (i) acknowledges that applications written in JavaScript have grown huge. A key ES6 feature fulfilling this goal is built-in modules.

Modules are also an answer to goal (ii). As an aside, the DOM is notoriously difficult to implement in JavaScript. [ES6 Proxies](#) should help here.

Several features were mainly added to make it easier to compile to JavaScript. Two examples are:

- `Math.fround()` — rounding Numbers to 32 bit floats
- `Math.imul()` — multiplying two 32 bit ints

They are both useful for, e.g., compiling C/C++ to JavaScript via [Emscripten](#).

1.5.2 Goal: Improve interoperation

The goal is: Improve interoperation, adopting de facto standards where possible.

Examples are:

- Classes: are based on how constructor functions are currently used.
- Modules: picked up design ideas from the CommonJS module format.
- Arrow functions: have syntax that is borrowed from CoffeeScript.
- Named function parameters: There is no built-in support for named parameters. Instead, the existing practice of naming parameters via object literals is supported via [destructuring in parameter definitions](#).

1.5.3 Goal: Versioning

The goal is: Keep versioning as simple and linear as possible.

As mentioned previously, ES6 avoids versioning via “[One JavaScript](#)”: In an ES6 code base, everything is ES6, there are no parts that are ES5-specific.

1.6 Categories of ES6 features

The introduction of the ES6 specification lists all new features:

Some of [ECMAScript 6's] major enhancements include modules, class declarations, lexical block scoping, iterators and generators, promises for asynchronous programming, destructuring patterns, and proper tail calls. The ECMAScript library of built-ins has been expanded to support additional data abstractions including maps, sets, and arrays of binary numeric values as well as additional support for Unicode supplemental characters in strings and regular expressions. The built-ins are now extensible via subclassing.

There are three major categories of features:

- Better syntax for features that already exist (e.g. via libraries). For example:
 - [Classes](#)
 - [Modules](#)
- New functionality in the standard library. For example:
 - New methods for [strings](#) and [Arrays](#)
 - [Promises](#)
 - [Maps, Sets](#)
- Completely new features. For example:
 - [Generators](#)
 - [Proxies](#)
 - [WeakMaps](#)

1.7 A brief history of ECMAScript

This section describes what happened on the road to ECMAScript 6.

1.7.1 The early years: ECMAScript 1–3

- **ECMAScript 1 (June 1997)** was the first version of the JavaScript language standard.
- **ECMAScript 2 (June 1998)** contained minor changes, to keep the spec in sync with a separate ISO standard for JavaScript.
- **ECMAScript 3 (December 1999)** introduced many features that have become popular parts of the language, as described in the introduction of the ES6 specification: “[...] regular expressions, better string handling, new control statements, try/catch exception handling, tighter definition of errors, formatting for numeric output and other enhancements.”

1.7.2 ECMAScript 4 (abandoned in July 2008)

Work on ES4 started after the release of ES3 in 1999. In 2003, an interim report was released after which work on ES4 paused. Subsets of the language described in the interim report were implemented by Adobe (in [ActionScript](#)) and by Microsoft (in [JScript.NET](#)).

In February 2005, Jesse James Garrett observed that a combination of techniques had become popular for implementing dynamic frontend apps in JavaScript. [He called those techniques Ajax](#). Ajax enabled a completely new class of web apps and led to a surge of interest in JavaScript.

That may have contributed to TC39 resuming work on ES4 in fall 2005. They based ES4 on ES3, the interim ES4 report and experiences with [ActionScript](#) and [JScript.NET](#).

There were now two groups working on future ECMAScript versions:

- ECMAScript 4 was designed by Adobe, Mozilla, Opera, and Google and was a massive upgrade. Its planned feature sets included:
 - Programming in the large (classes, interfaces, namespaces, packages, program units, optional type annotations, and optional static type checking and verification)
 - Evolutionary programming and scripting (structural types, duck typing, type definitions, and multimethods)
 - Data structure construction (parameterized types, getters and setters, and meta-level methods)
 - Control abstractions (proper tail calls, iterators, and generators)
 - Introspection (type meta-objects and stack marks)
- ECMAScript 3.1 was designed by Microsoft and Yahoo. It was planned as a subset of ES4 and an incremental upgrade of ECMAScript 3, with bug fixes and minor new features. ECMAScript 3.1 eventually became ECMAScript 5.

The two groups disagreed on the future of JavaScript and tensions between them continued to increase.



Sources of this section:

- [“Proposed ECMAScript 4th Edition – Language Overview”](#). 2007-10-23
- [“ECMAScript Harmony”](#) by John Resig. 2008-08-13

1.7.3 ECMAScript Harmony

At the end of July 2008, there was a TC39 meeting in Oslo, whose outcome was [described](#) as follows by Brendan Eich:

It's no secret that the JavaScript standards body, Ecma's Technical Committee 39, has been split for over a year, with some members favoring ES4 [...] and others advocating ES3.1 [...]. Now, I'm happy to report, the split is over.

The agreement that was worked out at the meeting consisted of four points:

1. Develop an incremental update of ECMAScript (which became ECMAScript 5).
2. Develop a major new release, which was to be more modest than ECMAScript 4, but much larger in scope than the version after ECMAScript 3. This version was code-named *Harmony*, due to the nature of the meeting in which it was conceived.
3. Features from ECMAScript 4 that would be dropped: packages, namespaces, early binding.
4. Other ideas were to be developed in consensus with all of TC39.

Thus: The ES4 group agreed to make Harmony less radical than ES4, the rest of TC39 agreed to keep moving things forward.

The next versions of ECMAScript are:

- **ECMAScript 5 (December 2009)**. This is the version of ECMAScript that most browsers support today. It brings several enhancements to the standard library and updated language semantics via a *strict mode*.
- **ECMAScript 5.1 (June 2011)**. ES5 was submitted as an ISO standard. In the process, minor corrections were made. ES5.1 contains those corrections. It is the same text as ISO/IEC 16262:2011.
- **ECMAScript 6 (June 2015)**. This version went through several name changes:
 - ECMAScript Harmony: was the initial code name for JavaScript improvements after ECMAScript 5.
 - ECMAScript.next: It became apparent that the plans for Harmony were too ambitious for a single version, so its features were split into two groups: The first group of features had highest priority and was to become the next version after ES5. The code name of that version was ECMAScript.next, to avoid prematurely committing to a version number, which proved problematic with ES4. The second group of features had time until after ECMAScript.next.
 - ECMAScript 6: As ECMAScript.next matured, its code name was dropped and everybody started to call it ECMAScript 6.
 - ECMAScript 2015: In late 2014, TC39 decided to change the official name of ECMAScript 6 to ECMAScript 2015, in light of upcoming yearly spec releases. However, given how established the name “ECMAScript 6” already is and how late TC39 changed their minds, I expect that that's how everybody will continue to refer to that version.
- **ECMAScript 2016** was previously called ECMAScript 7. Starting with ES2016, the language standard will see smaller yearly releases.

2. FAQ: ECMAScript 6

This chapter answers a few frequently asked questions about ECMAScript 6.

2.1 How can I use ES6 today?

Most of ES6 is already supported in current engines, consult [Kangax' ES6 compatibility table](#) to find out what is supported where.

For other options (e.g. interactive ES6 command lines and transpiling ES6 to ES5 via Babel), consult Chap. [“Deploying ECMAScript 6”](#) in [“Setting up ES6”](#).

2.2 Isn't ECMAScript 6 now called ECMAScript 2015?

Yes and no. The official name is ECMAScript 2015, but ES6 is the name that everyone knows and uses. That's why I decided to use the latter for this book.

After ES6, ECMAScript editions are created via [a new process](#) and a yearly release cycle. That seems like a good opportunity to switch to the new naming scheme. Therefore, I'll use the name "ECMAScript 2016" for the edition after ES6.

2.3 How do I migrate my ECMAScript 5 code to ECMAScript 6?

There is nothing to do: ECMAScript 6 is a superset of ECMAScript 5. Therefore, all of your ES5 code is automatically ES6 code. That helps tremendously with incrementally adopting this new version. How exactly ES6 stays completely backward-compatible is explained in [the chapter on "One JavaScript"](#).

2.4 Does it still make sense to learn ECMAScript 5?

ES6 is increasingly well supported everywhere. Does that mean that you shouldn't learn ECMAScript 5, anymore? It doesn't, for several reasons:

- ECMAScript 6 is a superset of ECMAScript 5 – new JavaScript versions must never break existing code. Thus, nothing you learn about ECMAScript 5 is learned in vain.
- There are several ECMAScript 6 features that kind of replace ECMAScript 5 features, but still use them as their foundations. It is important to understand those foundations. Two examples: classes are internally translated to constructors and methods are still functions (as they have always been).
- As long as ECMAScript 6 is compiled to ECMAScript 5, it is useful to understand the output of the compilation process. And you'll have to compile to ES5 for a while (probably years), until you can rely on ES6 being available in all relevant browsers.
- It's important to be able to understand legacy code.

2.5 Is ES6 bloated?

One occasionally comes across accusations of ES6 being bloated and introducing too much useless *syntactic sugar* (more convenient syntax for something that already exists).

However, in many ways, JavaScript is just now catching up with languages such as Python and Ruby. Both still have more features and come with a much richer standard library.

If someone complains about ES6 being too big, I suggest that they try it out for a while. Nobody forces you to use any of the new features. You can start small (consult Chap. ["Core ES6 features"](#) for suggestions) and then use more new features, as you grow more comfortable with ES6. So far, the feedback I get from people who have actually programmed with ES6 (as opposed to read about it) is overwhelmingly positive.

Furthermore, things that superficially look like syntactic sugar (such as classes and modules) bring much-needed standardization to the language and serve as foundations for future features.

Lastly, several features were not created for normal programmers, but for library authors (e.g. generators, iterators, proxies). "Normal programmers" only need to know them superficially if at all.

2.6 Isn't the ES6 specification very big?

The ECMAScript specification has indeed grown tremendously: The ECMAScript 5.1 PDF had 245 pages, the ES6 PDF has 593 pages. But, for comparison, the Java 8 language specification has 724 pages (excluding an index). Furthermore, the ES6 specification contains details that many other language specifications omit as implementation-defined. It also specifies how its standard library works².

2.7 Does ES6 have array comprehensions?

Originally, ES6 was to have Array and generator comprehensions (similarly to Haskell and Python). But they were not added, because TC39 wanted to explore two avenues:

- It may be possible to create comprehensions that work for arbitrary datatypes (think Microsoft's LINQ).
- It may also be possible that methods for iterators are a better way to achieve what comprehensions do.

2.8 Is ES6 statically typed?

Static typing is not part of ES6. However, the following two technologies add static typing to JavaScript. Similar features may eventually be standardized.

- **Microsoft TypeScript:** is basically ES6 plus optional type annotations. At the moment, it is compiled to ES5 and throws away the type information while doing so. Optionally, it can also make that information available at runtime, for type introspection and for runtime type checks.

- **Facebook Flow:** is a type checker for ECMAScript 6 that is based on flow analysis. As such, it only adds optional type annotations to the language and infers and checks types. It does not help with compiling ES6 to ES5.

Two benefits of static typing are:

- It allows you to detect a certain category of errors earlier, because the code is analyzed statically (during development, without running code). As such, static typing is complementary to testing and catches different errors.
- It helps IDEs with auto-completion.

Both TypeScript and Flow are using the same notation. Type annotations are optional, which makes this approach relatively lightweight. Even without annotations, types can often be inferred. Therefore, this kind of type checking is even useful for completely unannotated code, as a consistency check.

3. One JavaScript: avoiding versioning in ECMAScript 6

What is the best way to add new features to a language? This chapter describes the approach taken by ECMAScript 6. It is called *One JavaScript*, because it avoids versioning.

3.1 Versioning

In principle, a new version of a language is a chance to clean it up, by removing outdated features or by changing how features work. That means that new code doesn't work in older implementations of the language and that old code doesn't work in a new implementation. Each piece of code is linked to a specific version of the language. Two approaches are common for dealing with versions being different.

First, you can take an "all or nothing" approach and demand that, if a code base wants to use the new version, it must be upgraded completely. Python took that approach when upgrading from Python 2 to Python 3. A problem with it is that it may not be feasible to migrate all of an existing code base at once, especially if it is large. Furthermore, the approach is not an option for the web, where you'll always have old code and where JavaScript engines are updated automatically.

Second, you can permit a code base to contain code in multiple versions, by tagging code with versions. On the web, you could tag ECMAScript 6 code via a dedicated [Internet media type](#). Such a media type can be associated with a file via an HTTP header:

```
Content-Type: application/ecmascript;version=6
```

It can also be associated via the `type` attribute of the `<script>` element (whose [default value](#) is `text/javascript`):

```
<script type="application/ecmascript;version=6">
...
</script>
```

This specifies the version *out of band*, externally to the actual content. Another option is to specify the version inside the content (*in-band*). For example, by starting a file with the following line:

```
use version 6;
```

Both ways of tagging are problematic: out-of-band versions are brittle and can get lost, in-band versions add clutter to code.

A more fundamental issue is that allowing multiple versions per code base effectively forks a language into sub-languages that have to be maintained in parallel. This causes problems:

- Engines become bloated, because they need to implement the semantics of all versions. The same applies to tools analyzing the language (e.g. style checkers such as JSLint).
- Programmers need to remember how the versions differ.
- Code becomes harder to refactor, because you need to take versions into consideration when you move pieces of code.

Therefore, versioning is something to avoid, especially for JavaScript and the web.

3.1.1 Evolution without versioning

But how can we get rid of versioning? By always being backward-compatible. That means we must give up some of our ambitions w.r.t. cleaning up JavaScript: We can't introduce breaking changes. Being backward-compatible means not removing features and not changing features. The slogan for this principle is: "don't break the web".

We can, however, add new features and make existing features more powerful.

As a consequence, no versions are needed for new engines, because they can still run all old code. David Herman calls this approach to avoiding versioning [One JavaScript \(1JS\) \[1\]](#), because it avoids splitting up JavaScript into different versions or modes. As we shall see later, 1JS even undoes some of a split that already exists, due to strict mode.

One JavaScript does not mean that you have to completely give up on cleaning up the language. Instead of cleaning up existing features, you introduce new, clean, features. One example for that is `let`, which declares block-scoped variables and is an improved version of `var`. It does not, however, replace `var`. It exists alongside it, as the superior option.

One day, it may even be possible to eliminate features that nobody uses, anymore. Some of the ES6 features were designed by surveying JavaScript code on the web. Two examples are:

- `let`-declarations are difficult to add to non-strict mode, because `let` is not a reserved word in that mode. The only variant of `let` that looks like valid ES5 code is:

```
let[x] = arr;
```

Research yielded that no code on the web uses a variable `let` in non-strict mode in this manner. That enabled TC39 to add `let` to non-strict mode. Details of how this was done are described [later in this chapter](#).

- Function declarations do occasionally appear in non-strict blocks, which is why the ES6 specification describes measures that web browsers can take to ensure that such code doesn't break. [Details are explained later](#).

3.2 Strict mode and ECMAScript 6

Strict mode was introduced in ECMAScript 5 to clean up the language. It is switched on by putting the following line first in a file or in a function:

```
'use strict';
```

Strict mode introduces three kinds of breaking changes:

- Syntactic changes: some previously legal syntax is forbidden in strict mode. For example:
 - The `with` statement is forbidden. It lets users add arbitrary objects to the chain of variable scopes, which slows down execution and makes it tricky to figure out what a variable refers to.
 - Deleting an unqualified identifier (a variable, not a property) is forbidden.
 - Functions can only be declared at the top level of a scope.
 - More identifiers are [reserved](#): `implements` `interface` `let` `package` `private` `protected` `public` `static` `yield`
- More errors. For example:
 - Assigning to an undeclared variable causes a `ReferenceError`. In non-strict mode, a global variable is created in this case.
 - Changing read-only properties (such as the length of a string) causes a `TypeError`. In non-strict mode, it simply has no effect.
- Different semantics: Some constructs behave differently in strict mode. For example:
 - `arguments` doesn't track the current values of parameters, anymore.
 - `this` is `undefined` in non-method functions. In non-strict mode, it refers to the global object (`window`), which meant that global variables were created if you called a constructor without `new`.

Strict mode is a good example of why versioning is tricky: Even though it enables a cleaner version of JavaScript, its adoption is still relatively low. The main reasons are that it breaks some existing code, can slow down execution and is a hassle to add to files (let alone interactive command lines). I love *the idea* of strict mode and don't nearly use it often enough.

3.2.1 Supporting sloppy (non-strict) mode

One JavaScript means that we can't give up on sloppy mode: it will continue to be around (e.g. in HTML attributes). Therefore, we can't build ECMAScript 6 on top of strict mode, we must add its features to both strict mode and non-strict mode (a.k.a. sloppy mode). Otherwise, strict mode would be a different version of the language and we'd be back to versioning. Unfortunately, two ECMAScript 6 features are difficult to add to sloppy mode: `let` declarations and block-level function declarations. Let's examine why that is and how to add them, anyway.

3.2.2 `let` declarations in sloppy mode

`let` enables you to declare block-scoped variables. It is difficult to add to sloppy mode, because `let` is only a reserved word in strict mode. That is, the following two statements are legal ES5 sloppy code:

```
var let = [];  
let[x] = 'abc';
```

In strict ECMAScript 6, you get an exception in line 1, because you are using the reserved word `let` as a variable name. And the statement in line 2 is interpreted as a `let` variable declaration (that uses destructuring).

In sloppy ECMAScript 6, the first line does not cause an exception, but the second line is still interpreted as a `let` declaration. This way of using the identifier `let` is so rare on the web that ES6 can afford to make this interpretation. Other ways of writing `let` declarations can't be mistaken for sloppy ES5 syntax:

```
let foo = 123;  
let {x,y} = computeCoordinates();
```

3.2.3 Block-level function declarations in sloppy mode

ECMAScript 5 strict mode forbids function declarations in blocks. The specification allowed them in sloppy mode, but didn't specify how they should behave. Hence, various implementations of JavaScript support them, but handle them differently.

ECMAScript 6 wants a function declaration in a block to be local to that block. That is OK as an extension of ES5 strict mode, but breaks some sloppy code. Therefore, ES6 provides "[web legacy compatibility semantics](#)" for browsers that lets function declarations in blocks exist at function scope.

3.2.4 Other keywords

The identifiers `yield` and `static` are only reserved in ES5 strict mode. ECMAScript 6 uses context-specific syntax rules to make them work in sloppy mode:

- In sloppy mode, `yield` is only a reserved word inside a generator function.
- `static` is currently only used inside class literals, which are implicitly strict (see below).

3.2.5 Implicit strict mode

The bodies of modules and classes are implicitly in strict mode in ECMAScript 6 – there is no need for the `'use strict'` marker. Given that virtually all of our code will live in modules in the future, ECMAScript 6 effectively upgrades the whole language to strict mode.

The bodies of other constructs (such as arrow functions and generator functions) could have been made implicitly strict, too. But considering how small these constructs usually are, using them in sloppy mode would have resulted in code that is fragmented between the two modes. Classes and especially modules are large enough to make fragmentation less of an issue.

3.2.6 Things that can't be fixed

The downside of One JavaScript is that you can't fix existing quirks, especially the following two.

First, `typeof null` should return the string `'null'` and not `'object'`. TC39 tried fixing it, but it broke existing code. On the other hand, adding new results for new kinds of operands is OK, because current JavaScript engines already occasionally return custom values for host objects. One example are ECMAScript 6's symbols:

```
> typeof Symbol.iterator  
'symbol'
```

Second, the global object (window in browsers) shouldn't be in the scope chain of variables. But it is also much too late to change that now. At least, one won't be in global scope in modules and `let` never creates properties of the global object, not even when used in global scope.

3.3 Breaking changes in ES6

ECMAScript 6 does introduce a few minor breaking changes (nothing you're likely to encounter). They are listed in two annexes:

- [Annex D: Corrections and Clarifications in ECMAScript 2015 with Possible Compatibility Impact](#)
- [Annex E: Additions and Changes That Introduce Incompatibilities with Prior Editions](#)

3.4 Conclusion

One JavaScript means making ECMAScript 6 completely backward-compatible. It is great that that succeeded. Especially appreciated is that modules (and thus most of our code) are implicitly in strict mode.

In the short term, adding ES6 constructs to both strict mode and sloppy mode is more work when it comes to writing the language specification and to implementing it in

engines. In the long term, both the spec and engines profit from the language not being forked (less bloat etc.). Programmers profit immediately from One JavaScript, because it makes it easier to get started with ECMAScript 6.

3.5 Further reading

[1] The original 1JS proposal (warning: out of date): “[ES6 doesn't need opt-in](#)” by David Herman.

4. Core ES6 features

This chapter describes the core ES6 features. These features are easy to adopt; the remaining features are mainly of interest to library authors. I explain each feature via the corresponding ES5 code.

4.1 From `var` to `let/const`

In ES5, you declare variables via `var`. Such variables are *function-scoped*, their scopes are the innermost enclosing functions. The behavior of `var` is occasionally confusing. This is an example:

```
var x = 3;
function func(randomize) {
  if (randomize) {
    var x = Math.random(); // (A) scope: whole function
    return x;
  }
  return x; // accesses the x from line A
}
func(false); // undefined
```

That `func()` returns `undefined` may be surprising. You can see why if you rewrite the code so that it more closely reflects what is actually going on:

```
var x = 3;
function func(randomize) {
  var x;
  if (randomize) {
    x = Math.random();
    return x;
  }
  return x;
}
func(false); // undefined
```

In ES6, you can additionally declare variables via `let` and `const`. Such variables are *block-scoped*, their scopes are the innermost enclosing blocks. `let` is roughly a block-scoped version of `var`. `const` works like `let`, but creates variables whose values can't be changed.

`let` and `const` behave more strictly and throw more exceptions (e.g. when you access their variables inside their scope before they are declared). Block-scoping helps with keeping the effects of code fragments more local (see the next section for a demonstration). And it's more mainstream than function-scoping, which eases moving between JavaScript and other programming languages.

If you replace `var` with `let` in the initial version, you get different behavior:

```
let x = 3;
function func(randomize) {
  if (randomize) {
    let x = Math.random();
    return x;
  }
  return x;
}
func(false); // 3
```

That means that you can't blindly replace `var` with `let` or `const` in existing code; you have to be careful during refactoring.

My advice is:

- Prefer `const`. You can use it for all variables whose values never change.
- Otherwise, use `let` – for variables whose values do change.
- Avoid `var`.

More information: chapter “[Variables and scoping](#)”.

4.2 From IIFEs to blocks

In ES5, you had to use a pattern called IIFE (Immediately-Invoked Function Expression) if you wanted to restrict the scope of a variable `tmp` to a block:

```
(function () { // open IIFE
  var tmp = ...;
  ...
})
```

```
})(); // close IIFE
console.log(tmp); // ReferenceError
```

In ECMAScript 6, you can simply use a block and a `let` declaration (or a `const` declaration):

```
{ // open block
  let tmp = ...;
  ...
} // close block
console.log(tmp); // ReferenceError
```

More information: section “[Avoid IIFEs in ES6](#)”.

4.3 From concatenating strings to template literals

With ES6, JavaScript finally gets literals for string interpolation and multi-line strings.

4.3.1 String interpolation

In ES5, you put values into strings by concatenating those values and string fragments:

```
function printCoord(x, y) {
  console.log('('+x+', '+y+')');
}
```

In ES6 you can use string interpolation via template literals:

```
function printCoord(x, y) {
  console.log(`(${x}, ${y})`);
}
```

4.3.2 Multi-line strings

Template literals also help with representing multi-line strings.

For example, this is what you have to do to represent one in ES5:

```
var HTML5_SKELETON =
  '<!doctype html>\n' +
  '<html>\n' +
  '<head>\n' +
  '  <meta charset="UTF-8">\n' +
  '  <title></title>\n' +
  '</head>\n' +
  '<body>\n' +
  '</body>\n' +
  '</html>\n';
```

If you escape the newlines via backslashes, things look a bit nicer (but you still have to explicitly add newlines):

```
var HTML5_SKELETON = `
<!doctype html>\n
<html>\n
<head>\n
  <meta charset="UTF-8">\n
  <title></title>\n
</head>\n
<body>\n
</body>\n
</html>`;
```

ES6 template literals can span multiple lines:

```
const HTML5_SKELETON = `
<!doctype html>
<html>
<head>
  <meta charset="UTF-8">
  <title></title>
</head>
<body>
</body>
</html>`;
```

(The examples differ in how much whitespace is included, but that doesn't matter in this case.)

More information: chapter “[Template literals and tagged templates](#)”.

4.4 From function expressions to arrow functions

In current ES5 code, you have to be careful with `this` whenever you are using function expressions. In the following example, I create the helper variable `_this` (line A) so that the `this` of `UiComponent` can be accessed in line B.

```
function UiComponent() {
  var this = this; // (A)
  var button = document.getElementById('myButton');
  button.addEventListener('click', function () {
    console.log('CLICK');
    _this.handleClick(); // (B)
  });
}
```

```

    });
}
UiComponent.prototype.handleClick = function () {
    ...
};

```

In ES6, you can use arrow functions, which don't shadow `this` (line A):

```

function UiComponent() {
    var button = document.getElementById('myButton');
    button.addEventListener('click', () => {
        console.log('CLICK');
        this.handleClick(); // (A)
    });
}

```

(In ES6, you also have the option of using a class instead of a constructor function. That is explored later.)

Arrow functions are especially handy for short callbacks that only return results of expressions.

In ES5, such callbacks are relatively verbose:

```

var arr = [1, 2, 3];
var squares = arr.map(function (x) { return x * x });

```

In ES6, arrow functions are much more concise:

```

const arr = [1, 2, 3];
const squares = arr.map(x => x * x);

```

When defining parameters, you can even omit parentheses if the parameters are just a single identifier. Thus: `(x) => x * x` and `x => x * x` are both allowed.

More information: chapter "[Arrow functions](#)".

4.5 Handling multiple return values

Some functions or methods return multiple values via arrays or objects. In ES5, you always need to create intermediate variables if you want to access those values. In ES6, you can avoid intermediate variables via destructuring.

4.5.1 Multiple return values via arrays

`exec()` returns captured groups via an Array-like object. In ES5, you need an intermediate variable (`matchObj` in the example below), even if you are only interested in the groups:

```

var matchObj =
    /^(\d\d\d\d)-(\d\d)-(\d\d)$/
    .exec('2999-12-31');
var year = matchObj[1];
var month = matchObj[2];
var day = matchObj[3];

```

In ES6, destructuring makes this code simpler:

```

const [, year, month, day] =
    /^(\d\d\d\d)-(\d\d)-(\d\d)$/
    .exec('2999-12-31');

```

The empty slot at the beginning of the Array pattern skips the Array element at index zero.

4.5.2 Multiple return values via objects

The method `Object.getOwnPropertyDescriptor()` returns a *property descriptor*, an object that holds multiple values in its properties.

In ES5, even if you are only interested in the properties of an object, you still need an intermediate variable (`propDesc` in the example below):

```

var obj = { foo: 123 };

var propDesc = Object.getOwnPropertyDescriptor(obj, 'foo');
var writable = propDesc.writable;
var configurable = propDesc.configurable;

console.log(writable, configurable); // true true

```

In ES6, you can use destructuring:

```

const obj = { foo: 123 };

const {writable, configurable} =
    Object.getOwnPropertyDescriptor(obj, 'foo');

console.log(writable, configurable); // true true

```

`{writable, configurable}` is an abbreviation for:

```

{ writable: writable, configurable: configurable }

```

More information: chapter [“Destructuring”](#).

4.6 From `for` to `forEach()` to `for-of`

Prior to ES5, you iterated over Arrays as follows:

```
var arr = ['a', 'b', 'c'];
for (var i=0; i<arr.length; i++) {
  var elem = arr[i];
  console.log(elem);
}
```

In ES5, you have the option of using the Array method `forEach()`:

```
arr.forEach(function (elem) {
  console.log(elem);
});
```

A `for` loop has the advantage that you can break from it, `forEach()` has the advantage of conciseness.

In ES6, the `for-of` loop combines both advantages:

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

If you want both index and value of each array element, `for-of` has got you covered, too, via the new Array method `entries()` and destructuring:

```
for (const [index, elem] of arr.entries()) {
  console.log(index+' '+elem);
}
```

More information: Chap. [“The for-of loop”](#).

4.7 Handling parameter default values

In ES5, you specify default values for parameters like this:

```
function foo(x, y) {
  x = x || 0;
  y = y || 0;
  ...
}
```

ES6 has nicer syntax:

```
function foo(x=0, y=0) {
  ...
}
```

An added benefit is that in ES6, a parameter default value is only triggered by `undefined`, while it is triggered by any falsy value in the previous ES5 code.

More information: section [“Parameter default values”](#).

4.8 Handling named parameters

A common way of naming parameters in JavaScript is via object literals (the so-called *options object pattern*):

```
selectEntries({ start: 0, end: -1 });
```

Two advantages of this approach are: Code becomes more self-descriptive and it is easier to omit arbitrary parameters.

In ES5, you can implement `selectEntries()` as follows:

```
function selectEntries(options) {
  var start = options.start || 0;
  var end = options.end || -1;
  var step = options.step || 1;
  ...
}
```

In ES6, you can use destructuring in parameter definitions and the code becomes simpler:

```
function selectEntries({ start=0, end=-1, step=1 }) {
  ...
}
```

4.8.1 Making the parameter optional

To make the parameter `options` optional in ES5, you'd add line A to the code:

```
function selectEntries(options) {
  options = options || {}; // (A)
  var start = options.start || 0;
  var end = options.end || -1;
  var step = options.step || 1;
```



```

    ...
}

```

In ES6 you can specify {} as a parameter default value:

```

function selectEntries({ start=0, end=-1, step=1 } = {}) {
    ...
}

```

More information: section “[Simulating named parameters](#)”.

4.9 From arguments to rest parameters

In ES5, if you want a function (or method) to accept an arbitrary number of arguments, you must use the special variable arguments:

```

function logAllArguments() {
    for (var i=0; i < arguments.length; i++) {
        console.log(arguments[i]);
    }
}

```

In ES6, you can declare a rest parameter (args in the example below) via the ... operator:

```

function logAllArguments(...args) {
    for (const arg of args) {
        console.log(arg);
    }
}

```

Rest parameters are even nicer if you are only interested in trailing parameters:

```

function format(pattern, ...args) {
    ...
}

```

Handling this case in ES5 is clumsy:

```

function format(pattern) {
    var args = [].slice.call(arguments, 1);
    ...
}

```

Rest parameters make code easier to read: You can tell that a function has a variable number of parameters just by looking at its parameter definitions.

More information: section “[Rest parameters](#)”.

4.10 From apply() to the spread operator (...)

In ES5, you turn arrays into parameters via apply(). ES6 has the spread operator for this purpose.

4.10.1 Math.max()

Math.max() returns the numerically greatest of its arguments. It works for an arbitrary number of arguments, but not for Arrays.

ES5 – apply():

```

> Math.max.apply(Math, [-1, 5, 11, 3])
11

```

ES6 – spread operator:

```

> Math.max(...[-1, 5, 11, 3])
11

```

4.10.2 Array.prototype.push()

Array.prototype.push() appends all of its arguments as elements to its receiver. There is no method that destructively appends an Array to another one.

ES5 – apply():

```

var arr1 = ['a', 'b'];
var arr2 = ['c', 'd'];

arr1.push.apply(arr1, arr2);
// arr1 is now ['a', 'b', 'c', 'd']

```

ES6 – spread operator:

```

const arr1 = ['a', 'b'];
const arr2 = ['c', 'd'];

arr1.push(...arr2);
// arr1 is now ['a', 'b', 'c', 'd']

```

More information: section “[The spread operator \(...\)](#)”.

4.11 From `concat()` to the spread operator (`...`)

The spread operator can also (non-destructively) turn the contents of its operand into Array elements. That means that it becomes an alternative to the Array method `concat()`.

ES5 – `concat()`:

```
var arr1 = ['a', 'b'];
var arr2 = ['c'];
var arr3 = ['d', 'e'];

console.log(arr1.concat(arr2, arr3));
// [ 'a', 'b', 'c', 'd', 'e' ]
```

ES6 – spread operator:

```
const arr1 = ['a', 'b'];
const arr2 = ['c'];
const arr3 = ['d', 'e'];

console.log([...arr1, ...arr2, ...arr3]);
// [ 'a', 'b', 'c', 'd', 'e' ]
```

More information: section “[The spread operator \(...\)](#)”.

4.12 From function expressions in object literals to method definitions

In JavaScript, methods are properties whose values are functions.

In ES5 object literals, methods are created like other properties. The property values are provided via function expressions.

```
var obj = {
  foo: function () {
    ...
  },
  bar: function () {
    this.foo();
  }, // trailing comma is legal in ES5
}
```

ES6 has *method definitions*, special syntax for creating methods:

```
const obj = {
  foo() {
    ...
  },
  bar() {
    this.foo();
  },
}
```

More information: section “[Method definitions](#)”.

4.13 From constructors to classes

ES6 classes are mostly just more convenient syntax for constructor functions.

4.13.1 Base classes

In ES5, you implement constructor functions directly:

```
function Person(name) {
  this.name = name;
}
Person.prototype.describe = function () {
  return 'Person called ' + this.name;
};
```

In ES6, classes provide slightly more convenient syntax for constructor functions:

```
class Person {
  constructor(name) {
    this.name = name;
  }
  describe() {
    return 'Person called ' + this.name;
  }
}
```

Note the compact syntax for method definitions – no keyword `function` needed. Also note that there are no commas between the parts of a class.

4.13.2 Derived classes

Subclassing is complicated in ES5, especially referring to super-constructors and super-properties. This is the canonical way of creating a sub-constructor `Employee` of `Person`:

```
function Employee(name, title) {
  Person.call(this, name); // super(name)
```

```

    this.title = title;
  }
  Employee.prototype = Object.create(Person.prototype);
  Employee.prototype.constructor = Employee;
  Employee.prototype.describe = function () {
    return Person.prototype.describe.call(this) // super.describe()
      + ' (' + this.title + ')';
  };
};

```

ES6 has built-in support for subclassing, via the `extends` clause:

```

class Employee extends Person {
  constructor(name, title) {
    super(name);
    this.title = title;
  }
  describe() {
    return super.describe() + ' (' + this.title + ')';
  }
}

```

More information: chapter “Classes”.

4.14 From custom error constructors to subclasses of `Error`

In ES5, it is impossible to subclass the built-in constructor for exceptions, `Error`. The following code shows a work-around that gives the constructor `MyError` important features such as a stack trace:

```

function MyError() {
  // Use Error as a function
  var superInstance = Error.apply(null, arguments);
  copyOwnPropertiesFrom(this, superInstance);
}
MyError.prototype = Object.create(Error.prototype);
MyError.prototype.constructor = MyError;

function copyOwnPropertiesFrom(target, source) {
  Object.getOwnPropertyNames(source)
    .forEach(function(propKey) {
      var desc = Object.getOwnPropertyDescriptor(source, propKey);
      Object.defineProperty(target, propKey, desc);
    });
  return target;
};

```

In ES6, all built-in constructors can be subclassed, which is why the following code achieves what the ES5 code can only simulate:

```

class MyError extends Error {
}

```

More information: section “Subclassing built-in constructors”.

4.15 From objects to Maps

Using the language construct *object* as a map from strings to arbitrary values (a data structure) has always been a makeshift solution in JavaScript. The safest way to do so is by creating an object whose prototype is `null`. Then you still have to ensure that no key is ever the string `'__proto__'`, because that property key triggers special functionality in many JavaScript engines.

The following ES5 code contains the function `countWords` that uses the object `dict` as a map:

```

var dict = Object.create(null);
function countWords(word) {
  var escapedWord = escapeKey(word);
  if (escapedWord in dict) {
    dict[escapedWord]++;
  } else {
    dict[escapedWord] = 1;
  }
}
function escapeKey(key) {
  if (key.indexOf('__proto__') === 0) {
    return key+'%';
  } else {
    return key;
  }
}

```

In ES6, you can use the built-in data structure `Map` and don't have to escape keys. As a downside, incrementing values inside Maps is less convenient.

```

const map = new Map();
function countWords(word) {
  const count = map.get(word) || 0;
  map.set(word, count + 1);
}

```

Another benefit of Maps is that you can use arbitrary values as keys, not just strings.

More information:

- Section “[The dict Pattern: Objects Without Prototypes Are Better Maps](#)” in “Speaking JavaScript”
- Chapter “[Maps and Sets](#)”

4.16 New string methods

The ECMAScript 6 standard library provides several new methods for strings.

From `indexOf` to `startsWith`:

```
if (str.indexOf('x') === 0) {} // ES5
if (str.startsWith('x')) {} // ES6
```

From `indexOf` to `endsWith`:

```
function endsWith(str, suffix) { // ES5
  var index = str.indexOf(suffix);
  return index >= 0
    && index === str.length - suffix.length;
}
str.endsWith(suffix); // ES6
```

From `indexOf` to `includes`:

```
if (str.indexOf('x') >= 0) {} // ES5
if (str.includes('x')) {} // ES6
```

From `join` to `repeat` (the ES5 way of repeating a string is more of a hack):

```
new Array(3+1).join('#') // ES5
'#'.repeat(3) // ES6
```

More information: Chapter “[New string features](#)”

4.17 New Array methods

There are also several new Array methods in ES6.

4.17.1 From `Array.prototype.indexOf` to `Array.prototype.findIndex`

The latter can be used to find `NaN`, which the former can’t detect:

```
const arr = ['a', NaN];

arr.indexOf(NaN); // -1
arr.findIndex(x => Number.isNaN(x)); // 1
```

As an aside, the new `Number.isNaN()` provides a safe way to detect `NaN` (because it doesn’t coerce non-numbers to numbers):

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

4.17.2 From `Array.prototype.slice()` to `Array.from()` or the spread operator

In ES5, `Array.prototype.slice()` was used to convert Array-like objects to Arrays. In ES6, you have `Array.from()`:

```
var arr1 = Array.prototype.slice.call(arguments); // ES5
const arr2 = Array.from(arguments); // ES6
```

If a value is iterable (as all Array-like DOM data structure are by now), you can also use the spread operator (`...`) to convert it to an Array:

```
const arr1 = [...'abc'];
// ['a', 'b', 'c']
const arr2 = [...new Set().add('a').add('b')];
// ['a', 'b']
```

4.17.3 From `apply()` to `Array.prototype.fill()`

In ES5, you can use `apply()`, as a hack, to create in Array of arbitrary length that is filled with `undefined`:

```
// Same as Array(undefined, undefined)
var arr1 = Array.apply(null, new Array(2));
// [undefined, undefined]
```

In ES6, `fill()` is a simpler alternative:

```
const arr2 = new Array(2).fill(undefined);
// [undefined, undefined]
```

`fill()` is even more convenient if you want to create an Array that is filled with an arbitrary value:

```
// ES5
var arr3 = Array.apply(null, new Array(2))
  .map(function (x) { return 'x' });
// ['x', 'x']
```

```
// ES6
const arr4 = new Array(2).fill('x');
// ['x', 'x']
```

`fill()` replaces all Array elements with the given value. Holes are treated as if they were elements.

More information: Sect. “Creating Arrays filled with values”

4.18 From CommonJS modules to ES6 modules

Even in ES5, module systems based on either AMD syntax or CommonJS syntax have mostly replaced hand-written solutions such as [the revealing module pattern](#).

ES6 has built-in support for modules. Alas, no JavaScript engine supports them natively, yet. But tools such as browserify, webpack or jspm let you use ES6 syntax to create modules, making the code you write future-proof.

4.18.1 Multiple exports

4.18.1.1 Multiple exports in CommonJS

In CommonJS, you export multiple entities as follows:

```
//----- lib.js -----
var sqrt = Math.sqrt;
function square(x) {
  return x * x;
}
function diag(x, y) {
  return sqrt(square(x) + square(y));
}
module.exports = {
  sqrt: sqrt,
  square: square,
  diag: diag,
};

//----- main1.js -----
var square = require('lib').square;
var diag = require('lib').diag;

console.log(square(11)); // 121
console.log(diag(4, 3)); // 5
```

Alternatively, you can import the whole module as an object and access `square` and `diag` via it:

```
//----- main2.js -----
var lib = require('lib');
console.log(lib.square(11)); // 121
console.log(lib.diag(4, 3)); // 5
```

4.18.1.2 Multiple exports in ES6

In ES6, multiple exports are called *named exports* and handled like this:

```
//----- lib.js -----
export const sqrt = Math.sqrt;
export function square(x) {
  return x * x;
}
export function diag(x, y) {
  return sqrt(square(x) + square(y));
}

//----- main1.js -----
import { square, diag } from 'lib';
console.log(square(11)); // 121
console.log(diag(4, 3)); // 5
```

The syntax for importing modules as objects looks as follows (line A):

```
//----- main2.js -----
import * as lib from 'lib'; // (A)
console.log(lib.square(11)); // 121
console.log(lib.diag(4, 3)); // 5
```

4.18.2 Single exports

4.18.2.1 Single exports in CommonJS

Node.js extends CommonJS and lets you export single values from modules, via `module.exports`:

```
//----- myFunc.js -----
module.exports = function () { ... };

//----- main1.js -----
var myFunc = require('myFunc');
myFunc();
```

4.18.2.2 Single exports in ES6

In ES6, the same thing is done via a so-called *default export* (declared via `export default`):

```
//----- myFunc.js -----  
export default function () { ... } // no semicolon!  
  
//----- main1.js -----  
import myFunc from 'myFunc';  
myFunc();
```

More information: chapter “[Modules](#)”.

4.19 What to do next

Now that you got a first taste of ES6, you can continue your exploration by browsing the chapters: Each chapter covers a feature or a set of related features and starts with an overview. [The last chapter](#) collects all of these overview sections in a single location.

II Data

5. New number and `Math` features

5.1 Overview

5.1.1 New integer literals

You can now specify integers in binary and octal notation:

```
> 0xFF // ES5: hexadecimal  
255  
> 0b11 // ES6: binary  
3  
> 0o10 // ES6: octal  
8
```

5.1.2 New `Number` properties

The global object `Number` gained a few new properties:

- `Number.EPSILON` for comparing floating point numbers with a tolerance for rounding errors.
- `Number.isInteger(num)` checks whether `num` is an integer (a number without a decimal fraction):

```
> Number.isInteger(1.05)  
false  
> Number.isInteger(1)  
true  
  
> Number.isInteger(-3.1)  
false  
> Number.isInteger(-3)  
true
```
- A method and constants for determining whether a JavaScript integer is *safe* (within the signed 53 bit range in which there is no loss of precision):
 - `Number.isSafeInteger(number)`
 - `Number.MIN_SAFE_INTEGER`
 - `Number.MAX_SAFE_INTEGER`
- `Number.isNaN(num)` checks whether `num` is the value `NaN`. In contrast to the global function `isNaN()`, it doesn't coerce its argument to a number and is therefore safer for non-numbers:

```
> isNaN('???')  
true  
> Number.isNaN('???')  
false
```
- Three additional methods of `Number` are mostly equivalent to the global functions with the same names: `Number.isFinite`, `Number.parseFloat`, `Number.parseInt`.

5.1.3 New `Math` methods

The global object `Math` has new methods for numerical, trigonometric and bitwise operations. Let's look at four examples.

`Math.sign()` returns the sign of a number:

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

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

`Math.trunc()` removes the decimal fraction of a number:

```
> Math.trunc(3.1)
3
> Math.trunc(3.9)
3
> Math.trunc(-3.1)
-3
> Math.trunc(-3.9)
-3
```

`Math.log10()` computes the logarithm to base 10:

```
> Math.log10(100)
2
```

`Math.hypot()` Computes the square root of the sum of the squares of its arguments (Pythagoras' theorem):

```
> Math.hypot(3, 4)
5
```

5.2 New integer literals

ECMAScript 5 already has literals for hexadecimal integers:

```
> 0x9
9
> 0xA
10
> 0x10
16
> 0xFF
255
```

ECMAScript 6 brings two new kinds of integer literals:

- Binary literals have the prefix `0b` or `0B`:

```
> 0b11
3
> 0b100
4
```

- Octal literals have the prefix `0o` or `0O` (that's a zero followed by the capital letter O; the first variant is safer):

```
> 0o7
7
> 0o10
8
```

Remember that the `Number` method `toString(radix)` can be used to see numbers in a base other than 10:

```
> 255..toString(16)
'ff'
> 4..toString(2)
'100'
> 8..toString(8)
'10'
```

(The double dots are necessary so that the dot for property access isn't confused with a decimal dot.)

5.2.1 Use case for octal literals: Unix-style file permissions

In the Node.js [file system module](#), several functions have the parameter `mode`. Its value is used to specify file permissions, via an encoding that is a holdover from Unix:

- Permissions are specified for three categories of users:
 - User: the owner of the file
 - Group: the members of the group associated with the file
 - All: everyone
- Per category, the following permissions can be granted:
 - `r` (read): the users in the category are allowed to read the file
 - `w` (write): the users in the category are allowed to change the file
 - `x` (execute): the users in the category are allowed to run the file

That means that permissions can be represented by 9 bits (3 categories with 3 permissions each):

| | User | Group | All |
|-------------|---------|---------|---------|
| Permissions | r, w, x | r, w, x | r, w, x |
| Bit | 8, 7, 6 | 5, 4, 3 | 2, 1, 0 |

The permissions of a single category of users are stored in 3 bits:

| Bits | Permissions | Octal digit |
|------|-------------|-------------|
| 000 | — | 0 |
| 001 | —x | 1 |
| 010 | —w— | 2 |
| 011 | —wx | 3 |
| 100 | r— | 4 |
| 101 | r-x | 5 |
| 110 | rw— | 6 |
| 111 | rwX | 7 |

That means that octal numbers are a compact representation of all permissions, you only need 3 digits, one digit per category of users. Two examples:

- 755 = 111,101,101: I can change, read and execute; everyone else can only read and execute.
- 640 = 110,100,000: I can read and write; group members can read; everyone can't access at all.

5.2.2 `Number.parseInt()` and the new integer literals

`Number.parseInt()` (which does the same as the global function `parseInt()`) has the following signature:

```
Number.parseInt(string, radix?)
```

5.2.2.1 `Number.parseInt()`: hexadecimal number literals

`Number.parseInt()` provides special support for the hexadecimal literal notation – the prefix `0x` (or `0X`) of `string` is removed if:

- `radix` is missing or 0. Then `radix` is set to 16. As a rule, you should never omit the `radix`.
- `radix` is 16.

For example:

```
> Number.parseInt('0xFF')
255
> Number.parseInt('0xFF', 0)
255
> Number.parseInt('0xFF', 16)
255
```

In all other cases, digits are only parsed until the first non-digit:

```
> Number.parseInt('0xFF', 10)
0
> Number.parseInt('0xFF', 17)
0
```

5.2.2.2 `Number.parseInt()`: binary and octal number literals

However, `Number.parseInt()` does not have special support for binary or octal literals!

```
> Number.parseInt('0b111')
0
> Number.parseInt('0b111', 2)
0
> Number.parseInt('111', 2)
7

> Number.parseInt('0o10')
0
> Number.parseInt('0o10', 8)
0
> Number.parseInt('10', 8)
8
```

If you want to parse these kinds of literals, you need to use `Number()`:

```
> Number('0b111')
7
> Number('0o10')
8
```

`Number.parseInt()` works fine with numbers that have a different base, as long as there is no special prefix and the parameter `radix` is provided:

```
> Number.parseInt('111', 2)
7
> Number.parseInt('10', 8)
8
```

5.3 New static `Number` properties

This section describes new properties that the constructor `Number` has picked up in ECMAScript 6.

5.3.1 Previously global functions

Four number-related functions are already available as global functions and have been added to `Number`, as methods: `isFinite` and `isNaN`, `parseFloat` and `parseInt`. All of them work almost the same as their global counterparts, but `isFinite` and `isNaN` don't coerce their arguments to numbers, anymore, which is especially important for `isNaN`. The following subsections explain all the details.

5.3.1.1 `Number.isFinite(number)`

`Number.isFinite(number)` determines whether `number` 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
```

The advantage of this method is that it does not coerce its parameter to number (whereas the global function does):

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

5.3.1.2 `Number.isNaN(number)`

`Number.isNaN(number)` checks whether `number` is the value `NaN`.

One ES5 way of making this check is via `!==`:

```
> const x = NaN;
> x !== x
true
```

A more descriptive way is via the global function `isNaN()`:

```
> const x = NaN;
> isNaN(x)
true
```

However, this function coerces non-numbers to numbers and returns `true` if the result is `NaN` (which is usually not what you want):

```
> isNaN('???')
true
```

The new method `Number.isNaN()` does not exhibit this problem, because it does not coerce its arguments to numbers:

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

5.3.1.3 `Number.parseFloat` and `Number.parseInt`

The following two methods work exactly like the global functions with the same names. They were added to `Number` for completeness sake; now all number-related functions are available there.

- `Number.parseFloat(string)`¹
- `Number.parseInt(string, radix)`²

5.3.2 `Number.EPSILON`

Especially with decimal fractions, rounding errors can become a problem in JavaScript³. For example, 0.1 and 0.2 can't be represented precisely, which you notice if you add them and compare them to 0.3 (which can't be represented precisely, either).

```
> 0.1 + 0.2 === 0.3
false
```

`Number.EPSILON` specifies a reasonable margin of error when comparing floating point numbers. It provides a better way to compare floating point values, as demonstrated by the following function.

```
function epsEqu(x, y) {
  return Math.abs(x - y) < Number.EPSILON;
}
console.log(epsEqu(0.1+0.2, 0.3)); // true
```

5.3.3 `Number.isInteger(number)`

JavaScript has only floating point numbers (doubles). Accordingly, integers are simply floating point numbers without a decimal fraction.

`Number.isInteger(number)` returns `true` if `number` 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

```

5.3.4 Safe integers

JavaScript numbers have only enough storage space to represent 53 bit signed integers. That is, integers i in the range $-2^{53} < i < 2^{53}$ are *safe*. What exactly that means is explained momentarily. The following properties help determine whether a JavaScript integer is safe:

- `Number.isSafeInteger(number)`
- `Number.MIN_SAFE_INTEGER`
- `Number.MAX_SAFE_INTEGER`

The notion of *safe integers* centers on how mathematical integers are represented in JavaScript. In the range $(-2^{53}, 2^{53})$ (excluding the lower and upper bounds), JavaScript integers are *safe*: there is a one-to-one mapping between them and the mathematical integers they represent.

Beyond this range, JavaScript integers are *unsafe*: two or more mathematical integers are represented as the same JavaScript integer. For example, starting at 2^{53} , JavaScript can represent only every second mathematical integer:

```

> Math.pow(2, 53)
9007199254740992

> 9007199254740992
9007199254740992
> 9007199254740993
9007199254740992
> 9007199254740994
9007199254740994
> 9007199254740995
9007199254740996
> 9007199254740996
9007199254740996
> 9007199254740997
9007199254740996

```

Therefore, a safe JavaScript integer is one that unambiguously represents a single mathematical integer.

5.3.4.1 Static `Number` properties related to safe integers

The two static `Number` properties specifying the lower and upper bound of safe integers could be defined as follows:

```

Number.MAX_SAFE_INTEGER = Math.pow(2, 53)-1;
Number.MIN_SAFE_INTEGER = -Number.MAX_SAFE_INTEGER;

```

`Number.isSafeInteger()` determines whether a JavaScript number is a safe integer and could be defined as follows:

```

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

```

For a given value n , this function first checks whether n is a number and an integer. If both checks succeed, n is safe if it is greater than or equal to `MIN_SAFE_INTEGER` and less than or equal to `MAX_SAFE_INTEGER`.

5.3.4.2 When are computations with integers correct?

How can we make sure that results of computations with integers are correct? For example, the following result is clearly not correct:

```

> 9007199254740990 + 3
9007199254740992

```

We have two safe operands, but an unsafe result:

```

> Number.isSafeInteger(9007199254740990)
true
> Number.isSafeInteger(3)
true
> Number.isSafeInteger(9007199254740992)
false

```

The following result is also incorrect:

```
> 9007199254740995 - 10
9007199254740986
```

This time, the result is safe, but one of the operands isn't:

```
> Number.isSafeInteger(9007199254740995)
false
> Number.isSafeInteger(10)
true
> Number.isSafeInteger(9007199254740986)
true
```

Therefore, the result of applying an integer operator `op` is guaranteed to be correct only if all operands and the result are safe. More formally:

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

implies that `a op b` is a correct result.



Source of this section

"Clarify integer and safe integer resolution", email by Mark S. Miller to the es-discuss mailing list.

5.4 Math

The global object `Math` has several new methods in ECMAScript 6.

5.4.1 Various numerical functionality

5.4.1.1 `Math.sign(x)`

`Math.sign(x)` returns:

- -1 if `x` is a negative number (including `-Infinity`).
- 0 if `x` is zero⁴.
- +1 if `x` is a positive number (including `Infinity`).
- NaN if `x` is NaN or not a number.

Examples:

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

5.4.1.2 `Math.trunc(x)`

`Math.trunc(x)` removes the decimal fraction of `x`. Complements the other rounding methods `Math.floor()`, `Math.ceil()` and `Math.round()`.

```
> Math.trunc(3.1)
3
> Math.trunc(3.9)
3
> Math.trunc(-3.1)
-3
> Math.trunc(-3.9)
-3
```

You could implement `Math.trunc()` like this:

```
function trunc(x) {
  return Math.sign(x) * Math.floor(Math.abs(x));
}
```

5.4.1.3 `Math.cbrt(x)`

`Math.cbrt(x)` returns the cube root of `x` ($\sqrt[3]{x}$).

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

5.4.2 Using 0 instead of 1 with exponentiation and logarithm

A small fraction can be represented more precisely if it comes after zero. I'll demonstrate this with decimal fractions (JavaScript's numbers are internally stored with base 2, but the same reasoning applies).

Floating point numbers with base 10 are internally represented as *mantissa* $\times 10^{\text{exponent}}$. The *mantissa* has a single digit before the decimal dot and the *exponent* “moves” the dot as necessary. That means if you convert a small fraction to the internal representation, a zero before the dot leads to a smaller mantissa than a one before the dot. For example:

- (A) $0.000000234 = 2.34 \times 10^{-7}$. Significant digits: 234
- (B) $1.000000234 = 1.000000234 \times 10^0$. Significant digits: 1000000234

Precision-wise, the important quantity here is the capacity of the mantissa, as measured in significant digits. That’s why (A) gives you higher precision than (B).

Additionally, JavaScript represents numbers close to zero (e.g. small fractions) with higher precision.

5.4.2.1 `Math.expml(x)`

`Math.expml(x)` returns `Math.exp(x)-1`. The inverse of `Math.loglp()`.

Therefore, this method provides higher precision whenever `Math.exp()` has results close to 1. You can see the difference between the two in the following interaction:

```
> Math.expml(1e-10)
1.00000000005e-10
> Math.exp(1e-10)-1
1.000000082740371e-10
```

The former is the better result, which you can verify by using a library (such as [decimal.js](#)) for floating point numbers with arbitrary precision (“bigfloats”):

```
> var Decimal = require('decimal.js').config({precision:50});
> new Decimal(1e-10).exp().minus(1).toString()
'1.000000000050000000001666666666708333333e-10'
```

5.4.2.2 `Math.loglp(x)`

`Math.loglp(x)` returns `Math.log(1 + x)`. The inverse of `Math.expml()`.

Therefore, this method lets you specify parameters that are close to 1 with a higher precision. The following examples demonstrate why that is.

The following two calls of `log()` produce the same result:

```
> Math.log(1 + 1e-16)
0
> Math.log(1 + 0)
0
```

In contrast, `loglp()` produces different results:

```
> Math.loglp(1e-16)
1e-16
> Math.loglp(0)
0
```

The reason for the higher precision of `Math.loglp()` is that the correct result for `1 + 1e-16` has more significant digits than `1e-16`:

```
> 1 + 1e-16 === 1
true
> 1e-16 === 0
false
```

5.4.3 Logarithms to base 2 and 10

5.4.3.1 `Math.log2(x)`

`Math.log2(x)` computes the logarithm to base 2.

```
> Math.log2(8)
3
```

5.4.3.2 `Math.log10(x)`

`Math.log10(x)` computes the logarithm to base 10.

```
> Math.log10(100)
2
```

5.4.4 Support for compiling to JavaScript

[Emscripten](#) pioneered a coding style that was later picked up by [asm.js](#): The operations of a virtual machine (think bytecode) are expressed in static subset of JavaScript. That subset can be executed efficiently by JavaScript engines: If it is the result of a compilation from C++, it runs at about 70% of native speed.

The following `Math` methods were mainly added to support `asm.js` and similar compilation strategies, they are not that useful for other applications.

5.4.4.1 `Math.fround(x)`

`Math.fround(x)` rounds `x` to a 32 bit floating point value (`float`). Used by `asm.js` to tell an engine to internally use a `float` value.

5.4.4.2 `Math.imul(x, y)`

`Math.imul(x, y)` multiplies the two 32 bit integers `x` and `y` and returns the lower 32 bits of the result. This is the only 32 bit basic math operation that can't be simulated by using a JavaScript operator and coercing the result back to 32 bits. For example, `idiv` could be implemented as follows:

```
function idiv(x, y) {  
  return (x / y) | 0;  
}
```

In contrast, multiplying two large 32 bit integers may produce a double that is so large that lower bits are lost.

5.4.5 Bitwise operations

- `Math.clz32(x)`
Counts the leading zero bits in the 32 bit integer `x`.

```
> Math.clz32(0b01000000000000000000000000000000)  
1  
> Math.clz32(0b00100000000000000000000000000000)  
2  
> Math.clz32(2)  
30  
> Math.clz32(1)  
31
```

Why is this interesting? Quoting “[Fast, Deterministic, and Portable Counting Leading Zeros](#)” by Miro Samek:

Counting leading zeros in an integer number is a critical operation in many DSP algorithms, such as normalization of samples in sound or video processing, as well as in real-time schedulers to quickly find the highest-priority task ready-to-run.

5.4.6 Trigonometric methods

- `Math.sinh(x)`
Computes the hyperbolic sine of `x`.
- `Math.cosh(x)`
Computes the hyperbolic cosine of `x`.
- `Math.tanh(x)`
Computes the hyperbolic tangent of `x`.
- `Math.asinh(x)`
Computes the inverse hyperbolic sine of `x`.
- `Math.acosh(x)`
Computes the inverse hyperbolic cosine of `x`.
- `Math.atanh(x)`
Computes the inverse hyperbolic tangent of `x`.
- `Math.hypot(...values)`
Computes the square root of the sum of the squares of its arguments (Pythagoras' theorem):

```
> Math.hypot(3, 4)  
5
```

5.5 FAQ: numbers

5.5.1 How can I use integers beyond JavaScript's 53 bit range?

JavaScript's integers have a range of 53 bits. That is a problem whenever 64 bit integers are needed. For example: In its JSON API, Twitter had to switch from integers to strings when tweet IDs became too large.

At the moment, the only way around that limitation is to use a library for higher-precision numbers (bigints or bigfloats). One such library is [decimal.js](#).

Plans to support larger integers in JavaScript exist, but may take a while to come to fruition.

6. New string features

6.1 Overview

New string methods:

```
> 'hello'.startsWith('hell')
true
> 'hello'.endsWith('ello')
true
> 'hello'.includes('ell')
true
> 'doo '.repeat(3)
'doo doo doo '
```

ES6 has a new kind of string literal, the *template literal*:

```
// String interpolation via template literals (in backticks)
const first = 'Jane';
const last = 'Doe';
console.log(`Hello ${first} ${last}!`);
// Hello Jane Doe!

// Template literals also let you create strings with multiple lines
const multiLine = `
This is
a string
with multiple
lines`;
```

6.2 Unicode code point escapes

In ECMAScript 6, there is a new kind of Unicode escape that lets you specify any code point (even those beyond 16 bits):

```
console.log(`\u{1F680}`); // ES6: single code point
console.log(`\uD83D\uDE80`); // ES5: two code units
```

More information on escapes is given in [the chapter on Unicode](#).

6.3 String interpolation, multi-line string literals and raw string literals

Template literals are described in depth in [their own chapter](#). They provide three interesting features.

First, template literals support string interpolation:

```
const first = 'Jane';
const last = 'Doe';
console.log(`Hello ${first} ${last}!`);
// Hello Jane Doe!
```

Second, template literals can contain multiple lines:

```
const multiLine = `
This is
a string
with multiple
lines`;
```

Third, template literals are “raw” if you prefix them with the *tag* `String.raw` – the backslash is not a special character and escapes such as `\n` are not interpreted:

```
const str = String.raw`Not a newline: \n`;
console.log(str === 'Not a newline: \n'); // true
```

6.4 Iterating over strings

Strings are *iterable*, which means that you can use `for-of` to iterate over their characters:

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

And you can use the spread operator (`...`) to turn strings into Arrays:

```
const chars = [...'abc'];
// ['a', 'b', 'c']
```

6.4.1 Iteration honors Unicode code points

The string iterator splits strings along code point boundaries, which means that the strings it returns comprise one or two JavaScript characters:

```
for (const ch of 'x\uD83D\uDE80y') {
  console.log(ch.length);
}
// Output:
// 1
// 2
// 1
```

6.4.2 Counting code points

Iteration gives you a quick way to count the Unicode code points in a string:

```
> [...'x\uD83D\uDE80y'].length
3
```


6.4.3 Reversing strings with non-BMP code points

Iteration also helps with reversing strings that contain non-BMP code points (which are larger than 16 bit and encoded as two JavaScript characters):

```
const str = 'x\uD83D\uDE80y';

// ES5: \uD83D\uDE80 are (incorrectly) reversed
console.log(str.split('').reverse().join(''));
// 'y\uDE80\uD83Dx'

// ES6: order of \uD83D\uDE80 is preserved
console.log([...str].reverse().join(''));
// 'y\uD83D\uDE80x'
```



The two reversed strings in the Firefox console.



Remaining problem: combining marks

A *combining mark* is a sequence of two Unicode code points that is displayed as single symbol. The ES6 approach to reversing a string that I have presented here works for non-BMP code points, but not for combining marks. For those, you need a library, e.g. Mathias Bynens' [Esrever](#).

6.5 Numeric values of code points

The new method `codePointAt()` returns the numeric value of a code point at a given index in a string:

```
const str = 'x\uD83D\uDE80y';
console.log(str.codePointAt(0).toString(16)); // 78
console.log(str.codePointAt(1).toString(16)); // 1f680
console.log(str.codePointAt(3).toString(16)); // 79
```

This method works well when combined with iteration over strings:

```
for (const ch of 'x\uD83D\uDE80y') {
  console.log(ch.codePointAt(0).toString(16));
}
// Output:
// 78
// 1f680
// 79
```

The opposite of `codePointAt()` is `String.fromCodePoint()`:

```
> String.fromCodePoint(0x78, 0x1f680, 0x79) === 'x\uD83D\uDE80y'
true
```

6.6 Checking for inclusion

Three new methods check whether a string exists within another string:

```
> 'hello'.startsWith('hell')
true
> 'hello'.endsWith('ello')
true
> 'hello'.includes('ell')
true
```

Each of these methods has a position as an optional second parameter, which specifies where the string to be searched starts or ends:

```
> 'hello'.startsWith('ello', 1)
true
> 'hello'.endsWith('hell', 4)
true

> 'hello'.includes('ell', 1)
true
> 'hello'.includes('ell', 2)
false
```

6.7 Repeating strings

The `repeat()` method repeats strings:

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

6.8 String methods that delegate regular expression work to their parameters

In ES6, the four string methods that accept regular expression parameters do relatively little. They mainly call methods of their parameters:

- `String.prototype.match(regex)` calls `regex[Symbol.match](this)`.
- `String.prototype.replace(searchValue, replaceValue)` calls `searchValue[Symbol.replace](this, replaceValue)`.
- `String.prototype.search(regex)` calls `regex[Symbol.search](this)`.
- `String.prototype.split(separator, limit)` calls `separator[Symbol.split](this, limit)`.

The parameters don't have to be regular expressions, anymore. Any objects with appropriate methods will do.

6.9 Cheat sheet: the new string methods

Tagged templates:

- `String.raw(callSite, ...substitutions) : string`
Template tag for "raw" content (backslashes are not interpreted):

```
> String.raw`\` == '\\'
true
```

Consult [the chapter on template literals](#) for more information.

Unicode and code points:

- `String.fromCodePoint(...codePoints : number[]) : string`
Turns numbers denoting Unicode code points into a string.
- `String.prototype.codePointAt(pos) : number`
Returns the number of the code point starting at position `pos` (comprising one or two JavaScript characters).
- `String.prototype.normalize(form? : string) : string`
Different combinations of code points may look the same. [Unicode normalization](#) changes them all to the same value(s), their so-called *canonical representation*. That helps with comparing and searching for strings. The 'NFC' form is recommended for general text.

Finding strings:

- `String.prototype.startsWith(searchString, position=0) : boolean`
Does the receiver start with `searchString`? `position` lets you specify where the string to be checked starts.
- `String.prototype.endsWith(searchString, endPosition=searchString.length) : boolean`
Does the receiver end with `searchString`? `endPosition` lets you specify where the string to be checked ends.
- `String.prototype.includes(searchString, position=0) : boolean`
Does the receiver contain `searchString`? `position` lets you specify where the string to be searched starts.

Repeating strings:

- `String.prototype.repeat(count) : string`
Returns the receiver, concatenated `count` times.

7. Symbols

7.1 Overview

Symbols are a new primitive type in ECMAScript 6.

7.1.1 Use case 1: unique property keys

Symbols are mainly used as unique property keys – a symbol never clashes with any other property key (symbol or string). For example, you can make an object *iterable* (usable via the `for-of` loop and other language mechanisms), by using the symbol stored in `Symbol.iterator` as the key of a method (more information on iterables is given in [the chapter on iteration](#)):

```
const iterableObject = {
  [Symbol.iterator]() { // (A)
    const data = ['hello', 'world'];
    let index = 0;
    return {
      next() {
        if (index < data.length) {
          return { value: data[index++] };
        } else {
          return { done: true };
        }
      }
    };
  }
};
```



```

    }
  }
  for (const x of iterableObject) {
    console.log(x);
  }
}
// Output:
// hello
// world

```

In line A, a symbol is used as the key of the method. This unique marker makes the object iterable and enables us to use the `for-of` loop.

7.1.2 Use case 2: constants representing concepts

In ECMAScript 5, you may have used strings to represent concepts such as colors. In ES6, you can use symbols and be sure that they are always unique:

```

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);
  }
}

```

7.1.3 Pitfall: you can't coerce symbols to strings

Coercing (implicitly converting) symbols to strings throws exceptions:

```

const sym = Symbol('desc');

const str1 = '' + sym; // TypeError
const str2 = `${sym}`; // TypeError

```

The only solution is to convert explicitly:

```

const str2 = String(sym); // 'Symbol(desc)'
const str3 = sym.toString(); // 'Symbol(desc)'

```

Forbidding coercion prevents some errors, but also makes working with symbols more complicated.

7.1.4 Which operations related to property keys are aware of symbols?

The following operations are aware of symbols as property keys:

- `Reflect.ownKeys()`
- Property access via `[]`
- `Object.assign()`

The following operations ignore symbols as property keys:

- `Object.keys()`
- `Object.getOwnPropertyNames()`
- `for-in` loop

7.2 A new primitive type

ECMAScript 6 introduces a new primitive type: symbols. They are tokens that serve as unique IDs. You create symbols via the factory function `Symbol()` (which is loosely similar to `String` returning strings if called as a function):

```

const symbol1 = Symbol();

```

`Symbol()` has an optional string-valued parameter that lets you give the newly created `Symbol` a description. That description is used when the symbol is converted to a string (via `toString()` or `String()`):

```

> const symbol2 = Symbol('symbol2');
> String(symbol2)
'Symbol(symbol2)'

```

Every symbol returned by `Symbol()` is unique, every symbol has its own identity:

```

> Symbol() === Symbol()
false

```

You can see that symbols are primitive if you apply the `typeof` operator to one of them – it will return a new symbol-specific result:

```
> typeof Symbol()
'symbol'
```

7.2.1 Symbols as property keys

Symbols can be used as property keys:

```
const MY_KEY = Symbol();
const obj = {};

obj[MY_KEY] = 123;
console.log(obj[MY_KEY]); // 123
```

Classes and object literals have a feature called *computed property keys*: You can specify the key of a property via an expression, by putting it in square brackets. In the following object literal, we use a computed property key to make the value of `MY_KEY` the key of a property.

```
const MY_KEY = Symbol();
const obj = {
  [MY_KEY]: 123
};
```

A method definition can also have a computed key:

```
const FOO = Symbol();
const obj = {
  [FOO]() {
    return 'bar';
  }
};
console.log(obj[FOO]()); // bar
```

7.2.2 Enumerating own property keys

Given that there is now a new kind of value that can become the key of a property, the following terminology is used for ECMAScript 6:

- *Property keys* are either strings or symbols.
- String-valued property keys are called *property names*.
- Symbol-valued property keys are called *property symbols*.

Let's examine the API for enumerating own property keys by first creating an object.

```
const obj = {
  [Symbol('my_key')]: 1,
  enum: 2,
  nonEnum: 3
};
Object.defineProperty(obj,
  'nonEnum', { enumerable: false });
```

`Object.getOwnPropertyNames()` ignores symbol-valued property keys:

```
> Object.getOwnPropertyNames(obj)
['enum', 'nonEnum']
```

`Object.getOwnPropertySymbols()` ignores string-valued property keys:

```
> Object.getOwnPropertySymbols(obj)
[Symbol(my_key)]
```

`Reflect.ownKeys()` considers all kinds of keys:

```
> Reflect.ownKeys(obj)
[Symbol(my_key), 'enum', 'nonEnum']
```

`Object.keys()` only considers enumerable property keys that are strings:

```
> Object.keys(obj)
['enum']
```

The name `Object.keys` clashes with the new terminology (only string keys are listed). `Object.names` or `Object.getEnumerableOwnPropertyNames` would be a better choice now.

7.3 Using symbols to represent concepts

In ECMAScript 5, one often represents concepts (think enum constants) via strings. For example:

```
var COLOR_RED = 'Red';
var COLOR_ORANGE = 'Orange';
var COLOR_YELLOW = 'Yellow';
var COLOR_GREEN = 'Green';
var COLOR_BLUE = 'Blue';
var COLOR_VIOLET = 'Violet';
```

However, strings are not as unique as we'd like them to be. To see why, let's look at the following function.

```
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);
  }
}
```

It is noteworthy that you can use arbitrary expressions as `switch` cases, you are not limited in any way. For example:

```
function isThree(x) {
  switch (x) {
    case 1 + 1 + 1:
      return true;
    default:
      return false;
  }
}
```

We use the flexibility that `switch` offers us and refer to the colors via our constants (`COLOR_RED` etc.) instead of hard-coding them (`'RED'` etc.).

Interestingly, even though we do so, there can still be mix-ups. For example, someone may define a constant for a mood:

```
var MOOD_BLUE = 'BLUE';
```

Now the value of `BLUE` is not unique anymore and `MOOD_BLUE` can be mistaken for it. If you use it as a parameter for `getComplement()`, it returns `'ORANGE'` where it should throw an exception.

Let's use symbols to fix this example. Now we can also use the ES6 feature `const`, which lets us declare actual constants (you can't change what value is bound to a constant, but the value itself may be mutable).

```
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');
```

Each value returned by `Symbol` is unique, which is why no other value can be mistaken for `BLUE` now. Intriguingly, the code of `getComplement()` doesn't change at all if we use symbols instead of strings, which shows how similar they are.

7.4 Symbols as keys of properties

Being able to create properties whose keys never clash with other keys is useful in two situations:

- For non-public properties in inheritance hierarchies.
- To keep meta-level properties from clashing with base-level properties.

7.4.1 Symbols as keys of non-public properties

Whenever there are inheritance hierarchies in JavaScript (e.g. created via classes, mixins or a purely prototypal approach), you have two kinds of properties:

- *Public properties* are seen by clients of the code.
- *Private properties* are used internally within the pieces (e.g. classes, mixins or objects) that make up the inheritance hierarchy. (*Protected properties* are shared between several pieces and face the same issues as private properties.)

For usability's sake, public properties usually have string keys. But for private properties with string keys, accidental name clashes can become a problem. Therefore, symbols are a good choice. For example, in the following code, symbols are used for the private properties `_counter` and `_action`.

```
const _counter = Symbol('counter');
const _action = Symbol('action');
class Countdown {
  constructor(counter, action) {
    this[_counter] = counter;
    this[_action] = action;
  }
  dec() {
    let counter = this[_counter];
    if (counter < 1) return;
    counter--;
    this[_counter] = counter;
    if (counter === 0) {
```

```

        this[_action]();
    }
}

```

Note that symbols only protect you from name clashes, not from unauthorized access, because you can find out all own property keys – including symbols – of an object via `Reflect.ownKeys()`. If you want protection there, as well, you can use one of the approaches listed in Sect. “[Private data for classes](#)”.

7.4.2 Symbols as keys of meta-level properties

Symbols having unique identities makes them ideal as keys of public properties that exist on a different level than “normal” property keys, because meta-level keys and normal keys must not clash. One example of meta-level properties are methods that objects can implement to customize how they are treated by a library. Using symbol keys protects the library from mistaking normal methods as customization methods.

Iterability in ECMAScript 6 is one such customization. An object is *iterable* if it has a method whose key is the symbol (stored in) `Symbol.iterator`. In the following code, `obj` is iterable.

```

const obj = {
  data: [ 'hello', 'world' ],
  [Symbol.iterator]() {
    const self = this;
    let index = 0;
    return {
      next() {
        if (index < self.data.length) {
          return {
            value: self.data[index++]
          };
        } else {
          return { done: true };
        }
      }
    };
  }
};

```

The iterability of `obj` enables you to use the `for-of` loop and similar JavaScript features:

```

for (const x of obj) {
  console.log(x);
}

// Output:
// hello
// world

```

7.4.3 Examples of name clashes in JavaScript’s standard library

In case you think that name clashes don’t matter, here are three examples of where name clashes caused problems in the evolution of the JavaScript standard library:

- When the new method `Array.prototype.values()` was created, it broke existing code where `with` was used with an Array and shadowed a variable `values` in an outer scope ([bug report 1](#), [bug report 2](#)). Therefore, a mechanism was introduced to hide properties from `with` (`Symbol.unscopables`).
- `String.prototype.contains` clashed with a method added by MooTools and had to be renamed to `String.prototype.includes` ([bug report](#)).
- The ES2016 method `Array.prototype.contains` also clashed with a method added by MooTools and had to be renamed to `Array.prototype.includes` ([bug report](#)).

In contrast, [adding iterability to an object via the property key `Symbol.iterator`](#) can’t cause problems, because that key doesn’t clash with anything.

These examples demonstrate what it means to be a web language: backward compatibility is crucial, which is why compromises are occasionally necessary when evolving the language. As a side benefit, evolving old JavaScript code bases is simpler, too, because new ECMAScript versions never (well, hardly ever) break them.

7.5 Converting symbols to other primitive types

The following table shows what happens if you explicitly or implicitly convert symbols to other primitive types:

| Conversion to | Explicit conversion | Coercion (implicit conversion) |
|---------------|---|---|
| boolean | <code>Boolean(sym)</code> → OK | <code>!sym</code> → OK |
| number | <code>Number(sym)</code> → <code>TypeError</code> | <code>sym*2</code> → <code>TypeError</code> |
| string | <code>String(sym)</code> → OK <code>sym.toString()</code> → OK | <code>''+sym</code> → <code>TypeError</code> <code>`\${sym}`</code> → <code>TypeError</code> |

7.5.1 Pitfall: coercion to string

Coercion to string being forbidden can easily trip you up:

```
const sym = Symbol();

console.log('A symbol: '+sym); // TypeError
console.log(`A symbol: ${sym}`); // TypeError
```

To fix these problems, you need an explicit conversion to string:

```
console.log('A symbol: '+String(sym)); // OK
console.log(`A symbol: ${String(sym)}`); // OK
```

7.5.2 Making sense of the coercion rules

Coercion (implicit conversion) is often forbidden for symbols. This section explains why.

7.5.2.1 Truthiness checks are allowed

Coercion to boolean is always allowed, mainly to enable truthiness checks in `if` statements and other locations:

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

param = param || 0;
```

7.5.2.2 Accidentally turning symbols into property keys

Symbols are special property keys, which is why you want to avoid accidentally converting them to strings, which are a different kind of property keys. This could happen if you use the addition operator to compute the name of a property:

```
myObject['_' + value]
```

That's why a `TypeError` is thrown if `value` is a symbol.

7.5.2.3 Accidentally turning symbols into Array indices

You also don't want to accidentally turn symbols into Array indices. The following is code where that could happen if `value` is a symbol:

```
myArray[1 + value]
```

That's why the addition operator throws an error in this case.

7.5.3 Explicit and implicit conversion in the spec

7.5.3.1 Converting to boolean

To explicitly convert a symbol to boolean, you call `Boolean()`, which returns `true` for symbols:

```
> const sym = Symbol('hello');
> Boolean(sym)
true
```

`Boolean()` computes its result via the internal operation `ToBoolean()`, which returns `true` for symbols and other truthy values.

Coercion also uses `ToBoolean()`:

```
> !sym
false
```

7.5.3.2 Converting to number

To explicitly convert a symbol to number, you call `Number()`:

```
> const sym = Symbol('hello');
> Number(sym)
TypeError: can't convert symbol to number
```

`Number()` computes its result via the internal operation `ToNumber()`, which throws a `TypeError` for symbols.

Coercion also uses `ToNumber()`:

```
> +sym
TypeError: can't convert symbol to number
```

7.5.3.3 Converting to string

To explicitly convert a symbol to string, you call `String()`:

```
> const sym = Symbol('hello');
> String(sym)
'Symbol(hello)'
```

If the parameter of `String()` is a symbol then it handles the conversion to string itself and returns the string `Symbol()` wrapped around the description that was provided when creating the symbol. If no description was given, the empty string is used:

```
> String(Symbol())
'Symbol()'
```

The `toString()` method returns the same string as `String()`, but neither of these two operations calls the other one, they both call the same internal operation `SymbolDescriptiveString()`.

```
> Symbol('hello').toString()
'Symbol(hello)'
```

Coercion is handled via the internal operation `ToString()`, which throws a `TypeError` for symbols. One method that coerces its parameter to string is `Number.parseInt()`:

```
> Number.parseInt(Symbol())
TypeError: can't convert symbol to string
```

7.5.3.4 Not allowed: converting via the binary addition operator (+)

The `addition operator` works as follows:

- Convert both operands to primitives.
- If one of the operands is a string, coerce both operands to strings (via `ToString()`), concatenate them and return the result.
- Otherwise, coerce both operands to numbers, add them and return the result.

Coercion to either string or number throws an exception, which means that you can't (directly) use the addition operator for symbols:

```
> '' + Symbol()
TypeError: can't convert symbol to string
> 1 + Symbol()
TypeError: can't convert symbol to number
```

7.6 JSON and symbols

7.6.1 Generating JSON via `JSON.stringify()`

`JSON.stringify()` converts JavaScript data to JSON strings. A preprocessing step lets you customize that conversion: a callback, a so-called *replacer*, can replace any value inside the JavaScript data with another one. That means that it can encode JSON-incompatible values (such as symbols and dates) as JSON-compatible values (such as strings). `JSON.parse()` lets you reverse this process via a similar mechanism⁵.

However, `stringify` ignores non-string property keys, so this approach works only if symbols are property values. For example, like this:

```
function symbolReplacer(key, value) {
  if (typeof value === 'symbol') {
    return '@@' + Symbol.keyFor(value) + '@@';
  }
  return value;
}
const MY_SYMBOL = Symbol.for('http://example.com/my_symbol');
const obj = { myKey: MY_SYMBOL };

const str = JSON.stringify(obj, symbolReplacer);
console.log(str);
// {"myKey":"@@http://example.com/my_symbol@@"}

```

A symbol is encoded as a string by putting '@@' before and after the symbol's key. Note that only symbols that were created via `Symbol.for()` have such a key.

7.6.2 Parsing JSON via `JSON.parse()`

`JSON.parse()` converts JSON strings to JavaScript data. A postprocessing step lets you customize that conversion: a callback, a so-called *reviver*, can replace any value inside the initial output with another one. That means that it can decode non-JSON data (such as symbols and dates) stored in JSON data (such as strings)⁶. This looks as follows.

```
const REGEX_SYMBOL_STRING = /^@@(.*)@@$/;
function symbolReviver(key, value) {
  if (typeof value === 'string') {

```

```

    const match = REGEX_SYMBOL_STRING.exec(value);
    if (match) {
      const symbolKey = match[1];
      return Symbol.for(symbolKey);
    }
  }
  return value;
}

const parsed = JSON.parse(str, symbolReviver);
console.log(parse);

```

Strings that start and end with '@@' are converted to symbols by extracting the symbol key in the middle.

7.7 Wrapper objects for symbols

While all other primitive values have literals, you need to create symbols by function-calling `Symbol`. Thus, it is relatively easy to accidentally invoke `Symbol` as a constructor. That produces instances of `Symbol` and is not very useful. Therefore, an exception is thrown when you try to do that:

```

> new Symbol()
TypeError: Symbol is not a constructor

```

There is still a way to create wrapper objects, instances of `Symbol`: `Object`, called as a function, converts all values to objects, including symbols.

```

> const sym = Symbol();
> typeof sym
'symbol'

> const wrapper = Object(sym);
> typeof wrapper
'object'
> wrapper instanceof Symbol
true

```

7.7.1 Accessing properties via [] and wrapped keys

The square bracket operator `[]` for accessing properties unwraps string wrapper objects and symbol wrapper objects. Let's use the following object to examine this phenomenon.

```

const sym = Symbol('yes');
const obj = {
  [sym]: 'a',
  str: 'b',
};

```

Interaction:

```

> const wrappedSymbol = Object(sym);
> typeof wrappedSymbol
'object'
> obj[wrappedSymbol]
'a'

> const wrappedString = new String('str');
> typeof wrappedString
'object'
> obj[wrappedString]
'b'

```

7.7.1.1 Property access in the spec

The operator for getting and setting properties uses the internal operation `ToPropertyKey()`, which works as follows:

- Convert the operand to a primitive via `ToPrimitive()` with the preferred type `string`:
 - Primitive values are returned as is.
 - Most objects are converted via the method `toString()` – if it returns a primitive value. Otherwise, `valueOf()` is used – if it returns a primitive value. Otherwise, a `TypeError` is thrown.
 - Symbol objects are one exception: they are converted to the symbols that they wrap.
- If the result is a symbol, return it.
- Otherwise, coerce the result to string via `ToString()`.

7.8 Crossing realms with symbols



This is an advanced topic.

A *code realm* (short: realm) is a context in which pieces of code exist. It includes global variables, loaded modules and more. Even though code exists “inside” exactly one realm, it may have access to code in other realms. For example, each frame in a

browser has its own realm. And execution can jump from one frame to another, as the following HTML demonstrates.

```
<head>
  <script>
    function test(arr) {
      var iframe = frames[0];
      // This code and the iframe's code exist in
      // different realms. Therefore, global variables
      // such as Array are different:
      console.log(Array === iframe.Array); // false
      console.log(arr instanceof Array); // false
      console.log(arr instanceof iframe.Array); // true

      // But: symbols are the same
      console.log(Symbol.iterator ===
        iframe.Symbol.iterator); // true
    }
  </script>
</head>
<body>
  <iframe srcdoc="<script>window.parent.test([])</script>">
</iframe>
</body>
```

The problem is that each realm has its own local copy of `Array` and, because objects have individual identities, those local copies are considered different, even though they are essentially the same object. Similarly, libraries and user code are loaded once per realm and each realm has a different version of the same object.

In contrast, members of the primitive types boolean, number and string don't have individual identities and multiple copies of the same value are not a problem: The copies are compared "by value" (by looking at the content, not at the identity) and are considered equal.

Symbols have individual identities and thus don't travel across realms as smoothly as other primitive values. That is a problem for symbols such as `Symbol.iterator` that should work across realms: If an object is iterable in one realm, it should be iterable in others, too. If a cross-realm symbol is managed by the JavaScript engine, the engine can make sure that the same value is used in each realm. For libraries, however, we need extra support, which comes in the form of the *global symbol registry*: This registry is global to all realms and maps strings to symbols. For each symbol, libraries need to come up with a string that is as unique as possible. To create the symbol, they don't use `Symbol()`, they ask the registry for the symbol that the string is mapped to. If the registry already has an entry for the string, the associated symbol is returned. Otherwise, entry and symbol are created first.

You ask the registry for a symbol via `Symbol.for()` and retrieve the string associated with a symbol (its *key*) via `Symbol.keyFor()`:

```
> const sym = Symbol.for('Hello everybody!');
> Symbol.keyFor(sym)
'Hello everybody!'
```

Cross-realm symbols, such as `Symbol.iterator`, that are provided by the JavaScript engine are not in the registry:

```
> Symbol.keyFor(Symbol.iterator)
undefined
```

7.9 FAQ: symbols

7.9.1 Can I use symbols to define private properties?

The original plan was for symbols to support private properties (there would have been public and private symbols). But that feature was dropped, because using "get" and "set" (two meta-object protocol operations) for managing private data does not interact well with proxies:

- On one hand, you want a proxy to be able to completely isolate its target (for [membranes](#)) and to intercept all MOP operations applied to its target.
- On the other hand, proxies should not be able to extract private data from an object; private data should remain private.

These two goals are at odds.

The chapter on classes explains [your options for managing their private data](#).

7.9.2 Are symbols primitives or objects?

In some ways, symbols are like primitive values, in other ways, they are like objects:

- Symbols are like strings (primitive values) w.r.t. what they are used for: as representations of concepts and as property keys.
- Symbols are like objects in that each symbol has its own identity.

What are symbols then – primitive values or objects? In the end, they were turned into primitives, for two reasons.

First, symbols are more like strings than like objects: They are a fundamental value of the language, they are immutable and they can be used as property keys. Symbols having unique identities doesn't necessarily contradict them being like strings: UUID algorithms produce strings that are quasi-unique.

Second, symbols are most often used as property keys, so it makes sense to optimize the JavaScript specification and the implementations for that use case. Then many abilities of objects are unnecessary:

- Objects can become prototypes of other objects.
- Wrapping an object with a proxy doesn't change what it can be used for.
- Objects can be introspected: via `instanceof`, `Object.keys()`, etc.

Them not having these abilities makes life easier for the specification and the implementations. There are also reports from the V8 team that when handling property keys, it is simpler to treat primitives differently than objects.

7.9.3 Do we really need symbols? Aren't strings enough?

In contrast to strings, symbols are unique and prevent name clashes. That is nice to have for tokens such as colors, but it is essential for supporting meta-level methods such as the one whose key is `Symbol.iterator`. Python uses the special name `__iter__` to avoid clashes. You can reserve double underscore names for programming language mechanisms, but what is a library to do? With symbols, we have an extensibility mechanism that works for everyone. As you can see later, in the section on public symbols, JavaScript itself already makes ample use of this mechanism.

There is one hypothetical alternative to symbols when it comes to clash-free property keys: use a naming convention. For example, strings with URLs (e.g. `'http://example.com/iterator'`). But that would introduce a second category of property keys (versus “normal” property names that are usually valid identifiers and don't contain colons, slashes, dots, etc.), which is basically what symbols are, anyway. Thus it is more elegant to explicitly turn those keys into a different kind of value.

7.9.4 Are JavaScript's symbols like Ruby's symbols?

No, they are not.

Ruby's symbols are basically literals for creating values. Mentioning the same symbol twice produces the same value twice:

```
:foo == :foo
```

The JavaScript function `Symbol()` is a factory for symbols – each value it returns is unique:

```
Symbol('foo') !== Symbol('foo')
```

7.10 The spelling of well-known symbols: why `Symbol.iterator` and not `Symbol.ITERATOR` (etc.)?

Well-known symbols are stored in properties whose names start with lowercase characters and are camel-cased. In a way, these properties are constants and it is customary for constants to have all-caps names (`Math.PI` etc.). But the reasoning for their spelling is different: Well-known symbols are used instead of normal property keys, which is why their “names” follow the rules for property keys, not the rules for constants.

7.11 The symbol API

This section gives an overview of the ECMAScript 6 API for symbols.

7.11.1 The function `Symbol`

- `Symbol(description?)` : symbol
Creates a new symbol. The optional parameter `description` allows you to give the symbol a description. The only way to access the description is to convert the symbol to a string (via `toString()` or `String()`). The result of such a conversion is `'Symbol('+description+')'`.

`Symbol` is can't be used as a constructor – an exception is thrown if you invoke it via `new`.

7.11.2 Methods of symbols

The only useful method that symbols have is `toString()` (as provided via `Symbol.prototype.toString()`).

7.11.3 Converting symbols to other values

| Conversion to | Explicit conversion | Coercion (implicit conversion) |
|---------------|---|---|
| boolean | <code>Boolean(sym) → OK</code> | <code>!sym → OK</code> |
| number | <code>Number(sym) → TypeError</code> | <code>sym*2 → TypeError</code> |
| string | <code>String(sym) → OK</code> <code>sym.toString() → OK</code> | <code>''+sym → TypeError</code> <code>` \${sym} ` → TypeError</code> |
| object | <code>Object(sym) → OK</code> | <code>Object.keys(sym) → OK</code> |

7.11.4 Well-known symbols

The global object `Symbol` has several properties that serve as constants for so-called *well-known symbols*. These symbols let you configure how ES6 treats an object, by using them as property keys. This is a list of [all well-known symbols](#):

- Customizing basic language operations ([explained in Chap. “New OOP features besides classes”](#)):
 - `Symbol.hasInstance` (method)
Lets an object `C` customize the behavior of `x instanceof C`.
 - `Symbol.toPrimitive` (method)
Lets an object customize how it is converted to a primitive value. This is the first step whenever something is coerced to a primitive type (via operators etc.).
 - `Symbol.toStringTag` (string)
Called by `Object.prototype.toString()` to compute the default string description of an object `obj`: `'[object '+obj[Symbol.toStringTag]+'']`.
 - `Symbol.unscopables` (Object)
Lets an object hide some properties from the `with` statement.
- Iteration ([explained in the chapter on iteration](#)):
 - `Symbol.iterator` (method)
A method with this key makes an object *iterable* (its elements can be iterated over by language constructs such as the `for-of` loop and the spread operator `(...)`); it returns an *iterator*. Details: chapter “[Iterables and iterators](#)”.
- Forwarding calls from string methods: The following string methods are forwarded to methods of their parameters (usually regular expressions).
 - `Symbol.match` is used by `String.prototype.match`.
 - `Symbol.replace` is used by `String.prototype.replace`.
 - `Symbol.search` is used by `String.prototype.search`.
 - `Symbol.split` is used by `String.prototype.split`.

The details are explained in Sect. “[String methods that delegate regular expression work to their parameters](#)” in the chapter on strings.
- Miscellaneous:
 - `Symbol.species` (method)
Configures how built-in methods (such as `Array.prototype.map()`) create objects that are similar to this. The details are explained in [the chapter on classes](#).
 - `Symbol.isConcatSpreadable` (boolean)
Configures whether `Array.prototype.concat()` adds the indexed elements of an object to its result (“spreading”) or the object as a single element ([details are explained in the chapter on Arrays](#)).

7.11.5 Global symbol registry

If you want a symbol to be the same in all realms, you need to create it via the global symbol registry. The following method lets you do that:

- `Symbol.for(str) : symbol`
Returns the symbol whose key is the string `str` in the registry. If `str` isn't in the registry yet, a new symbol is created and filed in the registry under the key `str`.

Another method lets you make the reverse look up and found out under which key a string is stored in the registry. This is may be useful for serializing symbols.

- `Symbol.keyFor(sym) : string`
returns the string that is associated with the symbol `sym` in the registry. If `sym` isn't in the registry, this method returns `undefined`.

8. Template literals

8.1 Overview

ES6 has two new kinds of literals: *template literals* and *tagged template literals*. These two literals have similar names and look similar, but they are quite different. It is therefore important to distinguish:

- Web templates (data): HTML with blanks to be filled in
- Template literals (code): multi-line string literals plus interpolation
- Tagged template literals (code): function calls

Template literals are string literals that can stretch across multiple lines and include interpolated expressions:

```
const firstName = 'Jane';
console.log(`Hello ${firstName}!
How are you
today?`);

// Output:
// Hello Jane!
// How are you
// today?
```

Tagged template literals (short: *tagged templates*) are created by mentioning a function before a template literal:

```
> String.raw`A \tagged\ template`
'A \tagged\ template'
```

Tagged templates are function calls. In the previous example, the method `String.raw` is called to produce the result of the tagged template.

8.2 Introduction

Literals are syntactic constructs that produce values. Examples include string literals (which produce strings) and regular expression literals (which produce regular expression objects). ECMAScript 6 has two new literals:

- *Template literals* are string literals with support for interpolation and multiple lines.
- *Tagged template literals* (short: *tagged templates*): are function calls whose parameters are provided via template literals.

It is important to keep in mind that the names of template literals and tagged templates are slightly misleading. They have nothing to do with *templates*, as they are often used in web development: text files with blanks that can be filled in via (e.g.) JSON data.

8.2.1 Template literals

A template literal is a new kind of string literal that can span multiple lines and *interpolate* expressions (include their results). For example:

```
const firstName = 'Jane';
console.log(`Hello ${firstName}!
How are you
today?`);

// Output:
// Hello Jane!
// How are you
// today?
```

The literal itself is delimited by backticks (```), the interpolated expressions inside the literal are delimited by `${` and `}`. Template literals always produce strings.

8.2.2 Line terminators in template literals are always LF (`\n`)

Common ways of terminating lines are:

- Line feed (LF, `\n`, U+000A): used by Unix (incl. OS X)
- Carriage return (CR, `\r`, U+000D): used by the old Mac OS.
- CRLF (`\r\n`): used by Windows.

All of these line terminators are normalized to LF in template literals. That is, the following code logs `true` on all platforms:

```
const str = `BEFORE
AFTER`;
console.log(str === 'BEFORE\nAFTER'); // true
```



Spec: line terminators in template literals

In the ECMAScript specification, Sect. “[Static Semantics: TV and TRV](#)” defines how line terminators are to be interpreted in template literals:

- The TRV of LineTerminatorSequence :: <LF> is the code unit value 0x000A.
- The TRV of LineTerminatorSequence :: <CR> is the code unit value 0x000A.
- The TRV of LineTerminatorSequence :: <CR><LF> is the sequence consisting of the code unit value 0x000A.

8.2.3 Tagged template literals

The following is a *tagged template literal* (short: *tagged template*):

```
tagFunction`Hello ${firstName} ${lastName}!`
```

Putting a template literal after an expression triggers a function call, similar to how a parameter list (comma-separated values in parentheses) triggers a function call. The previous code is equivalent to the following function call (in reality, the function gets more information, but that is explained later).

```
tagFunction(['Hello ', ' ', '!'], firstName, lastName)
```

Thus, the name before the content in backticks is the name of a function to call, the *tag function*. The tag function receives two different kinds of data:

- *Template strings* such as 'Hello '.
- *Substitutions* such as `firstName` (delimited by `${}`). A substitution can be any expression.

Template strings are known statically (at compile time), substitutions are only known at runtime. The tag function can do with its parameters as it pleases: It can completely ignore the template strings, return values of any type, etc.

Additionally, tag functions get two versions of each template string:

- A “raw” version in which backslashes are not interpreted (``\n`` becomes `'\\n'`, a string of length 2)
- A “cooked” version in which backslashes are special (``\n`` becomes a string with just a newline in it).

That allows `String.raw` to do its work:

```
> String.raw`\n` === '\\n'
true
```

8.3 Examples of using tagged template literals

Tagged template literals allow you to implement custom embedded sub-languages (which are sometimes called *domain-specific languages*) with little effort, because JavaScript does much of the parsing for you. You only have to write a function that receives the results.

Let’s look at examples. Some of them are inspired by [the original proposal](#) for template literals, which refers to them via their old name, *quasi-literals*.

8.3.1 Raw strings

ES6 includes the tag function `String.raw` for *raw strings*, where backslashes have no special meaning:

```
const str = String.raw`This is a text
with multiple lines.
Escapes are not interpreted,
\n is not a newline.`;
```

This is useful whenever you need to create strings that have backslashes in them. For example:

```
function createNumberRegExp(english) {
  const PERIOD = english ? String.raw`\.`; // (A)
  return new RegExp(`[0-9]+(${PERIOD}[0-9]+)?`);
}
```

In line A, `String.raw` enables us to write the backslash as we would in a regular expression literal. With normal string literals, we have to escape twice: First, we need to escape the dot for the regular expression. Second, we need to escape the backslash for the string literal.

8.3.2 Shell commands

```
const proc = sh`ps ax | grep ${pid}`;
```

(Source: [David Herman](#))

8.3.3 Byte strings

```
const buffer = bytes`455336465457210a`;
```

(Source: [David Herman](#))

8.3.4 HTTP requests

```
POST`http://foo.org/bar?a=${a}&b=${b}`
Content-Type: application/json
X-Credentials: ${credentials}

{ "foo": ${foo},
  "bar": ${bar}}
,
(myOnReadyStateChangeHandler);
```

(Source: [Luke Hoban](#))

8.3.5 More powerful regular expressions

Steven Levithan has given [an example](#) of how tagged template literals could be used for his regular expression library [XRegExp](#).



XRegExp is highly recommended if you are working with regular expressions. You get many advanced features, but there is only a small performance penalty – once at creation time – because XRegExp compiles its input to native regular expressions.

Without tagged templates, you write code such as the following:

```
var parts = '/2015/10/Page.html'.match(XRegExp (
  '^ # match at start of string only \n' +
  '/ (?<year> [^/]+ ) # capture top dir name as year \n' +
  '/ (?<month> [^/]+ ) # capture subdir name as month \n' +
  '/ (?<title> [^/]+ ) # capture base name as title \n' +
  '\\.html? $ # .htm or .html file ext at end of path ', 'x'
));

console.log(parts.year); // 2015
```

We can see that XRegExp gives us named groups (`year`, `month`, `title`) and the `x` flag. With that flag, most whitespace is ignored and comments can be inserted.

There are two reasons that string literals don't work well here. First, we have to type every regular expression backslash twice, to escape it for the string literal. Second, it is cumbersome to enter multiple lines: Instead of adding strings, you could also end the line with a backslash. But that is brittle and you still have to explicitly add newlines via `\n`. These two problems go away with tagged templates:

```
var parts = '/2015/10/Page.html'.match(XRegExp.rx`
  ^ # match at start of string only
  / (?<year> [^/]+ ) # capture top dir name as year
  / (?<month> [^/]+ ) # capture subdir name as month
  / (?<title> [^/]+ ) # capture base name as title
  \.html? $ # .htm or .html file ext at end of path
`);
```

Tagged templates also let you insert values `v` via `${v}`. I'd expect a regular expression library to escape strings and to insert regular expressions verbatim. For example:

```
var str = 'really?';
var regex = XRegExp.rx`${str}`;
```

This would be equivalent to

```
var regex = XRegExp.rx`(really\?)*`;
```

8.3.6 Query languages

Example:

```
$`a.${className}[href*="//${domain}/"]`
```

This is a DOM query that looks for all `<a>` tags whose CSS class is `className` and whose target is a URL with the given domain. The tag function `$` ensures that the arguments are correctly escaped, making this approach safer than manual string concatenation.

8.3.7 React JSX via tagged templates

[Facebook React](#) is “a JavaScript library for building user interfaces”. It has the optional language extension JSX that enables you to build *virtual* DOM trees for user interfaces.

This extension makes your code more concise, but it is also non-standard and breaks compatibility with the rest of the JavaScript ecosystem.

The library `t7.js` provides an alternative to JSX and uses templates tagged with `t7`:

```
t7.module(function(t7) {
  function MyWidget(props) {
    return t7`
      <div>
        <span>I'm a widget ${ props.welcome }</span>
      </div>
    `;
  }

  t7.assign('Widget', MyWidget);

  t7`
    <div>
      <header>
        <Widget welcome="Hello world" />
      </header>
    </div>
  `;
});
```

In “[Why not Template Literals?](#)”, the React team explains why they opted not to use template literals. One challenge is accessing components inside tagged templates. For example, `MyWidget` is accessed from the second tagged template in the previous example. One verbose way of doing so would be:

```
<${MyWidget} welcome="Hello world" />
```

Instead, `t7.js` uses a registry which is filled via `t7.assign()`. That requires extra configuration, but the template literals look nicer, especially if there is both an opening and a closing tag.

8.3.8 Facebook GraphQL

[Facebook Relay](#) is a “JavaScript framework for building data-driven React applications”. One of its parts is the query language GraphQL whose queries can be created via templates tagged with `Relay.QL`. For example ([borrowed from the Relay homepage](#)):

```
class TeaStore extends React.Component {
  render() {
    return <ul>
      {this.props.store.teas.map(
        tea => <Tea tea={tea} />
      )}
    </ul>;
  }
}
TeaStore = Relay.createContainer(TeaStore, {
  fragments: { // (A)
    store: () => Relay.QL`
      fragment on Store {
        teas { ${Tea.getFragment('tea')} },
      }
    `,
  },
});
```

The query is created in method `fragments` (line A) and attached to the React component `TeaStore`. Its results of the query are put into `this.props.store`.

This is the data that the query operates on:

```
const STORE = {
  teas: [
    {name: 'Earl Grey Blue Star', steepingTime: 5},
    ...
  ],
};
```

8.3.9 Text localization (L10N)

This section describes a simple approach to text localization that supports different languages and different locales (how to format numbers, time, etc.). Given the following message.

```
alert(msg`Welcome to ${siteName}, you are visitor
  number ${visitorNumber}:d!`);
```

The tag function `msg` would work as follows.

First, The literal parts are concatenated to form a string that can be used to look up a translation in a table. An example for a lookup string is:

```
'Welcome to {0}, you are visitor number {1}!'
```

An example for a translation to German is:

```
'Besucher Nr. {1}, willkommen bei {0}!'
```

The English “translation” would be the same as the lookup string.

Second, the result from the lookup is used to display the substitutions. Because a lookup result includes indices, it can rearrange the order of the substitutions. That has been done in German, where the visitor number comes before the site name. How the substitutions are formatted can be influenced via annotations such as `:d`. This annotation means that a locale-specific decimal separator should be used for `visitorNumber`. Thus, a possible English result is:

Welcome to ACME Corp., you are visitor number 1,300!

In German, we have results such as:

Besucher Nr. 1.300, willkommen bei ACME Corp.!

8.3.10 Text templating via untagged template literals

Let's say we want to create HTML that display the following data in a table:

```
const data = [
  { first: '<Jane>', last: 'Bond' },
  { first: 'Lars', last: '<Croft>' },
];
```

As explained previously, template literals are not templates. They are code that is executed immediately, not text with holes that you can apply to data. The latter description gives us a hint how we can turn a template literal into an actual template – it sounds like the description of a function (data in, text out). Let's implement a template `tmpl` as a function from data to strings:

```
const tmpl = addr => `
  <table>
    ${addr.map(addr => `
      <tr><td>${addr.first}</td></tr>
      <tr><td>${addr.last}</td></tr>
    `).join('')}
  </table>
`;
console.log(tmpl(data));
// Output:
// <table>
//
//   <tr><td><Jane></td></tr>
//   <tr><td>Bond</td></tr>
//
//   <tr><td>Lars</td></tr>
//   <tr><td><Croft></td></tr>
//
// </table>
```

The outer template literal provides the bracketing `<table>` and `</table>`. Inside, we are embedding JavaScript code that produces a string by joining an Array of strings. The Array is created by mapping each address to two table rows. Note that the plain text pieces `<Jane>` and `<Croft>` are not properly escaped. How to do that via a tagged template is explained in the next section.

8.3.10.1 Should I use this technique in production code?

This is a useful quick solution for smaller templating tasks. For larger task, you may want more powerful solutions such as the templating engine [Handlebars.js](#) or the JSX syntax used in React.

Acknowledgement: This approach to text templating is based on [an idea](#) by Claus Reinke.

8.3.11 A tag function for HTML templating

Compared to using untagged templates for HTML templating, like we did in the previous section, tagged templates bring two advantages:

- They can escape characters for us if we prefix `${}` with a dollar sign. That is needed for the names, which contain characters that need to be escaped (`<Jane>`).
- They can automatically `join()` Arrays for us, so that we don't have to call that method ourselves.

Then the code for the template looks as follows. The name of the tag function is `html`:

```
const tmpl = addr => html`
  <table>
    ${addr.map(addr => html`
      <tr><td>${addr.first}</td></tr>
      <tr><td>${addr.last}</td></tr>
    `)}
  </table>
`;
const data = [
  { first: '<Jane>', last: 'Bond' },
  { first: 'Lars', last: '<Croft>' },
];
console.log(tmpl(data));
// Output:
// <table>
//
//   <tr><td>&lt;Jane&gt;</td></tr>
```

```
//      <tr><td>Bond</td></tr>
//
//      <tr><td>Lars</td></tr>
//      <tr><td>&lt;Croft&gt;</td></tr>
//
// </table>
```

Note that the angle brackets around `Jane` and `Croft` are escaped, whereas those around `tr` and `td` aren't.

The syntax ``${}`` is used for text that should be HTML-escaped. It is not in any way special; it's just the normal text ``` followed by the substitution ``{}``. Therefore, the tag function has to check the text preceding a substitution in order to determine whether to escape or not.

An implementation of `html` is shown later.

8.4 Implementing tag functions

The following is a tagged template literal:

```
tagFunction`lit1\n${subst1} lit2 ${subst2}`
```

This is a simplified version of the function call triggered by this literal:

```
tagFunction(['lit1\n', ' lit2 ', ''], subst1, subst2)
```

The exact function call looks more like this:

```
// Globally: add template object to per-realm template map
{
  // "Cooked" template strings: backslash is interpreted
  const templateObject = ['lit1\n', ' lit2 ', ''];
  // "Raw" template strings: backslash is verbatim
  templateObject.raw = ['lit1\n', ' lit2 ', ''];

  // The Arrays with template strings are frozen
  Object.freeze(templateObject.raw);
  Object.freeze(templateObject);

  __templateMap__[716] = templateObject;
}

// In-place: invocation of tag function
tagFunction(__templateMap__[716], subst1, subst2)
```

There are two kinds of input that the tag function receives:

1. Template strings: are delivered via the template object in the first parameter. They are the static parts that don't change (e.g. `' lit2 '`). You get both "cooked" template strings (with escapes such as `\n` interpreted) and "raw" template strings (with uninterpreted escapes).
2. Substitutions: are delivered via trailing parameters. They are embedded inside template literals via ``{}`` (e.g. `subst1`). Substitutions are dynamic, they can change with each invocation.

The number of template strings is always one plus the number of substitutions. If a substitution is first in a literal, it is prefixed by an empty template string. If a substitution is last, it is suffixed by an empty template string (as in the previous example).

The idea behind the template object is that the same tagged template might be executed multiple times (e.g. in a loop or a function). The template object enables the tag function to cache data from previous invocations: It can put data it derived from input source #1 into the object, to avoid recomputing it needlessly. Caching happens per *realm* (think frame in a browser). That is, there is one template object per call site and realm.

The following is a first example of a tag function, a reimplement of `String.raw`:

```
function raw(strs, ...substs) {
  let result = strs.raw[0];
  for (let [i, subst] of substs.entries()) {
    result += subst;
    result += strs.raw[i+1];
  }
  return result;
}
```



Tagged template literals in the spec

A section on [tagged template literals](#) explains how they are interpreted as function calls. A separate section explains how a template literal is turned into a list of arguments: the template object and the substitutions.

8.4.1 Escaping in tagged template literals: cooked versus raw

In tagged template literals, there are more rules for escaping, because *template strings* (the text fragments inside the backticks, excluding substitutions) are available in two

interpretations: cooked and raw. The rules are:

- In both cooked and raw interpretation, a backslash (\) in front of a dollar sign (\$) prevents \${ from being interpreted as starting a substitution.
- However, every single backslash is mentioned in the raw interpretation, even the ones that escape substitutions.

The tag function `describe` allows us to explore what that means.

```
function describe(tmplObj, ...subs) {
  console.log('Cooked:', intersperse(tmplObj, subs));
  console.log('Raw:   ', intersperse(tmplObj.raw, subs));
}

function intersperse(tmplStrs, subs) {
  // There is always at least one element in tmplStrs
  let result = tmplStrs[0];
  subs.forEach((subst, i) => {
    result += String(subst);
    result += tmplStrs[i+1];
  });
  return result;
}
```

Let's use this tag function (I am not showing the result `undefined` of these function calls):

```
> describe`${3+3}`
Cooked: 6
Raw:    6

> describe`${3+3}`
Cooked: ${3+3}
Raw:    `${3+3}`

> describe`${3+3}`
Cooked: \6
Raw:    `${3+3}`

> describe`${3+3}`
Cooked: \${3+3}
Raw:    `${3+3}`
```

As you can see, whenever the cooked interpretation has a substitution then so does the raw interpretation. However, all backslashes from the literal appear in the raw interpretation; if a backslash precedes the characters `${` then it prevented a substitution.

Other occurrences of the backslash are interpreted similarly:

- In cooked mode, the backslash is handled like in string literals.
- In raw mode, the backslash is used verbatim.

For example:

```
> `\\n`
'\\n'
> String.raw`\\n`
'\\n'
```

The only time the backslash ever has an effect in raw mode is when it appears in front of a substitution (which it escapes).



Escaping in tagged template literals in the spec

In [the grammar for template literals](#), you can see that, within a template literal, there must be no open curly brace (}) after a dollar sign (\$). However, an escaped dollar sign (\\$) can be followed by an open curly brace. The rules for interpreting the characters of a template literal are explained in [a separate section](#).

8.4.2 Example: implementing a tag function for HTML templating

I previously demonstrated the tag function `html` for HTML templating:

```
const tmpl = addr => html`
  <table>
    ${addr.map(addr => html`
      <tr><td>${addr.first}</td></tr>
      <tr><td>${addr.last}</td></tr>
    `)}
  </table>
`;

const data = [
  { first: '<Jane>', last: 'Bond' },
  { first: 'Lars', last: '<Croft>' },
];

console.log(tmpl(data));
// Output:
// <table>
//   <tr><td>&lt;Jane&gt;</td></tr>
//   <tr><td>Bond</td></tr>
//   <tr><td>Lars</td></tr>
//   <tr><td>&lt;Croft&gt;</td></tr>
```

```
//
// </table>
```

The syntax ``${}`` is used for text that should be HTML-escaped. It is not in any way special; it's just the normal text ``` followed by the substitution `{}``. Therefore, the tag function has to check the text preceding a substitution in order to determine whether to escape or not.

This is an implementation of `html`:

```
function html(templateObject, ...substs) {
  // Use raw template strings: we don't want
  // backslashes (\n etc.) to be interpreted
  const raw = templateObject.raw;

  let result = '';

  substs.forEach((subst, i) => {
    // Retrieve the template string preceding
    // the current substitution
    let lit = raw[i];

    // In the example, map() returns an Array:
    // If substitution is an Array (and not a string),
    // we turn it into a string
    if (Array.isArray(subst)) {
      subst = subst.join(' ');
    }

    // If the substitution is preceded by a dollar sign,
    // we escape special characters in it
    if (lit.endsWith('$')) {
      subst = htmlEscape(subst);
      lit = lit.slice(0, -1);
    }

    result += lit;
    result += subst;
  });
  // Take care of last template string
  // (Never fails, because an empty tagged template
  // produces one template string, an empty string)
  result += raw[raw.length-1]; // (A)

  return result;
}
```

Each substitution is always surrounded by template strings. If the tagged template literal ends with a substitution, the last template string is an empty string. Accordingly, the following expression is always true:

```
templateObject.length === substs.length + 1
```

That's why we need to append the last template string in line A.

The following is a simple implementation of `htmlEscape()`.

```
function htmlEscape(str) {
  return str.replace(/&/g, '&amp;') // first!
    .replace(/>/g, '&gt;')
    .replace(/</g, '&lt;')
    .replace(/"/g, '&quot;')
    .replace(/'/g, '&#39;')
    .replace(/`/g, '&#96;');
}
```

8.4.2.1 More ideas

There are more things you can do with this approach to templating:

- This approach isn't limited to HTML, it would work just as well for other kinds of text. Obviously, escaping would have to be adapted.
- if-then-else inside the template can be done via the ternary operator (`cond? then:else`) or via the logical Or operator (`||`):

```
`${addr.first ? addr.first : '(No first name)'}`
`${addr.first || '(No first name)'}`
```

- Some of the leading whitespace in each line can be trimmed if the first non-whitespace character in the first line defines where the first column is.
- Destructuring can be used:

```
`${addrs.map(({first,last}) => html`
  <tr><td>${first}</td></tr>
  <tr><td>${last}</td></tr>
`)}
`)
```

8.4.3 Example: assembling regular expressions

There are two ways of creating regular expression instances.

- Statically (at compile time), via a regular expression literal: `/^abc$/i`
- Dynamically (at runtime), via the `RegExp` constructor: `new RegExp('^abc$', 'i')`

If you use the latter, it is because you have to wait until runtime so that all necessary ingredients are available. You are creating the regular expression by concatenating three kinds of pieces:

1. Static text
2. Dynamic regular expressions
3. Dynamic text

For #3, special characters (dots, square brackets, etc.) have to be escaped, while #1 and #2 can be used verbatim. A regular expression tag function `regex` can help with this task:

```
const INTEGER = /\d+/;
const decimalPoint = '.'; // locale-specific! E.g., ',' in Germany
const NUMBER = regex`${INTEGER} (${decimalPoint}${INTEGER})?`;
```

`regex` looks like this:

```
function regex(tmplObj, ...substs) {
  // Static text: verbatim
  let regexText = tmplObj.raw[0];
  for ([i, subst] of substs.entries()) {
    if (subst instanceof RegExp) {
      // Dynamic regular expressions: verbatim
      regexText += String(subst);
    } else {
      // Other dynamic data: escaped
      regexText += quoteText(String(subst));
    }
    // Static text: verbatim
    regexText += tmplObj.raw[i+1];
  }
  return new RegExp(regexText);
}
function quoteText(text) {
  return text.replace(/[\^$. *+?() [\] {} !<>:-]/g, '\\$&');
}
```

8.5 FAQ: template literals and tagged template literals

8.5.1 Where do template literals and tagged template literals come from?

Template literals and tagged template literals were borrowed from the language E, which calls this feature *quasi literals*.

8.5.2 What is the difference between macros and tagged template literals?

Macros allow you to implement constructs that have custom syntax. Providing macros for a language whose syntax is as complex as JavaScript's is difficult and ongoing research (see Mozilla's [sweet.js](#)).

While macros are much more powerful for implementing sub-languages than tagged templates, they depend on the tokenization of the language. Therefore, tagged templates are complementary, because they specialize on text content.

8.5.3 Can I load a template literal from an external source?

What if I want to load a template literal such as ``Hello ${name}!`` from an external source (e.g., a file)?

Note that you are abusing this mechanism if you do so. Given that a template literal can contain arbitrary expressions and is a literal, loading it from somewhere else is similar to loading an expression or a string literal – you have to use `eval()` or something similar.

Coming back to the example, this is how you'd do it:

```
const str = `Hello ${name}!`; // external source
const func = new Function('name', 'return ' + str);
const name = 'Jane';
const result = func(name);
```

Every variable that isn't declared inside the template literal has to become a parameter of the function `func` that we are creating. Alternatively, you could load a whole function and eval it⁷:

```
const str = '(name) => `Hello ${name}!`; // external source
const func = eval.call(null, str); // indirect eval
const name = 'Jane';
const result = func(name);
```

8.5.4 Why are backticks the delimiters for template literals?

The backtick was one of the few ASCII characters that were still unused. The syntax ``{}`` for interpolation is the de-facto standard (Unix shells, etc.).

8.5.5 Weren't template literals once called template strings?

The template literal terminology changed relatively late during the creation of the ES6 spec. The following are the old terms:

- Template string (literal): the old name for *template literal*.
- Tagged template string (literal): the old name for *tagged template literal*.
- Template handler: the old name for *tag function*.
- Literal section: the old name for *template string* (the term *substitution* remains the same).

9. Variables and scoping

9.1 Overview

ES6 provides two new ways of declaring variables: `let` and `const`, which mostly replace the ES5 way of declaring variables, `var`.

9.1.1 `let`

`let` works similarly to `var`, but the variable it declares is *block-scoped*, it only exists within the current block. `var` is *function-scoped*.

In the following code, you can see that the `let`-declared variable `tmp` only exists with the block that starts in line A:

```
function order(x, y) {
  if (x > y) { // (A)
    let tmp = x;
    x = y;
    y = tmp;
  }
  console.log(tmp===x); // ReferenceError: tmp is not defined
  return [x, y];
}
```

9.1.2 `const`

`const` works like `let`, but the variable you declare must be immediately initialized, with a value that can't be changed afterwards.

```
const foo;
// SyntaxError: missing = in const declaration

const bar = 123;
bar = 456;
// TypeError: `bar` is read-only
```

Since `for-of` creates one *binding* (storage space for a variable) per loop iteration, it is OK to `const`-declare the loop variable:

```
for (const x of ['a', 'b']) {
  console.log(x);
}
// Output:
// a
// b
```

9.1.3 Ways of declaring variables

The following table gives an overview of six ways in which variables can be declared in ES6 (inspired by [a table by kangax](#)):

| | Hoisting | Scope | Creates global properties |
|-----------------------|--------------------|---------------|---------------------------|
| <code>var</code> | Declaration | Function | Yes |
| <code>let</code> | Temporal dead zone | Block | No |
| <code>const</code> | Temporal dead zone | Block | No |
| <code>function</code> | Complete | Block | Yes |
| <code>class</code> | No | Block | No |
| <code>import</code> | Complete | Module-global | No |

9.2 Block scoping via `let` and `const`

Both `let` and `const` create variables that are *block-scoped* – they only exist within the innermost block that surrounds them. The following code demonstrates that the `const`-declared variable `tmp` only exists inside the then-block of the `if` statement:

```
function func() {
  if (true) {
    const tmp = 123;
  }
  console.log(tmp); // ReferenceError: tmp is not defined
}
```

In contrast, `var`-declared variables are function-scoped:

```
function func() {
  if (true) {
    var tmp = 123;
  }
  console.log(tmp); // 123
}
```

Block scoping means that you can shadow variables within a function:

```
function func() {
  const foo = 5;
  if (...) {
    const foo = 10; // shadows outer `foo`
    console.log(foo); // 10
  }
  console.log(foo); // 5
}
```

9.3 `const` creates immutable variables

Variables created by `let` are mutable:

```
let foo = 'abc';
foo = 'def';
console.log(foo); // def
```

Constants, variables created by `const`, are immutable – you can't assign them a different value:

```
const foo = 'abc';
foo = 'def'; // TypeError
```



Spec detail: changing a `const` variable always throws a `TypeError`

Normally, changing an immutable binding only causes an exception in strict mode, as per [SetMutableBinding\(\)](#). But `const`-declared variables always produce strict bindings – see [FunctionDeclarationInstantiation\(func, argumentsList\)](#), step 35.b.i.1.

9.3.1 Pitfall: `const` does not make the value immutable

`const` only means that a variable always has the same value, but it does not mean that the value itself is or becomes immutable. For example, `obj` is a constant, but the value it points to is mutable – we can add a property to it:

```
const obj = {};
obj.prop = 123;
console.log(obj.prop); // 123
```

We cannot, however assign a different value to `obj`:

```
obj = {}; // TypeError
```

If you want the value of `obj` to be immutable, you have to take care of it, yourself, e.g. by [freezing it](#):

```
const obj = Object.freeze({});
obj.prop = 123; // TypeError
```

9.3.1.1 Pitfall: `Object.freeze()` is shallow

Keep in mind that `Object.freeze()` is *shallow*; it only freezes the properties of its argument, not the objects stored in its properties. For example, the object `obj` is frozen:

```
> const obj = Object.freeze({ foo: {} });
> obj.bar = 123
TypeError: Can't add property bar, object is not extensible
> obj.foo = {}
TypeError: Cannot assign to read only property 'foo' of #<Object>
```

But the object `obj.foo` is not.

```
> obj.foo.qux = 'abc';
> obj.foo.qux
'abc'
```

9.3.2 `const` in loop bodies

Once a `const` variable has been created, it can't be changed. But that doesn't mean that you can't re-enter its scope and start fresh, with a new value. For example, via a loop:

```
function logArgs(...args) {
  for (const [index, elem] of args.entries()) {
    const message = index + '. ' + elem;
    console.log(message);
  }
}
```

```

    }
  }
  logArgs('Hello', 'everyone');

// Output:
// 0. Hello
// 1. everyone

```

9.4 The temporal dead zone

A variable declared by `let` or `const` has a so-called *temporal dead zone* (TDZ): When entering its scope, it can't be accessed (got or set) until execution reaches the declaration. Let's compare the life cycles of `var`-declared variables (which don't have TDZs) and `let`-declared variables (which have TDZs).

9.4.1 The life cycle of `var`-declared variables

`var` variables don't have temporal dead zones. Their life cycle comprises the following steps:

- When the scope (its surrounding function) of a `var` variable is entered, storage space (a *binding*) is created for it. The variable is immediately initialized, by setting it to `undefined`.
- When the execution within the scope reaches the declaration, the variable is set to the value specified by the *initializer* (an assignment) – if there is one. If there isn't, the value of the variable remains `undefined`.

9.4.2 The life cycle of `let`-declared variables

Variables declared via `let` have temporal dead zones and their life cycles look like this:

- When the scope (its surrounding block) of a `let` variable is entered, storage space (a *binding*) is created for it. The variable remains uninitialized.
- Getting or setting an uninitialized variable causes a `ReferenceError`.
- When the execution within the scope reaches the declaration, the variable is set to the value specified by the *initializer* (an assignment) – if there is one. If there isn't then the value of the variable is set to `undefined`.

`const` variables work similarly to `let` variables, but they must have an initializer (i.e., be set to a value immediately) and can't be changed.

9.4.3 Examples

Within a TDZ, an exception is thrown if a variable is got or set:

```

let tmp = true;
if (true) { // enter new scope, TDZ starts
  // Uninitialized binding for `tmp` is created
  console.log(tmp); // ReferenceError

  let tmp; // TDZ ends, `tmp` is initialized with `undefined`
  console.log(tmp); // undefined

  tmp = 123;
  console.log(tmp); // 123
}
console.log(tmp); // true

```

If there is an initializer then the TDZ ends *after* the assignment was made:

```

let foo = console.log(foo); // ReferenceError

```

The following code demonstrates that the dead zone is really *temporal* (based on time) and not spatial (based on location):

```

if (true) { // enter new scope, TDZ starts
  const func = function () {
    console.log(myVar); // OK!
  };

  // Here we are within the TDZ and
  // accessing `myVar` would cause a `ReferenceError`

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

```

9.4.4 `typeof` throws a `ReferenceError` for a variable in the TDZ

If you access a variable in the temporal dead zone via `typeof`, you get an exception:

```

if (true) {
  console.log(typeof foo); // ReferenceError (TDZ)
  console.log(typeof aVariableThatDoesntExist); // 'undefined'
  let foo;
}

```

Why? The rationale is as follows: `foo` is not undeclared, it is uninitialized. You should be aware of its existence, but aren't. Therefore, being warned seems desirable.

Furthermore, this kind of check is only useful for conditionally creating global variables. That's something that only advanced JavaScript programmers should do and it can only be achieved via `var`.

There is a way to check whether a global variable exists that does not involve `typeof`:

```
// With `typeof`
if (typeof someGlobal === 'undefined') {
  var someGlobal = { ... };
}

// Without `typeof`
if (!('someGlobal' in window)) {
  window.someGlobal = { ... };
}
```

The former way of creating a global variable only works in global scope (and therefore not inside ES6 modules).

9.4.5 Why is there a temporal dead zone?

- To catch programming errors: Being able to access a variable before its declaration is strange. If you do so, it is normally by accident and you should be warned about it.
- For `const`: Making `const` work properly is difficult. [Quoting Allen Wirfs-Brock](#): “TDZs ... provide a rational semantics for `const`. There was significant technical discussion of that topic and TDZs emerged as the best solution.” `let` also has a temporal dead zone so that switching between `let` and `const` doesn't change behavior in unexpected ways.
- Future-proofing for guards: JavaScript may eventually have *guards*, a mechanism for enforcing at runtime that a variable has the correct value (think runtime type check). If the value of a variable is `undefined` before its declaration then that value may be in conflict with the guarantee given by its guard.

9.4.6 Further reading

Sources of this section:

- [“Performance concern with `let/const`”](#)
- [“Bug 3009 – `typeof` on TDZ variable”](#)

9.5 `let` and `const` in loop heads

The following loops allow you to declare variables in their heads:

- `for`
- `for-in`
- `for-of`

To make a declaration, you can use either `var`, `let` or `const`. Each of them has a different effect, as I'll explain next.

9.5.1 `for` loop

`var`-declaring a variable in the head of a `for` loop creates a single *binding* (storage space) for that variable:

```
const arr = [];
for (var i=0; i < 3; i++) {
  arr.push(() => i);
}
arr.map(x => x()); // [3,3,3]
```

Every `i` in the bodies of the three arrow functions refers to the same binding, which is why they all return the same value.

If you `let`-declare a variable, a new binding is created for each loop iteration:

```
const arr = [];
for (let i=0; i < 3; i++) {
  arr.push(() => i);
}
arr.map(x => x()); // [0,1,2]
```

This time, each `i` refers to the binding of one specific iteration and preserves the value that was current at that time. Therefore, each arrow function returns a different value.

`const` works like `var`, but you can't change the initial value of a `const`-declared variable:

```
// TypeError: Assignment to constant variable
// (due to i++)
for (const i=0; i<3; i++) {
  console.log(i);
}
```

Getting a fresh binding for each iteration may seem strange at first, but it is very useful whenever you use loops to create functions that refer to loop variables, as explained in [a later section](#).



for loop: per-iteration bindings in the spec

The evaluation of the `for` loop handles `var` as the second case and `let/const` as the third case. Only `let`-declared variables are added to the list `perIterationLets` (step 9), which is passed to `ForBodyEvaluation()` as the second-to-last parameter, `perIterationBindings`.

9.5.2 for-of loop and for-in loop

In a `for-of` loop, `var` creates a single binding:

```
const arr = [];
for (var i of [0, 1, 2]) {
  arr.push(() => i);
}
arr.map(x => x()); // [2,2,2]
```

`let` creates one binding per iteration:

```
const arr = [];
for (let i of [0, 1, 2]) {
  arr.push(() => i);
}
arr.map(x => x()); // [0,1,2]
```

`const` also creates one binding per iteration, but the bindings it creates are immutable.

The `for-in` loop works similarly to the `for-of` loop.



for-of loop: per-iteration bindings in the spec

Per-iteration bindings in `for-of` are handled by `ForIn/OfBodyEvaluation`. In step 5.b, a new environment is created and bindings are added to it via `BindingInstantiation` (mutable for `let`, immutable for `const`). The current iteration value is stored in the variable `nextValue` and used to initialize the bindings in either one of two ways:

- Declaration of single variable (step 5.h.i): is handled via `InitializeReferencedBinding`
- Destructuring (step 5.i.iii): is handled via `one case of BindingInitialization` (`ForDeclaration`), which invokes `another case of BindingInitialization` (`BindingPattern`).

9.5.3 Why are per-iteration bindings useful?

The following is an HTML page that displays three links:

1. If you click on “yes”, it is translated to “ja”.
2. If you click on “no”, it is translated to “nein”.
3. If you click on “perhaps”, it is translated to “vielleicht”.

```
<!doctype html>
<html>
<head>
  <meta charset="UTF-8">
</head>
<body>
  <div id="content"></div>
  <script>
    const entries = [
      ['yes', 'ja'],
      ['no', 'nein'],
      ['perhaps', 'vielleicht'],
    ];
    const content = document.getElementById('content');
    for (let [source, target] of entries) { // (A)
      content.insertAdjacentHTML('beforeend',
        `<div><a id="${source}" href="#">${source}</a></div>`);
      document.getElementById(source).addEventListener(
        'click', (event) => {
          event.preventDefault();
          alert(target); // (B)
        });
    }
  </script>
</body>
</html>
```

What is displayed depends on the variable `target` (line B). If we were to use `var` instead of `let` in line (A), there would be a single binding for the whole loop and `target` would have the value `'vielleicht'`, afterwards. Therefore, no matter what link you click on, you would always get the translation `'vielleicht'`.

Thankfully, with `let`, we get one binding per loop iteration and the translations are displayed correctly.

9.6 Parameters

9.6.1 Parameters versus local variables

If you `let`-declare a variable that has the same name as a parameter, you get a static (load-time) error:

```
function func(arg) {
  let arg; // static error: duplicate declaration of `arg`
}
```

Doing the same inside a block shadows the parameter:

```
function func(arg) {
  {
    let arg; // shadows parameter `arg`
  }
}
```

In contrast, `var`-declaring a variable that has the same name as a parameter does nothing, just like re-declaring a `var` variable within the same scope does nothing.

```
function func(arg) {
  var arg; // does nothing
}

function func(arg) {
  {
    // We are still in same `var` scope as `arg`
    var arg; // does nothing
  }
}
```

9.6.2 Parameter default values and the temporal dead zone

If parameters have default values, they are treated like a sequence of `let` statements and are subject to temporal dead zones:

```
// OK: `y` accesses `x` after it has been declared
function foo(x=1, y=x) {
  return [x, y];
}
foo(); // [1,1]

// Exception: `x` tries to access `y` within TDZ
function bar(x=y, y=2) {
  return [x, y];
}
bar(); // ReferenceError
```

9.6.3 Parameter default values don't see the scope of the body

The scope of parameter default values is separate from the scope of the body (the former surrounds the latter). That means that methods or functions defined "inside" parameter default values don't see the local variables of the body:

```
const foo = 'outer';
function bar(func = x => foo) {
  const foo = 'inner';
  console.log(func()); // outer
}
bar();
```

9.7 The global object

JavaScript's [global object](#) (window in web browsers, global in Node.js) is more a bug than a feature, especially with regard to performance. That's why it makes sense that ES6 introduces a distinction:

- All properties of the global object are global variables. In global scope, the following declarations create such properties:
 - `var` declarations
 - Function declarations
- But there are now also global variables that are not properties of the global object. In global scope, the following declarations create such variables:
 - `let` declarations
 - `const` declarations
 - Class declarations

9.8 Function declarations and class declarations

Function declarations...

- are block-scoped, like `let`.
- create properties on the global object (while in global scope), like `var`.
- are *hoisted*: independently of where a function declaration is mentioned in its scope, it is always created at the beginning of the scope.

The following code demonstrates the hoisting of function declarations:

```
{ // Enter a new scope

  console.log(foo()); // OK, due to hoisting
  function foo() {
```

```

    return 'hello';
  }
}

```

Class declarations...

- are block-scoped.
- don't create properties on the global object.
- are *not* hoisted.

Classes not being hoisted may be surprising, because, under the hood, they create functions. The rationale for this behavior is that the values of their `extends` clauses are defined via expressions and those expressions have to be executed at the appropriate times.

```

{ // Enter a new scope

  const identity = x => x;

  // Here we are in the temporal dead zone of `MyClass`
  const inst = new MyClass(); // ReferenceError

  // Note the expression in the `extends` clause
  class MyClass extends identity(Object) {
  }
}

```

9.9 Coding style: `const` versus `let` versus `var`

I recommend to always use either `let` or `const`:

1. Prefer `const`. You can use it whenever a variable never changes its value. In other words: the variable should never be the left-hand side of an assignment or the operand of `++` or `--`. Changing an object that a `const` variable refers to is allowed:

```

const foo = {};
foo.prop = 123; // OK

```

You can even use `const` in a `for-of` loop, because one (immutable) binding is created per loop iteration:

```

for (const x of ['a', 'b']) {
  console.log(x);
}
// Output:
// a
// b

```

Inside the body of the `for-of` loop, `x` can't be changed.

2. Otherwise, use `let` – when the initial value of a variable changes later on.

```

let counter = 0; // initial value
counter++; // change

let obj = {}; // initial value
obj = { foo: 123 }; // change

```

3. Avoid `var`.

If you follow these rules, `var` will only appear in legacy code, as a signal that careful refactoring is required.

`var` does one thing that `let` and `const` don't: variables declared via it become properties of the global object. However, that's generally not a good thing. You can achieve the same effect by assigning to `window` (in browsers) or `global` (in Node.js).

9.9.1 An alternative approach

An alternative to the just mentioned style rules is to use `const` only for things that are completely immutable (primitive values and frozen objects). Then we have two approaches:

1. **Prefer `const` (recommended):** `const` marks immutable bindings
2. **Prefer `let` (alternative):** `const` marks immutable values

#2 is perfectly acceptable; I only lean slightly in favor of #1.

10. Destructuring

10.1 Overview

Destructuring is a convenient way of extracting values from data stored in (possibly nested) objects and Arrays. It can be used in locations that receive data (such as the left-hand side of an assignment). How to extract the values is specified via patterns (read on for examples).

10.1.1 Object destructuring

Destructuring objects:

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

// {prop} is short for {prop: prop}
const {first, last} = obj;
// first = 'Jane'; last = 'Doe'
```

Destructuring helps with processing return values:

```
const obj = { foo: 123 };

const {writable, configurable} =
  Object.getOwnPropertyDescriptor(obj, 'foo');

console.log(writable, configurable); // true true
```

10.1.2 Array destructuring

Array destructuring (works for all iterable values):

```
const iterable = ['a', 'b'];
const [x, y] = iterable;
// x = 'a'; y = 'b'
```

Destructuring helps with processing return values:

```
const [all, year, month, day] =
  /^(\d\d\d\d)-(\d\d)-(\d\d)$/
  .exec('2999-12-31');
```

10.1.3 Where can destructuring be used?

Destructuring can be used in the following locations:

```
// Variable declarations:
const [x] = ['a'];
let [x] = ['a'];
var [x] = ['a'];

// Assignments:
[x] = ['a'];

// Parameter definitions:
function f([x]) { ... }
f(['a']);
```

You can also destructure in a `for-of` loop:

```
const arr1 = ['a', 'b'];
for (const [index, element] of arr1.entries()) {
  console.log(index, element);
}
// Output:
// 0 a
// 1 b

const arr2 = [
  {name: 'Jane', age: 41},
  {name: 'John', age: 40},
];
for (const {name, age} of arr2) {
  console.log(name, age);
}
// Output:
// Jane 41
// John 40
```

10.2 Background: Constructing data versus extracting data

To fully understand what destructuring is, let's first examine its broader context. JavaScript has operations for constructing data:

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

And it has operations for extracting data:

```
const f = obj.first;
const l = obj.last;
```

Note that we are using the same syntax that we have used for constructing.

There is nicer syntax for constructing – an *object literal*:

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

Destructuring in ECMAScript 6 enables the same syntax for extracting data, where it is called an *object pattern*:

```
const { first: f, last: l } = obj;
```

Just as the object literal lets us create multiple properties at the same time, the object pattern lets us extract multiple properties at the same time.

You can also destructure Arrays via patterns:

```
const [x, y] = ['a', 'b']; // x = 'a'; y = 'b'
```

10.3 Patterns

The following two parties are involved in destructuring:

- **Destructuring source:** the data to be destructured. For example, the right-hand side of a destructuring assignment.
- **Destructuring target:** the pattern used for destructuring. For example, the left-hand side of a destructuring assignment.

The destructuring target is either one of three patterns:

- **Assignment target.** For example: `x`
 - In variable declarations and parameter definitions, only references to variables are allowed. In destructuring assignment, you have more options, as I'll explain later.
- **Object pattern.** For example: `{ first: «pattern», last: «pattern» }`
 - The parts of an object pattern are properties, the property values are again patterns (recursively).
- **Array pattern.** For example: `[«pattern», «pattern»]`
 - The parts of an Array pattern are elements, the elements are again patterns (recursively).

That means that you can nest patterns, arbitrarily deeply:

```
const obj = { a: [{ foo: 123, bar: 'abc' }, {}], b: true };
const { a: [{foo: f}] } = obj; // f = 123
```

10.3.1 Pick what you need

If you destructure an object, you mention only those properties that you are interested in:

```
const { x: x } = { x: 7, y: 3 }; // x = 7
```

If you destructure an Array, you can choose to only extract a prefix:

```
const [x,y] = ['a', 'b', 'c']; // x='a'; y='b';
```

10.4 How do patterns access the innards of values?

In an assignment `pattern = someValue`, how does the `pattern` access what's inside `someValue`?

10.4.1 Object patterns coerce values to objects

The object pattern coerces destructuring sources to objects before accessing properties. That means that it works with primitive values:

```
const {length : len} = 'abc'; // len = 3
const {toString: s} = 123; // s = Number.prototype.toString
```

10.4.1.1 Failing to object-destructure a value

The coercion to object is not performed via `Object()`, but via the internal operation `ToObject()`. `Object()` never fails:

```
> typeof Object('abc')
'object'
> var obj = {};
> Object(obj) === obj
true
> Object(undefined)
{}
> Object(null)
{}
> Object(0)
0
```

`ToObject()` throws a `TypeError` if it encounters `undefined` or `null`. Therefore, the following destructurings fail, even before destructuring accesses any properties:

```
const { prop: x } = undefined; // TypeError
const { prop: y } = null; // TypeError
```

As a consequence, you can use the empty object pattern `{}` to check whether a value is coercible to an object. As we have seen, only `undefined` and `null` aren't:

```
({} = [true, false]); // OK, Arrays are coercible to objects
({} = 'abc'); // OK, strings are coercible to objects

({} = undefined); // TypeError
({} = null); // TypeError
```

The parentheses around the expressions are necessary because statements must not begin with curly braces in JavaScript (details are explained later).

10.4.2 Array patterns work with iterables

Array destructuring uses an iterator to get to the elements of a source. Therefore, you can Array-destructure any value that is iterable. Let's look at examples of iterable values.

Strings are iterable:

```
const [x,...y] = 'abc'; // x='a'; y=['b', 'c']
```

Don't forget that the iterator over strings returns code points ("Unicode characters", 21 bits), not code units ("JavaScript characters", 16 bits). (For more information on Unicode, consult the chapter "[Chapter 24. Unicode and JavaScript](#)" in "Speaking JavaScript".) For example:

```
const [x,y,z] = 'a\uD83D\uDC9c'; // x='a'; y='\uD83D\uDC9c'; z='c'
```

You can't access the elements of a Set via indices, but you can do so via an iterator. Therefore, Array destructuring works for Sets:

```
const [x,y] = new Set(['a', 'b']); // x='a'; y='b';
```

The Set iterator always returns elements in the order in which they were inserted, which is why the result of the previous destructuring is always the same.

Infinite sequences. Destructuring also works for iterators over infinite sequences. The generator function `allNaturalNumbers()` returns an iterator that yields 0, 1, 2, etc.

```
function* allNaturalNumbers() {
  for (let n = 0; ; n++) {
    yield n;
  }
}
```

The following destructuring extracts the first three elements of that infinite sequence.

```
const [x, y, z] = allNaturalNumbers(); // x=0; y=1; z=2
```

10.4.2.1 Failing to Array-destructure a value

A value is iterable if it has a method whose key is `Symbol.iterator` that returns an object. Array-destructuring throws a `TypeError` if the value to be destructured isn't iterable:

```
let x;
[x] = [true, false]; // OK, Arrays are iterable
[x] = 'abc'; // OK, strings are iterable
[x] = { * [Symbol.iterator]() { yield 1 } }; // OK, iterable

[x] = {}; // TypeError, empty objects are not iterable
[x] = undefined; // TypeError, not iterable
[x] = null; // TypeError, not iterable
```

The `TypeError` is thrown even before accessing elements of the iterable, which means that you can use the empty Array pattern `[]` to check whether a value is iterable:

```
[] = {}; // TypeError, empty objects are not iterable
[] = undefined; // TypeError, not iterable
[] = null; // TypeError, not iterable
```

10.5 If a part has no match

Similarly to how JavaScript handles non-existent properties and Array elements, destructuring fails silently if the target mentions a part that doesn't exist in the source: the interior of the part is matched against `undefined`. If the interior is a variable that means that the variable is set to `undefined`:

```
const [x] = []; // x = undefined
const {prop:y} = {}; // y = undefined
```

Remember that object patterns and Array patterns throw a `TypeError` if they are matched against `undefined`.

10.5.1 Default values

Default values are a feature of patterns: If a part (an object property or an Array element) has no match in the source, it is matched against:

- its *default value* (if specified)
- `undefined` (otherwise)

That is, providing a default value is optional.

Let's look at an example. In the following destructuring, the element at index 0 has no match on the right-hand side. Therefore, destructuring continues by matching `x` against 3, which leads to `x` being set to 3.

```
const [x=3, y] = []; // x = 3; y = undefined
```

You can also use default values in object patterns:

```
const {foo: x=3, bar: y} = {}; // x = 3; y = undefined
```

10.5.1.1 undefined triggers default values

Default values are also used if a part does have a match and that match is `undefined`:

```
const [x=1] = [undefined]; // x = 1
const {prop: y=2} = {prop: undefined}; // y = 2
```

The rationale for this behavior is explained in the next chapter, in [the section on parameter default values](#).

10.5.1.2 Default values are computed on demand

The default values themselves are only computed when they are needed. In other words, this destructuring:

```
const {prop: y=someFunc()} = someValue;
```

is equivalent to:

```
let y;
if (someValue.prop === undefined) {
  y = someFunc();
} else {
  y = someValue.prop;
}
```

You can observe that if you use `console.log()`:

```
> function log(x) { console.log(x); return 'YES' }

> const [a=log('hello')] = [];
hello
> a
'YES'

> const [b=log('hello')] = [123];
> b
123
```

In the second destructuring, the default value is not triggered and `log()` is not called.

10.5.1.3 Default values can refer to other variables in the pattern

A default value can refer to any variable, including another variable in the same pattern:

```
const [x=3, y=x] = []; // x=3; y=3
const [x=3, y=x] = [7]; // x=7; y=7
const [x=3, y=x] = [7, 2]; // x=7; y=2
```

However, order matters: the variables `x` and `y` are declared from left to right and produce a `ReferenceError` if they are accessed before their declaration:

```
const [x=y, y=3] = []; // ReferenceError
```

10.5.1.4 Default values for patterns

So far we have only seen default values for variables, but you can also associate them with patterns:

```
const [{ prop: x } = {}] = [];
```

What does this mean? Recall the rule for default values:

If the part has no match in the source, destructuring continues with the default value [...].

The element at index 0 has no match, which is why destructuring continues with:

```
const { prop: x } = {}; // x = undefined
```

You can more easily see why things work this way if you replace the pattern `{ prop: x }` with the variable `pattern`:

```
const [pattern = {}] = [];
```

More complex default values. Let's further explore default values for patterns. In the following example, we assign a value to `x` via the default value `{ prop: 123 }`:

```
const [{ prop: x } = { prop: 123 }] = [];
```

Because the Array element at index 0 has no match on the right-hand side, destructuring continues as follows and `x` is set to 123.

```
const { prop: x } = { prop: 123 }; // x = 123
```

However, `x` is not assigned a value in this manner if the right-hand side has an element at index 0, because then the default value isn't triggered.

```
const [{ prop: x } = { prop: 123 }] = [{}];
```

In this case, destructuring continues with:

```
const { prop: x } = {}; // x = undefined
```

Thus, if you want `x` to be 123 if either the object or the property is missing, you need to specify a default value for `x` itself:

```
const [{ prop: x=123 } = {}] = [{}];
```

Here, destructuring continues as follows, independently of whether the right-hand side is `[{}]` or `[]`.

```
const { prop: x=123 } = {}; // x = 123
```



Still confused?

A [later section](#) explains destructuring from a different angle, as an algorithm. That may give you additional insight.

10.6 More object destructuring features

10.6.1 Property value shorthands

Property value shorthands are a feature of object literals: If the value of a property is provided via a variable whose name is the same as the key, you can omit the key. This works for destructuring, too:

```
const { x, y } = { x: 11, y: 8 }; // x = 11; y = 8
```

This declaration is equivalent to:

```
const { x: x, y: y } = { x: 11, y: 8 };
```

You can also combine property value shorthands with default values:

```
const { x, y = 1 } = {}; // x = undefined; y = 1
```

10.6.2 Computed property keys

Computed property keys are another object literal feature that also works for destructuring: You can specify the key of a property via an expression, if you put it in square brackets:

```
const FOO = 'foo';
const { [FOO]: f } = { foo: 123 }; // f = 123
```

Computed property keys allow you to destructure properties whose keys are symbols:

```
// Create and destructure a property whose key is a symbol
const KEY = Symbol();
const obj = { [KEY]: 'abc' };
const { [KEY]: x } = obj; // x = 'abc'
```

```
// Extract Array.prototype[Symbol.iterator]
const { [Symbol.iterator]: func } = [];
console.log(typeof func); // function
```

10.7 More Array destructuring features

10.7.1 Elision

Elision lets you use the syntax of Array “holes” to skip elements during destructuring:

```
const [, x, y] = ['a', 'b', 'c', 'd']; // x = 'c'; y = 'd'
```

10.7.2 Rest operator (...)

The *rest operator* lets you extract the remaining elements of an Array into an Array. You can only use the operator as the last part inside an Array pattern:

```
const [x, ...y] = ['a', 'b', 'c']; // x='a'; y=['b', 'c']
```



The rest operator extracts data. The same syntax (...) is used by the *spread operator*, which contributes data to Array literals and function calls and is explained in [the next chapter](#).

If the operator can't find any elements, it matches its operand against the empty Array. That is, it never produces `undefined` or `null`. For example:

```
const [x, y, ...z] = ['a']; // x='a'; y=undefined; z=[]
```

The operand of the rest operator doesn't have to be a variable, you can use patterns, too:

```
const [x, ...[y, z]] = ['a', 'b', 'c'];  
// x = 'a'; y = 'b'; z = 'c'
```

The rest operator triggers the following destructuring:

```
[y, z] = ['b', 'c']
```



The spread operator (...) looks exactly like the rest operator, but it is used inside function calls and Array literals (not inside destructuring patterns).

10.8 You can assign to more than just variables

If you assign via destructuring, each assignment target can be everything that is allowed on the left-hand side of a normal assignment, including a reference to a property (`obj.prop`) and a reference to an Array element (`arr[0]`).

```
const obj = {};  
const arr = [];  
  
({ foo: obj.prop, bar: arr[0] } = { foo: 123, bar: true });  
  
console.log(obj); // {prop:123}  
console.log(arr); // [true]
```

You can also assign to object properties and Array elements via the rest operator (...):

```
const obj = {};  
[first, ...obj.prop] = ['a', 'b', 'c'];  
// first = 'a'; obj.prop = ['b', 'c']
```

If you *declare* variables or define parameters via destructuring then you must use simple identifiers, you can't refer to object properties and Array elements.

10.9 Pitfalls of destructuring

There are two things to be mindful of when using destructuring:

- You can't start a statement with a curly brace.
- During destructuring, you can either declare variables or assign to them, but not both.

The next two sections have the details.

10.9.1 Don't start a statement with a curly brace

Because code blocks begin with a curly brace, statements must not begin with one. This is unfortunate when using object destructuring in an assignment:

```
{ a, b } = someObject; // SyntaxError
```

The work-around is to put the complete expression in parentheses:

```
({ a, b } = someObject); // OK
```

The following syntax does not work:

```
(( { a, b } ) = someObject); // SyntaxError
```

With `let`, `var` and `const`, curly braces never cause problems:

```
const { a, b } = someObject; // OK
```

10.9.2 You can't mix declaring and assigning to existing variables

Within a destructuring variable declaration, every variable in the source is declared. In the following example, we are trying to declare the variable `b` and refer to the existing variable `f`, which doesn't work.

```
let f;  
...  
let { foo: f, bar: b } = someObject;  
// During parsing (before running the code):  
// SyntaxError: Duplicate declaration, f
```

The fix is to use a destructuring assignment and to declare `b` beforehand:

```
let f;  
...  
let { bar: b } = someObject;
```



```
let b;
({ foo: f, bar: b } = someObject);
```

10.10 Examples of destructuring

Let's start with a few smaller examples.

The `for-of` loop supports destructuring:

```
const map = new Map().set(false, 'no').set(true, 'yes');
for (const [key, value] of map) {
  console.log(key + ' is ' + value);
}
```

You can use destructuring to swap values. That is something that engines could optimize, so that no Array would be created.

```
[a, b] = [b, a];
```

You can use destructuring to split an Array:

```
const [first, ...rest] = ['a', 'b', 'c'];
// first = 'a'; rest = ['b', 'c']
```

10.10.1 Destructuring returned Arrays

Some built-in JavaScript operations return Arrays. Destructuring helps with processing them:

```
const [all, year, month, day] =
  /^(?!\d\d\d\d)-(?!\d\d)-(?!\d\d)$/
  .exec('2999-12-31');
```

If you are only interested in the groups (and not in the complete match, `all`), you can use elision to skip the array element at index 0:

```
const [, year, month, day] =
  /^(?!\d\d\d\d)-(?!\d\d)-(?!\d\d)$/
  .exec('2999-12-31');
```

`exec()` returns `null` if the regular expression doesn't match. Unfortunately, you can't handle `null` via default values, which is why you must use the Or operator (`||`) in this case:

```
const [, year, month, day] =
  /^(?!\d\d\d\d)-(?!\d\d)-(?!\d\d)$/
  .exec(someStr) || [];
```

`Array.prototype.split()` returns an Array. Therefore, destructuring is useful if you are interested in the elements, not the Array:

```
const cells = 'Jane\tDoe\tCTO'
const [firstName, lastName, title] = cells.split('\t');
console.log(firstName, lastName, title);
```

10.10.2 Destructuring returned objects

Destructuring is also useful for extracting data from objects that are returned by functions or methods. For example, the iterator method `next()` returns an object with two properties, `done` and `value`. The following code logs all elements of Array `arr` via the iterator `iter`. Destructuring is used in line A.

```
const arr = ['a', 'b'];
const iter = arr[Symbol.iterator]();
while (true) {
  const {done, value} = iter.next(); // (A)
  if (done) break;
  console.log(value);
}
```

10.10.3 Array-destructuring iterable values

Array-destructuring works with any iterable value. That is occasionally useful:

```
const [x, y] = new Set().add('a').add('b');
// x = 'a'; y = 'b'

const [a, b] = 'foo';
// a = 'f'; b = 'o'
```

10.10.4 Multiple return values

To see the usefulness of multiple return values, let's implement a function `findElement(a, p)` that searches for the first element in the Array `a` for which the function `p` returns `true`. The question is: what should that function return? Sometimes one is interested in the element itself, sometimes in its index, sometimes in both. The following implementation returns both.

```
function findElement(array, predicate) {
  for (const [index, element] of array.entries()) { // (A)
    if (predicate(element)) {
      return { element, index }; // (B)
    }
  }
}
```

```

    }
  }
  return { element: undefined, index: -1 };
}

```

In line A, the Array method `entries()` returns an iterable over `[index,element]` pairs. We destructure one pair per iteration. In line B, we use property value shorthands to return the object `{ element: element, index: index }`.

Let's use `findElement()`. In the following example, several ECMAScript 6 features allow us to write more concise code: The callback is an arrow function, the return value is destructured via an object pattern with property value shorthands.

```

const arr = [7, 8, 6];
const {element, index} = findElement(arr, x => x % 2 === 0);
// element = 8, index = 1

```

Due to `index` and `element` also referring to property keys, the order in which we mention them doesn't matter:

```

const {index, element} = findElement(...);

```

We have successfully handled the case of needing both `index` and `element`. What if we are only interested in one of them? It turns out that, thanks to ECMAScript 6, our implementation can take care of that, too. And the syntactic overhead compared to functions with single return values is minimal.

```

const a = [7, 8, 6];
const {element} = findElement(a, x => x % 2 === 0);
// element = 8

const {index} = findElement(a, x => x % 2 === 0);
// index = 1

```

Each time, we only extract the value of the one property that we need.

10.11 The destructuring algorithm

This section looks at destructuring from a different angle: as a recursive pattern matching algorithm.



This different angle should especially help with understanding default values. If you feel you don't fully understand them yet, read on.

At the end, I'll use the algorithm to explain the difference between the following two function declarations.

```

function move({x=0, y=0} = {}) { ... }
function move({x, y} = { x: 0, y: 0 }) { ... }

```

10.11.1 The algorithm

A destructuring assignment looks like this:

```

«pattern» = «value»

```

We want to use `pattern` to extract data from `value`. I'll now describe an algorithm for doing so, which is known in functional programming as *pattern matching* (short: *matching*). The algorithm specifies the operator `←` ("match against") for destructuring assignment that matches a `pattern` against a `value` and assigns to variables while doing so:

```

«pattern» ← «value»

```

The algorithm is specified via recursive rules that take apart both operands of the `←` operator. The declarative notation may take some getting used to, but it makes the specification of the algorithm more concise. Each rule has two parts:

- The head specifies which operands are handled by the rule.
- The body specifies what to do next.

I only show the algorithm for destructuring assignment. Destructuring variable declarations and destructuring parameter definitions work similarly.

I don't cover advanced features (computed property keys; property value shorthands; object properties and array elements as assignment targets), either. Only the basics.

10.11.1.1 Patterns

A pattern is either:

- A variable: `x`
- An object pattern: `{«properties»}`
- An Array pattern: `[«elements»]`

Each of the following sections describes one of these three cases.

10.11.1.2 Variable

- (1) $x \leftarrow \text{value}$ (including undefined and null)
 $x = \text{value}$

10.11.1.3 Object pattern

- (2a) $\{\langle\text{properties}\rangle\} \leftarrow \text{undefined}$
`throw new TypeError();`
- (2b) $\{\langle\text{properties}\rangle\} \leftarrow \text{null}$
`throw new TypeError();`
- (2c) $\{\text{key}: \langle\text{pattern}\rangle, \langle\text{properties}\rangle\} \leftarrow \text{obj}$
 `$\langle\text{pattern}\rangle \leftarrow \text{obj}.\text{key}$
 $\{\langle\text{properties}\rangle\} \leftarrow \text{obj}$`
- (2d) $\{\text{key}: \langle\text{pattern}\rangle = \text{default_value}, \langle\text{properties}\rangle\} \leftarrow \text{obj}$
`const tmp = obj.key;
if (tmp !== undefined) {
 $\langle\text{pattern}\rangle \leftarrow \text{tmp}$
} else {
 $\langle\text{pattern}\rangle \leftarrow \text{default_value}$
}
 $\{\langle\text{properties}\rangle\} \leftarrow \text{obj}$`
- (2e) $\{\} \leftarrow \text{obj}$
`// No properties left, nothing to do`

10.11.1.4 Array pattern

Array pattern and iterable. The algorithm for Array destructuring starts with an Array pattern and an iterable:

- (3a) $[\langle\text{elements}\rangle] \leftarrow \text{non_iterable}$
`assert(!isIterable(non_iterable))`
`throw new TypeError();`
- (3b) $[\langle\text{elements}\rangle] \leftarrow \text{iterable}$
`assert(isIterable(iterable))`
`const iterator = iterable[Symbol.iterator]();
 $\langle\text{elements}\rangle \leftarrow \text{iterator}$`

Helper function:

```
function isIterable(value) {  
  return (value !== null  
    && typeof value === 'object'  
    && typeof value[Symbol.iterator] === 'function');  
}
```

Array elements and iterator. The algorithm continues with the elements of the pattern (left-hand side of the arrow) and the iterator that was obtained from the iterable (right-hand side of the arrow).

- (3c) $\langle\text{pattern}\rangle, \langle\text{elements}\rangle \leftarrow \text{iterator}$
 `$\langle\text{pattern}\rangle \leftarrow \text{getNext}(\text{iterator})$ // undefined after last item
 $\langle\text{elements}\rangle \leftarrow \text{iterator}$`
- (3d) $\langle\text{pattern}\rangle = \text{default_value}, \langle\text{elements}\rangle \leftarrow \text{iterator}$
`const tmp = getNext(iterator); // undefined after last item
if (tmp !== undefined) {
 $\langle\text{pattern}\rangle \leftarrow \text{tmp}$
} else {
 $\langle\text{pattern}\rangle \leftarrow \text{default_value}$
}
 $\langle\text{elements}\rangle \leftarrow \text{iterator}$`
- (3e) $\langle\text{pattern}\rangle, \langle\text{elements}\rangle \leftarrow \text{iterator}$ (hole, elision)
`getNext(iterator); // skip
 $\langle\text{elements}\rangle \leftarrow \text{iterator}$`
- (3f) $\dots\langle\text{pattern}\rangle \leftarrow \text{iterator}$ (always last part!)
`const tmp = [];
for (const elem of iterator) {
 tmp.push(elem);
}
 $\langle\text{pattern}\rangle \leftarrow \text{tmp}$`
- (3g) $\langle\text{pattern}\rangle \leftarrow \text{iterator}$

```
// No elements left, nothing to do
```

Helper function:

```
function getNext(iterator) {  
  const {done,value} = iterator.next();  
  return (done ? undefined : value);  
}
```

10.11.2 Applying the algorithm

The following function definition has *named parameters*, a technique that is sometimes called *options object* and explained in [the chapter on parameter handling](#). The parameters use destructuring and default values in such a way that `x` and `y` can be omitted. But the object with the parameter can be omitted, too, as you can see in the last line of the code below. This feature is enabled via the `= {}` in the head of the function definition.

```
function move1({x=0, y=0} = {}) {  
  return [x, y];  
}  
move1({x: 3, y: 8}); // [3, 8]  
move1({x: 3}); // [3, 0]  
move1({}); // [0, 0]  
move1(); // [0, 0]
```

But why would you define the parameters as in the previous code snippet? Why not as follows – which is also completely legal ES6 code?

```
function move2({x, y} = { x: 0, y: 0 }) {  
  return [x, y];  
}
```

To see why `move1()` is correct, let's use both functions for two examples. Before we do that, let's see how the passing of parameters can be explained via matching.

10.11.2.1 Background: passing parameters via matching

For function calls, formal parameters (inside function definitions) are matched against actual parameters (inside function calls). As an example, take the following function definition and the following function call.

```
function func(a=0, b=0) { ... }  
func(1, 2);
```

The parameters `a` and `b` are set up similarly to the following destructuring.

```
[a=0, b=0] ← [1, 2]
```

10.11.2.2 Using `move2()`

Let's examine how destructuring works for `move2()`.

Example 1. `move2()` leads to this destructuring:

```
[{x, y} = { x: 0, y: 0 }] ← []
```

The only Array element on the left-hand side does not have a match on the right-hand side, which is why `{x, y}` is matched against the default value and not against data from the right-hand side (rules 3b, 3d):

```
{x, y} ← { x: 0, y: 0 }
```

The left-hand side contains *property value shorthands*, it is an abbreviation for:

```
{x: x, y: y} ← { x: 0, y: 0 }
```

This destructuring leads to the following two assignments (rule 2c, 1):

```
x = 0;  
y = 0;
```

However, this is the only case in which the default value is used.

Example 2. Let's examine the function call `move2({z:3})` which leads to the following destructuring:

```
[{x, y} = { x: 0, y: 0 }] ← [{z:3}]
```

There is an Array element at index 0 on the right-hand side. Therefore, the default value is ignored and the next step is (rule 3d):

```
{x, y} ← { z: 3 }
```

That leads to both `x` and `y` being set to `undefined`, which is not what we want.

10.11.2.3 Using `move1()`

Let's try `move1()`.

Example 1: `move1()`

```
[{x=0, y=0} = {}] ← []
```

We don't have an Array element at index 0 on the right-hand side and use the default value (rule 3d):

```
{x=0, y=0} ← {}
```

The left-hand side contains property value shorthands, which means that this destructuring is equivalent to:

```
{x: x=0, y: y=0} ← {}
```

Neither property `x` nor property `y` have a match on the right-hand side. Therefore, the default values are used and the following destructurings are performed next (rule 2d):

```
x ← 0  
y ← 0
```

That leads to the following assignments (rule 1):

```
x = 0  
y = 0
```

Example 2: `move1({z:3})`

```
[{x=0, y=0} = {}] ← [{z:3}]
```

The first element of the Array pattern has a match on the right-hand side and that match is used to continue destructuring (rule 3d):

```
{x=0, y=0} ← {z:3}
```

Like in example 1, there are no properties `x` and `y` on the right-hand side and the default values are used:

```
x = 0  
y = 0
```

10.11.3 Conclusion

The examples demonstrate that default values are a feature of pattern parts (object properties or Array elements). If a part has no match or is matched against `undefined` then the default value is used. That is, the pattern is matched against the default value, instead.

11. Parameter handling



For this chapter, it is useful to be familiar with destructuring (which is explained in [the previous chapter](#)).

11.1 Overview

Parameter handling has been significantly upgraded in ECMAScript 6. It now supports parameter default values, rest parameters (varargs) and destructuring.

Default parameter values:

```
function findClosestShape(x=0, y=0) {  
  // ...  
}
```

Rest parameters:

```
function format(pattern, ...params) {  
  return params;  
}  
console.log(format('a', 'b', 'c')); // ['b', 'c']
```

Named parameters via destructuring:

```
function selectEntries({ start=0, end=-1, step=1 } = {}) {  
  // The object pattern is an abbreviation of:  
  // { start: start=0, end: end=-1, step: step=1 }  
  
  // Use the variables `start`, `end` and `step` here  
  ...  
}  
  
selectEntries({ start: 10, end: 30, step: 2 });  
selectEntries({ step: 3 });
```

```
selectEntries({});
selectEntries();
```

11.1.1 Spread operator (...)

In function and constructor calls, the spread operator turns iterable values into arguments:

```
> Math.max(-1, 5, 11, 3)
11
> Math.max(...[-1, 5, 11, 3])
11
> Math.max(-1, ...[-1, 5, 11], 3)
11
```

In Array literals, the spread operator turns iterable values into Array elements:

```
> [1, ...[2,3], 4]
[1, 2, 3, 4]
```

11.2 Parameter handling as destructuring

The ES6 way of handling parameters is equivalent to destructuring the actual parameters via the formal parameters. That is, the following function call:

```
function func («FORMAL_PARAMETERS») {
  «CODE»
}
func («ACTUAL_PARAMETERS»);
```

is roughly equivalent to:

```
{
  let [«FORMAL_PARAMETERS»] = [«ACTUAL_PARAMETERS»];
  {
    «CODE»
  }
}
```

Example – the following function call:

```
function logSum(x=0, y=0) {
  console.log(x + y);
}
logSum(7, 8);
```

becomes:

```
{
  let [x=0, y=0] = [7, 8];
  {
    console.log(x + y);
  }
}
```

Let's look at specific features next.

11.3 Parameter default values

ECMAScript 6 lets you specify default values for parameters:

```
function f(x, y=0) {
  return [x, y];
}
```

Omitting the second parameter triggers the default value:

```
> f(1)
[1, 0]
> f()
[undefined, 0]
```

Watch out – `undefined` triggers the default value, too:

```
> f(undefined, undefined)
[undefined, 0]
```

The default value is computed on demand, only when it is actually needed:

```
> const log = console.log.bind(console);
> function g(x=log('x'), y=log('y')) {return 'DONE'}
> g()
x
y
'DONE'
> g(1)
y
'DONE'
> g(1, 2)
'DONE'
```

11.3.1 Why does `undefined` trigger default values?

It isn't immediately obvious why `undefined` should be interpreted as a missing parameter or a missing part of an object or Array. The rationale for doing so is that it

enables you to delegate the definition of default values. Let's look at two examples.

In the first example (source: [Rick Waldron's TC39 meeting notes from 2012-07-24](#)), we don't have to define a default value in `setOptions()`, we can delegate that task to `setLevel()`.

```
function setLevel(newLevel = 0) {
  light.intensity = newLevel;
}
function setOptions(options) {
  // Missing prop returns undefined => use default
  setLevel(options.dimmerLevel);
  setMotorSpeed(options.speed);
  ...
}
setOptions({speed:5});
```

In the second example, `square()` doesn't have to define a default for `x`, it can delegate that task to `multiply()`:

```
function multiply(x=1, y=1) {
  return x * y;
}
function square(x) {
  return multiply(x, x);
}
```

Default values further entrench the role of `undefined` as indicating that something doesn't exist, versus `null` indicating emptiness.

11.3.2 Referring to other parameters in default values

Within a parameter default value, you can refer to any variable, including other parameters:

```
function foo(x=3, y=x) { ... }
foo(); // x=3; y=3
foo(7); // x=7; y=7
foo(7, 2); // x=7; y=2
```

However, order matters: parameters are declared from left to right and within a default value, you get a `ReferenceError` if you access a parameter that hasn't been declared, yet.

11.3.3 Referring to "inner" variables in default values

Default values exist in their own scope, which is between the "outer" scope surrounding the function and the "inner" scope of the function body. Therefore, you can't access "inner" variables from the default values:

```
const x = 'outer';
function foo(a = x) {
  const x = 'inner';
  console.log(a); // outer
}
```

If there were no outer `x` in the previous example, the default value `x` would produce a `ReferenceError` (if triggered).

This restriction is probably most surprising if default values are closures:

```
function bar(callback = () => QUX) {
  const QUX = 3; // can't be accessed from default value
  callback();
}
bar(); // ReferenceError
```

To see why that is the case, consider the following implementation of `bar()` which is roughly equivalent to the previous one:

```
function bar(...args) { // (A)
  const [callback = () => QUX] = args; // (B)
  { // (C)
    const QUX = 3; // can't be accessed from default value
    callback();
  }
}
```

Within the scope started by the opening curly brace at the end of line A, you can only refer to variables that are declared either in that scope or in a scope surrounding it. Therefore, variables declared in the scope starting in line C are out of reach for the statement in line B.

11.4 Rest parameters

Putting the rest operator (`...`) in front of the last formal parameter means that it will receive all remaining actual parameters in an Array.

```
function f(x, ...y) {
  ...
}
f('a', 'b', 'c'); // x = 'a'; y = ['b', 'c']
```

If there are no remaining parameters, the rest parameter will be set to the empty Array:

```
f(); // x = undefined; y = []
```



The **spread operator** (`...`) looks exactly like the rest operator, but it is used inside function calls and Array literals (not inside destructuring patterns).

11.4.1 No more arguments!

Rest parameters can completely replace JavaScript's infamous special variable `arguments`. They have the advantage of always being Arrays:

```
// ECMAScript 5: arguments
function logAllArguments() {
  for (var i=0; i < arguments.length; i++) {
    console.log(arguments[i]);
  }
}

// ECMAScript 6: rest parameter
function logAllArguments(...args) {
  for (const arg of args) {
    console.log(arg);
  }
}
```

11.4.1.1 Combining destructuring and access to the destructured value

One interesting feature of `arguments` is that you can have normal parameters and an Array of all parameters at the same time:

```
function foo(x=0, y=0) {
  console.log('Arity: '+arguments.length);
  ...
}
```

You can avoid `arguments` in such cases if you combine a rest parameter with Array destructuring. The resulting code is longer, but more explicit:

```
function foo(...args) {
  let [x=0, y=0] = args;
  console.log('Arity: '+args.length);
  ...
}
```

The same technique works for named parameters (options objects):

```
function bar(options = {}) {
  let { namedParam1, namedParam2 } = options;
  ...
  if ('extra' in options) {
    ...
  }
}
```

11.4.1.2 arguments is iterable

`arguments` is *iterable*⁸ in ECMAScript 6, which means that you can use `for-of` and the spread operator:

```
> (function () { return typeof arguments[Symbol.iterator] }())
'function'
> (function () { return Array.isArray([...arguments]) }())
true
```

11.5 Simulating named parameters

When calling a function (or method) in a programming language, you must map the actual parameters (specified by the caller) to the formal parameters (of a function definition). There are two common ways to do so:

- *Positional parameters* are mapped by position. The first actual parameter is mapped to the first formal parameter, the second actual to the second formal, and so on.
- *Named parameters* use *names* (labels) to perform the mapping. Names are associated with formal parameters in a function definition and label actual parameters in a function call. It does not matter in which order named parameters appear, as long as they are correctly labeled.

Named parameters have two main benefits: they provide descriptions for arguments in function calls and they work well for optional parameters. I'll first explain the benefits and then show you how to simulate named parameters in JavaScript via object literals.

11.5.1 Named Parameters as Descriptions

As soon as a function has more than one parameter, you might get confused about what each parameter is used for. For example, let's say you have a function,

`selectEntries()`, that returns entries from a database. Given the function call:

```
selectEntries(3, 20, 2);
```

what do these three numbers mean? Python supports named parameters, and they make it easy to figure out what is going on:

```
# Python syntax
selectEntries(start=3, end=20, step=2)
```

11.5.2 Optional Named Parameters

Optional positional parameters work well only if they are omitted at the end. Anywhere else, you have to insert placeholders such as `null` so that the remaining parameters have correct positions.

With optional named parameters, that is not an issue. You can easily omit any of them. Here are some examples:

```
# Python syntax
selectEntries(step=2)
selectEntries(end=20, start=3)
selectEntries()
```

11.5.3 Simulating Named Parameters in JavaScript

JavaScript does not have native support for named parameters like Python and many other languages. But there is a reasonably elegant simulation: name parameters via an object literal, passed as a single actual parameter. When you use this technique, an invocation of `selectEntries()` looks as follows:

```
selectEntries({ start: 3, end: 20, step: 2 });
```

The function receives an object with the properties `start`, `end`, and `step`. You can omit any of them:

```
selectEntries({ step: 2 });
selectEntries({ end: 20, start: 3 });
selectEntries();
```

In ECMAScript 5, you'd implement `selectEntries()` as follows:

```
function selectEntries(options) {
  options = options || {};
  var start = options.start || 0;
  var end = options.end || -1;
  var step = options.step || 1;
  ...
}
```

In ECMAScript 6, you can use destructuring, which looks like this:

```
function selectEntries({ start=0, end=-1, step=1 }) {
  ...
}
```

If you call `selectEntries()` with zero arguments, the destructuring fails, because you can't match an object pattern against `undefined`. That can be fixed via a default value. In the following code, the object pattern is matched against `{}` if there isn't at least one argument.

```
function selectEntries({ start=0, end=-1, step=1 } = {}) {
  ...
}
```

You can also combine positional parameters with named parameters. It is customary for the latter to come last:

```
someFunc(posArg1, { namedArg1: 7, namedArg2: true });
```

In principle, JavaScript engines could optimize this pattern so that no intermediate object is created, because both the object literals at the call sites and the object patterns in the function definitions are static.



In JavaScript, the pattern for named parameters shown here is sometimes called *options* or *option object* (e.g., by the jQuery documentation).

11.6 Examples of destructuring in parameter handling

11.6.1 `forEach()` and destructuring

You will probably mostly use the `for-of` loop in ECMAScript 6, but the Array method `forEach()` also profits from destructuring. Or rather, its callback does.

First example: destructuring the Arrays in an Array.

```
const items = [ ['foo', 3], ['bar', 9] ];
items.forEach(([,word, count]) => {
  console.log(word+' '+count);
});
```

Second example: destructuring the objects in an Array.

```
const items = [
  { word:'foo', count:3 },
  { word:'bar', count:9 },
];
items.forEach(({word, count}) => {
  console.log(word+' '+count);
});
```

11.6.2 Transforming Maps

An ECMAScript 6 Map doesn't have a method `map()` (like Arrays). Therefore, one has to:

1. Convert it to an Array of `[key,value]` pairs.
2. `map()` the Array.
3. Convert the result back to a Map.

This looks as follows.

```
const map0 = new Map([
  [1, 'a'],
  [2, 'b'],
  [3, 'c'],
]);

const map1 = new Map( // step 3
  [...map0] // step 1
  .map(([k, v]) => [k*2, '_'+v]) // step 2
);
// Resulting Map: {2 -> '_a', 4 -> '_b', 6 -> '_c'}
```

11.6.3 Handling an Array returned via a Promise

The tool method `Promise.all()` works as follows:

- Input: an Array of Promises.
- Output: a Promise that resolves to an Array as soon as the last input Promise is resolved. The Array contains the resolutions of the input Promises.

Destructuring helps with handling the Array that the result of `Promise.all()` resolves to:

```
const urls = [
  'http://example.com/foo.html',
  'http://example.com/bar.html',
  'http://example.com/baz.html',
];

Promise.all(urls.map(downloadUrl))
  .then([fooStr, barStr, bazStr]) => {
    ...
  });

// This function returns a Promise that resolves to
// a string (the text)
function downloadUrl(url) {
  return fetch(url).then(request => request.text());
}
```

`fetch()` is a Promise-based version of `XMLHttpRequest`. It is [part of the Fetch standard](#).

11.7 Coding style tips

This section mentions a few tricks for descriptive parameter definitions. They are clever, but they also have downsides: they add visual clutter and can make your code harder to understand.

11.7.1 Optional parameters

I occasionally use the parameter default value `undefined` to mark a parameter as optional (unless it already has a default value):

```
function foo(requiredParam, optionalParam = undefined) {
  ...
}
```

11.7.2 Required parameters

In ECMAScript 5, you have a few options for ensuring that a required parameter has been provided, which are all quite clumsy:

```
function foo(mustBeProvided) {
  if (arguments.length < 1) {
    throw new Error();
  }
  if (! (0 in arguments)) {
    throw new Error();
  }
  if (mustBeProvided === undefined) {
```

```

        throw new Error();
    }
    ...
}

```

In ECMAScript 6, you can (ab)use default parameter values to achieve more concise code (credit: idea by Allen Wirfs-Brock):

```

/**
 * Called if a parameter is missing and
 * the default value is evaluated.
 */
function mandatory() {
    throw new Error('Missing parameter');
}
function foo(mustBeProvided = mandatory()) {
    return mustBeProvided;
}

```

Interaction:

```

> foo()
Error: Missing parameter
> foo(123)
123

```

11.7.3 Enforcing a maximum arity

This section presents three approaches to enforcing a maximum arity. The running example is a function `f` whose maximum arity is 2 – if a caller provides more than 2 parameters, an error should be thrown.

The first approach collects all actual parameters in the formal rest parameter `args` and checks its length.

```

function f(...args) {
    if (args.length > 2) {
        throw new Error();
    }
    // Extract the real parameters
    let [x, y] = args;
}

```

The second approach relies on unwanted actual parameters appearing in the formal rest parameter `empty`.

```

function f(x, y, ...empty) {
    if (empty.length > 0) {
        throw new Error();
    }
}

```

The third approach uses a sentinel value that is gone if there is a third parameter. One caveat is that the default value `OK` is also triggered if there is a third parameter whose value is `undefined`.

```

const OK = Symbol();
function f(x, y, arity=OK) {
    if (arity !== OK) {
        throw new Error();
    }
}

```

Sadly, each one of these approaches introduces significant visual and conceptual clutter. I'm tempted to recommend checking `arguments.length`, but I also want `arguments` to go away.

```

function f(x, y) {
    if (arguments.length > 2) {
        throw new Error();
    }
}

```

11.8 The spread operator (...)

The spread operator (...) looks exactly like the rest operator, but is its opposite:

- The rest operator collects the remaining items of an iterable value into an Array and is used for [rest parameters](#) and [destructuring](#).
- The spread operator turns the items of an iterable value into arguments of a function call or into elements of an Array.

11.8.1 Spreading into function and method calls

`Math.max()` is a good example for demonstrating how the spread operator works in method calls. `Math.max(x1, x2, ...)` returns the argument whose value is greatest. It accepts an arbitrary number of arguments, but can't be applied to Arrays. The spread operator fixes that:

```

> Math.max(-1, 5, 11, 3)
11
> Math.max(...[-1, 5, 11, 3])
11

```

In contrast to the rest operator, you can use the spread operator anywhere in a sequence of parts:

```
> Math.max(-1, ...[-1, 5, 11], 3)
11
```

Another example is JavaScript not having a way to destructively append the elements of one Array to another one. However, Arrays do have the method `push(x1, x2, ...)`, which appends all of its arguments to its receiver. The following code shows how you can use `push()` to append the elements of `arr2` to `arr1`.

```
const arr1 = ['a', 'b'];
const arr2 = ['c', 'd'];

arr1.push(...arr2);
// arr1 is now ['a', 'b', 'c', 'd']
```

11.8.2 Spreading into constructors

In addition to function and method calls, the spread operator also works for constructor calls:

```
new Date(...[1912, 11, 24]) // Christmas Eve 1912
```

That is something that is [difficult to achieve in ECMAScript 5](#).

11.8.3 Spreading into Arrays

The spread operator can also be used inside Array literals:

```
> [1, ...[2,3], 4]
[1, 2, 3, 4]
```

That gives you a convenient way to concatenate Arrays:

```
const x = ['a', 'b'];
const y = ['c'];
const z = ['d', 'e'];

const arr = [...x, ...y, ...z]; // ['a', 'b', 'c', 'd', 'e']
```

One advantage of the spread operator is that its operand can be any iterable value (`concat()` does not support iteration).

11.8.3.1 Converting iterable or Array-like objects to Arrays

The spread operator lets you convert any iterable value to an Array:

```
const arr = [...someIterableObject];
```

Let's convert a Set to an Array:

```
const set = new Set([11, -1, 6]);
const arr = [...set]; // [11, -1, 6]
```

Your own iterable objects can be converted to Arrays in the same manner:

```
const obj = {
  * [Symbol.iterator]() {
    yield 'a';
    yield 'b';
    yield 'c';
  }
};
const arr = [...obj]; // ['a', 'b', 'c']
```

Note that, just like the `for-of` loop, the spread operator only works for iterable values. Most important objects are iterable: Arrays, Maps, Sets and `arguments`. Most DOM data structures will also eventually be iterable.

Should you ever encounter something that is not iterable, but Array-like (indexed elements plus a property `length`), you can use `Array.from()`⁹ to convert it to an Array.

```
const arrayLike = {
  '0': 'a',
  '1': 'b',
  '2': 'c',
  length: 3
};

// ECMAScript 5:
var arr1 = [].slice.call(arrayLike); // ['a', 'b', 'c']

// ECMAScript 6:
const arr2 = Array.from(arrayLike); // ['a', 'b', 'c']

// TypeError: Cannot spread non-iterable value
const arr3 = [...arrayLike];
```

III Modularity

12. Callable entities in ECMAScript 6

This chapter gives advice on **how to properly use entities you can call** (via function calls, method calls, etc.) in ES6.

Sections in this chapter:

- [An overview of callable entities in ES6](#)
- [Ways of calling in ES6](#)
- [Recommendations for using callable entities](#)
- [ES6 callable entities in detail](#)
- [Dispatched and direct method calls in ES5 and ES6](#)
- [The `name` property of functions](#)
- [FAQ: callable entities](#)

12.1 Overview

In ES5, a single construct, the (traditional) function, played three roles:

- Real (non-method) function
- Method
- Constructor

In ES6, there is more specialization. The three duties are now handled as follows (a class definition is either one of the two constructs for creating classes – a class declaration or a class expression):

- **Real (non-method) function:**
 - Arrow functions (only have an expression form)
 - Traditional functions (created via function expressions and function declarations)
 - Generator functions (created via generator function expressions and generator function declarations)
- **Method:**
 - Methods (created by method definitions in object literals and class definitions)
 - Generator methods (created by generator method definitions in object literals and class definitions)
- **Constructor:**
 - Classes (created via class definitions)

This list is a simplification. There are quite a few libraries that use `this` as an implicit parameter for callbacks. Then you have to use traditional functions.

Note that I distinguish:

- The entity: e.g. traditional function
- The syntax that creates the entity: e.g. function expression and function declaration

Even though their behaviors differ (as explained later), all of these entities are functions. For example:

```
> typeof (() => {}) // arrow function
'function'
> typeof function* () {} // generator function
'function'
> typeof class {} // class
'function'
```

12.2 Ways of calling in ES6

Some calls can be made anywhere, others are restricted to specific locations.

12.2.1 Calls that can be made anywhere

Three kinds of calls can be made anywhere in ES6:

- Function calls: `func(3, 1)`
- Method calls: `obj.method('abc')`
- Constructor calls: `new Constr(8)`

For function calls, it is important to remember that most ES6 code will be contained in modules and that module bodies are implicitly in strict mode.

12.2.2 Calls via `super` are restricted to specific locations

Two kinds of calls can be made via the `super` keyword; their use is restricted to specific locations:

- Super-method calls: `super.method('abc')`
Only available within method definitions inside either object literals or derived class definitions.
- Super-constructor calls: `super(8)`
Only available inside the special method `constructor()` inside a derived class definition.

12.2.3 Non-method functions versus methods

The difference between non-method functions and methods is becoming more pronounced in ECMAScript 6. There are now special entities for both and things that only they can do:

- Arrow functions are made for non-method functions. They pick up `this` (and other variables) from their surrounding scopes ("lexical `this`").
- Method definitions are made for methods. They provide support for `super`, to refer to super-properties and to make super-method calls.

12.3 Recommendations for using callable entities

This section gives tips for using callable entities: When it's best to use which entity; etc.

12.3.1 Prefer arrow functions as callbacks

As callbacks, arrow functions have two advantages over traditional functions:

- `this` is lexical and therefore safer to use.
- Their syntax is more compact. That matters especially in functional programming, where there are many higher-order functions and methods (functions and methods whose parameters are functions).



For callbacks that span multiple lines, I find traditional function expressions acceptable, too. But you have to be careful with `this`.

12.3.1.1 Problem: `this` as an implicit parameter

Alas, some JavaScript APIs use `this` as an implicit argument for their callbacks, which prevents you from using arrow functions. For example: The `this` in line B is an implicit argument of the function in line A.

```
beforeEach(function () { // (A)
  this.addMatchers({ // (B)
    toBeInRange: function (start, end) {
      ...
    }
  });
});
```

This pattern is less explicit and prevents you from using arrow functions.

12.3.1.2 Solution 1: change the API

This is easy to fix, but requires the API to change:

```
beforeEach(api => {
  api.addMatchers({
    toBeInRange(start, end) {
      ...
    }
  });
});
```

We have turned the API from an implicit parameter `this` into an explicit parameter `api`. I like this kind of explicitness.

12.3.1.3 Solution 2: access the value of `this` in some other way

In some APIs, there are alternate ways to get to the value of `this`. For example, the following code uses `this`.

```
var $button = $('#myButton');
$button.on('click', function () {
  this.classList.toggle('clicked');
});
```

But the target of the event can also be accessed via `event.target`:

```
var $button = $('#myButton');
$button.on('click', event => {
  event.target.classList.toggle('clicked');
});
```

12.3.2 Prefer function declarations as stand-alone functions

As stand-alone functions (versus callbacks), I prefer function declarations:

```
function foo(arg1, arg2) {  
  ...  
}
```

The benefits are:

- Subjectively, I find they look nicer. In this case, the verbose keyword `function` is an advantage – you want the construct to stand out.
- They look like generator function declarations, leading to more visual consistency of the code.

There is one caveat: Normally, you don't need `this` in stand-alone functions. If you use it, you want to access the `this` of the surrounding scope (e.g. a method which contains the stand-alone function). Alas, function declarations don't let you do that – they have their own `this`, which shadows the `this` of the surrounding scope. Therefore, you may want to let a linter warn you about `this` in function declarations.

Another option for stand-alone functions is assigning arrow functions to variables. Problems with `this` are avoided, because it is lexical.

```
const foo = (arg1, arg2) => {  
  ...  
};
```

12.3.3 Prefer method definitions for methods

Method definitions are the only way to create methods that use `super`. They are the obvious choice in object literals and classes (where they are the only way to define methods), but what about adding a method to an existing object? For example:

```
MyClass.prototype.foo = function (arg1, arg2) {  
  ...  
};
```

The following is a quick way to do the same thing in ES6 (caveat: `Object.assign()` doesn't move methods with `super` properly).

```
Object.assign(MyClass.prototype, {  
  foo(arg1, arg2) {  
    ...  
  }  
});
```

For more information and caveats, consult [the section on `Object.assign\(\)`](#).

12.3.4 Methods versus callbacks

There is a subtle difference between an object with methods and an object with callbacks.

12.3.4.1 An object whose properties are methods

The `this` of a method is the *receiver* of the method call (e.g. `obj` if the method call is `obj.m(...)`).

For example, you can use [the WHATWG streams API](#) as follows:

```
const surroundingObject = {  
  surroundingMethod() {  
    const obj = {  
      data: 'abc',  
      start(controller) {  
        ...  
        console.log(this.data); // abc (*)  
        this.pull(); // (**)  
        ...  
      },  
      pull() {  
        ...  
      },  
      cancel() {  
        ...  
      },  
    };  
    const stream = new ReadableStream(obj);  
  },  
};
```

That is, `obj` is an object whose properties `start`, `pull` and `cancel` are methods. Accordingly, these methods can use `this` to access object-local state (line `*`) and to call each other (line `**`).

12.3.4.2 An object whose properties are callbacks

The `this` of an arrow function is the `this` of the surrounding scope (*lexical this*). Arrow functions make great callbacks, because that is the behavior you normally want for

callbacks (real, non-method functions). A callback shouldn't have its own `this` that shadows the `this` of the surrounding scope.

If the properties `start`, `pull` and `cancel` are arrow functions then they pick up the `this` of `surroundingMethod()` (their surrounding scope):

```
const surroundingObject = {
  surroundingData: 'xyz',
  surroundingMethod() {
    const obj = {
      start: controller => {
        ...
        console.log(this.surroundingData); // xyz (*)
        ...
      },

      pull: () => {
        ...
      },

      cancel: () => {
        ...
      },
    };
    const stream = new ReadableStream(obj);
  },
};
const stream = new ReadableStream();
```

If the output in line `*` surprises you then consider the following code:

```
const obj = {
  foo: 123,
  bar() {
    const f = () => console.log(this.foo); // 123
    const o = {
      p: () => console.log(this.foo), // 123
    };
  },
}
```

Inside method `bar()`, `f` and `o.p` work the same, because both arrow functions have the same surrounding lexical scope, `bar()`. The latter arrow function being surrounded by an object literal does not change that.

12.3.5 Avoid IIFEs in ES6

This section gives tips for avoiding IIFEs in ES6.

12.3.5.1 Replace an IIFE with a block

In ES5, you had to use an IIFE if you wanted to keep a variable local:

```
(function () { // open IIFE
  var tmp = ...;
  ...
})(); // close IIFE

console.log(tmp); // ReferenceError
```

In ECMAScript 6, you can simply use a block and a `let` or `const` declaration:

```
{ // open block
  let tmp = ...;
  ...
} // close block

console.log(tmp); // ReferenceError
```

12.3.5.2 Replace an IIFE with a module

In ECMAScript 5 code that doesn't use modules via libraries (such as RequireJS, browserify or webpack), the *revealing module pattern* is popular, and based on an IIFE. Its advantage is that it clearly separates between what is public and what is private:

```
var my_module = (function () {
  // Module-private variable:
  var countInvocations = 0;

  function myFunc(x) {
    countInvocations++;
    ...
  }

  // Exported by module:
  return {
    myFunc: myFunc
  };
})();
```

This module pattern produces a global variable and is used as follows:

```
my_module.myFunc(33);
```

In ECMAScript 6, **modules** are built in, which is why the barrier to adopting them is low:


```
// my_module.js
// Module-private variable:
let countInvocations = 0;

export function myFunc(x) {
  countInvocations++;
  ...
}
```

This module does not produce a global variable and is used as follows:

```
import { myFunc } from 'my_module.js';

myFunc(33);
```

12.3.5.3 Immediately-invoked arrow functions

There is one use case where you still need an immediately-invoked function in ES6: Sometimes you only can produce a result via a sequence of statements, not via a single expression. If you want to inline those statements, you have to immediately invoke a function. In ES6, you can use immediately-invoked arrow functions if you want to:

```
const SENTENCE = 'How are you?';
const REVERSED_SENTENCE = (() => {
  // Iteration over the string gives us code points
  // (better for reversal than characters)
  const arr = [...SENTENCE];
  arr.reverse();
  return arr.join('');
})();
```

Note that you must parenthesize as shown (the parens are around the arrow function, not around the complete function call). Details are explained in [the chapter on arrow functions](#).

12.3.6 Use classes as constructors

In ES5, constructor functions were the mainstream way of creating factories for objects (but there were also many other techniques, some arguably more elegant). In ES6, classes are the mainstream way of implementing constructor functions. Several frameworks support them as alternatives to their custom inheritance APIs.

12.4 ES6 callable entities in detail

This section starts with a cheat sheet, before describing each ES6 callable entity in detail.

12.4.1 Cheat sheet: callable entities

12.4.1.1 The behavior and structure of callable entities

Value:

| | Func decl/Func expr | Arrow | Class | Method |
|-----------------------|---------------------------|-------|-------|--------|
| Function-callable | ✓ | ✓ | × | ✓ |
| Constructor-callable | ✓ | × | ✓ | × |
| Prototype | F.p | F.p | SC | F.p |
| Property prototype | ✓ | × | ✓ | × |

Whole construct:

| | Func decl | Func expr | Arrow | Class | Method |
|-----------------------------------|--------------|--------------|-------|-------|--------|
| Hoisted | ✓ | | | × | |
| Creates window prop. (1) | ✓ | | | × | |
| Inner name (2) | × | ✓ | | ✓ | × |

Body:

| | Func decl | Func expr | Arrow | Class (3) | Method |
|------------|--------------|--------------|-------|--------------|--------|
| this | ✓ | ✓ | lex | ✓ | ✓ |
| new.target | ✓ | ✓ | lex | ✓ | ✓ |
| super.prop | x | x | lex | ✓ | ✓ |
| super() | x | x | x | ✓ | x |

Abbreviations in cells:

- ✓ exists, allowed
- x does not exist, not allowed
- Empty cell: not applicable, not relevant
- lex: lexical, inherited from surrounding lexical scope
- F.p: `Function.prototype`
- SC: superclass for derived classes, `Function.prototype` for base classes. The details are explained in [the chapter on classes](#).

Notes:

- (1) The rules for what declarations create properties for the global object are explained in [the chapter on variables and scoping](#).
- (2) The inner names of named function expressions and classes are explained in [the chapter on classes](#).
- (3) This column is about the body of the class constructor.

What about generator functions and methods? Those work like their non-generator counterparts, with two exceptions:

- Generator functions and methods have the prototype `(GeneratorFunction).prototype` ((`GeneratorFunction`) is an internal object, see diagram in Sect. ["Inheritance within the iteration API \(including generators\)"](#)).
- You can't constructor-call generator functions.

12.4.1.2 The rules for `this`

| | FC strict | FC sloppy | MC | new |
|----------------------|------------------------|------------------------|------------------------|------------------------|
| Traditional function | undefined | window | receiver | instance |
| Generator function | undefined | window | receiver | <code>TypeError</code> |
| Method | undefined | window | receiver | <code>TypeError</code> |
| Generator method | undefined | window | receiver | <code>TypeError</code> |
| Arrow function | lexical | lexical | lexical | <code>TypeError</code> |
| Class | <code>TypeError</code> | <code>TypeError</code> | <code>TypeError</code> | SC protocol |

Abbreviations in column titles:

- FC: function call
- MC: method call

Abbreviations in cells:

- lexical: inherited from surrounding lexical scope
- SC protocol: [subclassing protocol](#) (new instance in base class, received from superclass in derived class)

12.4.2 Traditional functions

These are the functions that you know from ES5. There are two ways to create them:

- Function expression:

```
const foo = function (x) { ... };
```
- Function declaration:

```
function foo(x) { ... }
```

Rules for `this`:

- Function calls: `this` is `undefined` in strict mode and the global object in sloppy mode.
- Method calls: `this` is the receiver of the method call (or the first argument of `call`/`apply`).
- Constructor calls: `this` is the newly created instance.

12.4.3 Generator functions

Generator functions are explained in [the chapter on generators](#). Their syntax is similar to traditional functions, but they have an extra asterisk:

- Generator function expression:

```
const foo = function* (x) { ... };
```

- Generator function declaration:

```
function* foo(x) { ... }
```

The rules for `this` are as follows. Note that it never refers to the generator object.

- Function/method calls: `this` is handled like it is with traditional functions. The results of such calls are generator objects.
- Constructor calls: You can't constructor-call generator functions. A `TypeError` is thrown if you do.

12.4.4 Method definitions

Method definitions can appear [inside object literals](#):

```
const obj = {
  add(x, y) {
    return x + y;
  }, // comma is required
  sub(x, y) {
    return x - y;
  }, // comma is optional
};
```

And [inside class definitions](#):

```
class AddSub {
  add(x, y) {
    return x + y;
  } // no comma
  sub(x, y) {
    return x - y;
  } // no comma
}
```

As you can see, you must separate method definitions in an object literal with commas, but there are no separators between them in a class definition. The former is necessary to keep the syntax consistent, especially with regard to getters and setters.

Method definitions are the only place where you can use `super` to refer to super-properties. Only method definitions that use `super` produce functions that have the property `[[HomeObject]]`, which is required for that feature (details are explained in [the chapter on classes](#)).

Rules:

- Function calls: If you extract a method and call it as a function, it behaves like a traditional function.
- Method calls: work as with traditional functions, but additionally allow you to use `super`.
- Constructor calls: produce a `TypeError`.

Inside class definitions, methods whose name is `constructor` are special, as explained later.

12.4.5 Generator method definitions

Generator methods are explained in [the chapter on generators](#). Their syntax is similar to method definitions, but they have an extra asterisk:

```
const obj = {
  * generatorMethod(...) {
    ...
  },
};
class MyClass {
  * generatorMethod(...) {
    ...
  }
}
```

Rules:

- Calling a generator method returns a generator object.
- You can use `this` and `super` as you would in normal method definitions.

12.4.6 Arrow functions

Arrow functions are explained in [their own chapter](#):

```
const squares = [1,2,3].map(x => x * x);
```

The following variables are *lexical* inside an arrow function (picked up from the surrounding scope):

- arguments
- super
- this
- new.target

Rules:

- Function calls: lexical `this` etc.
- Method calls: You can use arrow functions as methods, but their `this` continues to be lexical and does not refer to the receiver of a method call.
- Constructor calls: produce a `TypeError`.

12.4.7 Classes

Classes are explained in [their own chapter](#).

```
// Base class: no `extends`
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  toString() {
    return `${this.x}, ${this.y}`;
  }
}

// This class is derived from `Point`
class ColorPoint extends Point {
  constructor(x, y, color) {
    super(x, y);
    this.color = color;
  }
  toString() {
    return super.toString() + ' in ' + this.color;
  }
}
```

The Method `constructor` is special, because it “becomes” the class. That is, classes are very similar to constructor functions:

```
> Point.prototype.constructor === Point
true
```

Rules:

- Function/method calls: Classes can’t be called as functions or methods (why is explained in [the chapter on classes](#)).
- Constructor calls: follow [a protocol that supports subclassing](#). In a base class, an instance is created and `this` refers to it. A derived class receives its instance from its superclass, which is why it needs to call `super` before it can access `this`.

12.5 Dispatched and direct method calls in ES5 and ES6

There are two ways to call methods in JavaScript:

- Via dispatch, e.g. `obj.someMethod(arg0, arg1)`
- Directly, e.g. `someFunc.call(thisValue, arg0, arg1)`

This section explains how these two work and why you will rarely call methods directly in ECMAScript 6. Before we get started, I’ll refresh your knowledge w.r.t. to prototype chains.

12.5.1 Background: prototype chains

Remember that each object in JavaScript is actually a chain of one or more objects. The first object inherits properties from the later objects. For example, the prototype chain of an Array `['a', 'b']` looks as follows:

1. The instance, holding the elements `'a'` and `'b'`
2. `Array.prototype`, the properties provided by the `Array` constructor
3. `Object.prototype`, the properties provided by the `Object` constructor
4. `null` (the end of the chain, so not really a member of it)

You can examine the chain via `Object.getPrototypeOf()`:

```
> var arr = ['a', 'b'];
> var p = Object.getPrototypeOf;

> p(arr) === Array.prototype
true
> p(p(arr)) === Object.prototype
true
> p(p(p(arr)))
null
```

Properties in “earlier” objects override properties in “later” objects. For example, `Array.prototype` provides an Array-specific version of the `toString()` method, overriding `Object.prototype.toString()`.

```
> var arr = ['a', 'b'];
> Object.getPrototypeOfNames(Array.prototype)
```

```
[ 'toString', 'join', 'pop', ... ]
> arr.toString()
'a,b'
```

12.5.2 Dispatched method calls

If you look at the method call `arr.toString()` you can see that it actually performs two steps:

1. Dispatch: In the prototype chain of `arr`, retrieve the value of the first property whose name is `toString`.
2. Call: Call the value and set the implicit parameter `this` to the *receiver* `arr` of the method invocation.

You can make the two steps explicit by using the `call()` method of functions:

```
> var func = arr.toString; // dispatch
> func.call(arr) // direct call, providing a value for `this`
'a,b'
```

12.5.3 Direct method calls

There are two ways to make direct method calls in JavaScript:

- `Function.prototype.call(thisValue, arg0?, arg1?, ...)`
- `Function.prototype.apply(thisValue, argArray)`

Both method `call` and method `apply` are invoked on functions. They are different from normal function calls in that you specify a value for `this`. `call` provides the arguments of the method call via individual parameters, `apply` provides them via an Array.

With a dispatched method call, the receiver plays two roles: It is used to find the method and it is an implicit parameter. A problem with the first role is that a method must be in the prototype chain of an object if you want to invoke it. With a direct method call, the method can come from anywhere. That allows you to borrow a method from another object. For example, you can borrow `Object.prototype.toString` and thus apply the original, un-overridden implementation of `toString` to an Array `arr`:

```
> const arr = ['a','b','c'];
> Object.prototype.toString.call(arr)
'[object Array]'
```

The Array version of `toString()` produces a different result:

```
> arr.toString() // dispatched
'a,b,c'
> Array.prototype.toString.call(arr); // direct
'a,b,c'
```

Methods that work with a variety of objects (not just with instances of “their” constructor) are called *generic*. *Speaking JavaScript* has [a list](#) of all methods that are generic. The list includes most Array methods and all methods of `Object.prototype` (which have to work with all objects and are thus implicitly generic).

12.5.4 Use cases for direct method calls

This section covers use cases for direct method calls. Each time, I'll first describe the use case in ES5 and then how it changes with ES6 (where you'll rarely need direct method calls).

12.5.4.1 ES5: Provide parameters to a method via an Array

Some functions accept multiple values, but only one value per parameter. What if you want to pass the values via an Array?

For example, `push()` lets you destructively append several values to an Array:

```
> var arr = ['a', 'b'];
> arr.push('c', 'd')
4
> arr
[ 'a', 'b', 'c', 'd' ]
```

But you can't destructively append a whole Array. You can work around that limitation by using `apply()`:

```
> var arr = ['a', 'b'];
> Array.prototype.push.apply(arr, ['c', 'd'])
4
> arr
[ 'a', 'b', 'c', 'd' ]
```

Similarly, `Math.max()` and `Math.min()` only work for single values:

```
> Math.max(-1, 7, 2)
7
```

With `apply()`, you can use them for Arrays:

```
> Math.max.apply(null, [-1, 7, 2])
7
```

12.5.4.2 ES6: The spread operator (...) mostly replaces apply()

Making a direct method call via `apply()` only because you want to turn an Array into arguments is clumsy, which is why ECMAScript 6 has the spread operator (...) for this. It provides this functionality even in dispatched method calls.

```
> Math.max(...[-1, 7, 2])
7
```

Another example:

```
> const arr = ['a', 'b'];
> arr.push(...['c', 'd'])
4
> arr
[ 'a', 'b', 'c', 'd' ]
```

As a bonus, spread also works with the `new` operator:

```
> new Date(...[2011, 11, 24])
Sat Dec 24 2011 00:00:00 GMT+0100 (CET)
```

Note that `apply()` can't be used with `new` – the above feat can only be achieved via a [complicated work-around](#) in ECMAScript 5.

12.5.4.3 ES5: Convert an Array-like object to an Array

Some objects in JavaScript are *Array-like*, they are almost Arrays, but don't have any of the Array methods. Let's look at two examples.

First, the special variable `arguments` of functions is Array-like. It has a `length` and indexed access to elements.

```
> var args = function () { return arguments }('a', 'b');
> args.length
2
> args[0]
'a'
```

But `arguments` isn't an instance of `Array` and does not have the method `forEach()`.

```
> args instanceof Array
false
> args.forEach
undefined
```

Second, the DOM method `document.querySelectorAll()` returns an instance of `NodeList`.

```
> document.querySelectorAll('a[href]') instanceof NodeList
true
> document.querySelectorAll('a[href]').forEach // no Array methods!
undefined
```

Thus, for many complex operations, you need to convert Array-like objects to Arrays first. That is achieved via `Array.prototype.slice()`. This method copies the elements of its receiver into a new Array:

```
> var arr = ['a', 'b'];
> arr.slice()
[ 'a', 'b' ]
> arr.slice() === arr
false
```

If you call `slice()` directly, you can convert a `NodeList` to an Array:

```
var domLinks = document.querySelectorAll('a[href]');
var links = Array.prototype.slice.call(domLinks);
links.forEach(function (link) {
  console.log(link);
});
```

And you can convert `arguments` to an Array:

```
function format(pattern) {
  // params start at arguments[1], skipping `pattern`
  var params = Array.prototype.slice.call(arguments, 1);
  return params;
}
console.log(format('a', 'b', 'c')); // ['b', 'c']
```

12.5.4.4 ES6: Array-like objects are less burdensome

On one hand, ECMAScript 6 has `Array.from()`, a simpler way of converting Array-like objects to Arrays:

```
const domLinks = document.querySelectorAll('a[href]');
const links = Array.from(domLinks);
links.forEach(function (link) {
  console.log(link);
});
```

On the other hand, you won't need the Array-like arguments, because ECMAScript 6 has *rest parameters* (declared via a triple dot):

```
function format(pattern, ...params) {
  return params;
}
console.log(format('a', 'b', 'c')); // ['b', 'c']
```

12.5.4.5 ES5: Using `hasOwnProperty()` safely

`obj.hasOwnProperty('prop')` tells you whether `obj` has the *own* (non-inherited) property `prop`.

```
> var obj = { prop: 123 };
> obj.hasOwnProperty('prop')
true
> 'toString' in obj // inherited
true
> obj.hasOwnProperty('toString') // own
false
```

However, calling `hasOwnProperty` via dispatch can cease to work properly if `Object.prototype.hasOwnProperty` is overridden.

```
> var obj1 = { hasOwnProperty: 123 };
> obj1.hasOwnProperty('toString')
TypeError: Property 'hasOwnProperty' is not a function
```

`hasOwnProperty` may also be unavailable via dispatch if `Object.prototype` is not in the prototype chain of an object.

```
> var obj2 = Object.create(null);
> obj2.hasOwnProperty('toString')
TypeError: Object has no method 'hasOwnProperty'
```

In both cases, the solution is to make a direct call to `hasOwnProperty`:

```
> var obj1 = { hasOwnProperty: 123 };
> Object.prototype.hasOwnProperty.call(obj1, 'hasOwnProperty')
true
> var obj2 = Object.create(null);
> Object.prototype.hasOwnProperty.call(obj2, 'toString')
false
```

12.5.4.6 ES6: Less need for `hasOwnProperty()`

`hasOwnProperty()` is mostly used to implement Maps via objects. Thankfully, ECMAScript 6 has a built-in Map data structure, which means that you'll need `hasOwnProperty()` less.

12.5.4.7 ES5: Avoiding intermediate objects

Applying an Array method such as `join()` to a string normally involves two steps:

```
var str = 'abc';
var arr = str.split(''); // step 1
var joined = arr.join('-'); // step 2
console.log(joined); // a-b-c
```

Strings are Array-like and can become the `this` value of generic Array methods. Therefore, a direct call lets you avoid step 1:

```
var str = 'abc';
var joined = Array.prototype.join.call(str, '-');
```

Similarly, you can apply `map()` to a string either after you split it or via a direct method call:

```
> function toUpper(x) { return x.toUpperCase(); }
> 'abc'.split('').map(toUpper)
[ 'A', 'B', 'C' ]
> Array.prototype.map.call('abc', toUpper)
[ 'A', 'B', 'C' ]
```

Note that the direct calls may be more efficient, but they are also much less elegant. Be sure that they are really worth it!

12.5.4.8 ES6: Avoiding intermediate objects

`Array.from()` can convert and map in a single step, if you provide it with a callback as the second argument.

```
> Array.from('abc', ch => ch.toUpperCase())
[ 'A', 'B', 'C' ]
```

As a reminder, the two step solution is:

```
> 'abc'.split('').map(function (x) { return x.toUpperCase(); })
[ 'A', 'B', 'C' ]
```

12.5.5 Abbreviations for `Object.prototype` and `Array.prototype`

You can access the methods of `Object.prototype` via an empty object literal (whose prototype is `Object.prototype`). For example, the following two direct method calls are equivalent:

```
Object.prototype.hasOwnProperty.call(obj, 'propKey')
{}.hasOwnProperty.call(obj, 'propKey')
```

The same trick works for `Array.prototype`:

```
Array.prototype.slice.call(arguments)
[].slice.call(arguments)
```

This pattern has become quite popular. It does not reflect the intention of the author as clearly as the longer version, but it's much less verbose. [Speed-wise](#), there isn't much of a difference between the two versions.

12.6 The `name` property of functions

The `name` property of a function contains its name:

```
> function foo() {}
> foo.name
'foo'
```

This property is useful for debugging (its value shows up in stack traces) and some metaprogramming tasks (picking a function by name etc.).

Prior to ECMAScript 6, this property was already supported by most engines. With ES6, it becomes part of the language standard and is frequently filled in automatically.

12.6.1 Constructs that provide names for functions

The following sections describe how `name` is set up automatically for various programming constructs.

12.6.1.1 Variable declarations and assignments

Functions pick up names if they are created via variable declarations:

```
let func1 = function () {};
console.log(func1.name); // func1

const func2 = function () {};
console.log(func2.name); // func2

var func3 = function () {};
console.log(func3.name); // func3
```

But even with a normal assignment, `name` is set up properly:

```
let func4;
func4 = function () {};
console.log(func4.name); // func4

var func5;
func5 = function () {};
console.log(func5.name); // func5
```

With regard to names, arrow functions are like anonymous function expressions:

```
const func = () => {};
console.log(func.name); // func
```

From now on, whenever you see an anonymous function expression, you can assume that an arrow function works the same way.

12.6.1.2 Default values

If a function is a default value, it gets its name from its variable or parameter:

```
let [func1 = function () {}] = [];
console.log(func1.name); // func1

let { f2: func2 = function () {} } = {};
console.log(func2.name); // func2

function g(func3 = function () {}) {
  return func3.name;
}
console.log(g()); // func3
```

12.6.1.3 Named function definitions

Function declarations and function expression are *function definitions*. This scenario has been supported for a long time: a function definition with a name passes it on to the `name` property.

For example, a function declaration:


```
function foo() {}
console.log(foo.name); // foo
```

The name of a named function expression also sets up the `name` property.

```
const bar = function baz() {};
console.log(bar.name); // baz
```

Because it comes first, the function expression's name `baz` takes precedence over other names (e.g. the name `bar` provided via the variable declaration):

However, as in ES5, the name of a function expression is only a variable inside the function expression:

```
const bar = function baz() {
  console.log(baz.name); // baz
};
bar();
console.log(baz); // ReferenceError
```

12.6.1.4 Methods in object literals

If a function is the value of a property, it gets its name from that property. It doesn't matter if that happens via a method definition (line A), a traditional property definition (line B), a property definition with a computed property key (line C) or a property value shorthand (line D).

```
function func() {}
let obj = {
  m1() {}, // (A)
  m2: function () {}, // (B)
  ['m' + '3']: function () {}, // (C)
  func, // (D)
};
console.log(obj.m1.name); // m1
console.log(obj.m2.name); // m2
console.log(obj.m3.name); // m3
console.log(obj.func.name); // func
```

The names of getters are prefixed with `'get'`, the names of setters are prefixed with `'set'`:

```
let obj = {
  get foo() {},
  set bar(value) {},
};
let getter = Object.getOwnPropertyDescriptor(obj, 'foo').get;
console.log(getter.name); // 'get foo'

let setter = Object.getOwnPropertyDescriptor(obj, 'bar').set;
console.log(setter.name); // 'set bar'
```

12.6.1.5 Methods in class definitions

The naming of methods in class definitions is similar to object literals:

```
class C {
  m1() {}
  ['m' + '2']() {} // computed property key

  static classMethod() {}
}
console.log(C.prototype.m1.name); // m1
console.log(new C().m1.name); // m1

console.log(C.prototype.m2.name); // m2

console.log(C.classMethod.name); // classMethod
```

Getters and setters again have the name prefixes `'get'` and `'set'`, respectively.

```
class C {
  get foo() {}
  set bar(value) {}
}
let getter = Object.getOwnPropertyDescriptor(C.prototype, 'foo').get;
console.log(getter.name); // 'get foo'

let setter = Object.getOwnPropertyDescriptor(C.prototype, 'bar').set;
console.log(setter.name); // 'set bar'
```

12.6.1.6 Methods whose keys are symbols

In ES6, the key of a method can be a symbol. The `name` property of such a method is still a string:

- If the symbol has a description, the method's name is the description in square brackets.
- Otherwise, the method's name is the empty string (`''`).

```
const key1 = Symbol('description');
const key2 = Symbol();

let obj = {
  [key1]() {},
  [key2]() {},
};
```

```
console.log(obj[key1].name); // '[description]'
console.log(obj[key2].name); // ''
```

12.6.1.7 Class definitions

Remember that class definitions create functions. Those functions also have their `property name` set up correctly:

```
class Foo {}
console.log(Foo.name); // Foo

const Bar = class {};
console.log(Bar.name); // Bar
```

12.6.1.8 Default exports

All of the following statements set `name` to `'default'`:

```
export default function () {}
export default (function () {});

export default class {}
export default (class {});

export default () => {};
```

12.6.1.9 Other programming constructs

- Generator functions and generator methods get their names the same way that normal functions and methods do.
- `new Function()` produces functions whose `name` is `'anonymous'`. [A webkit bug](#) describes why that is necessary on the web.
- `func.bind(...)` produces a function whose `name` is `'bound '+func.name`:

```
function foo(x) {
  return x
}
const bound = foo.bind(undefined, 123);
console.log(bound.name); // 'bound foo'
```

12.6.2 Caveats

12.6.2.1 Caveat: the name of a function is always assigned at creation

Function names are always assigned during creation and never changed later on. That is, JavaScript engines detect the previously mentioned patterns and create functions that start their lives with the correct names. The following code demonstrates that the name of the function created by `functionFactory()` is assigned in line A and not changed by the declaration in line B.

```
function functionFactory() {
  return function () {}; // (A)
}
const foo = functionFactory(); // (B)
console.log(foo.name.length); // 0 (anonymous)
```

One could, in theory, check for each assignment whether the right-hand side evaluates to a function and whether that function doesn't have a name, yet. But that would incur a significant performance penalty.

12.6.2.2 Caveat: minification

Function names are subject to minification, which means that they will usually change in minified code. Depending on what you want to do, you may have to manage function names via strings (which are not minified) or you may have to tell your minifier what names not to minify.

12.6.3 Changing the names of functions

These are the attributes of `property name`:

```
> let func = function () {}
> Object.getOwnPropertyDescriptor(func, 'name')
{ value: 'func',
  writable: false,
  enumerable: false,
  configurable: true }
```

The property not being writable means that you can't change its value via assignment:

```
> func.name = 'foo';
> func.name
'func'
```

The property is, however, configurable, which means that you can change it by re-defining it:

```
> Object.defineProperty(func, 'name', {value: 'foo', configurable: true});
> func.name
'foo'
```

If the property `name` already exists then you can omit the descriptor property `configurable`, because missing descriptor properties mean that the corresponding attributes are not changed.

If the property `name` does not exist yet then the descriptor property `configurable` ensures that `name` remains configurable (the default attribute values are all `false` or `undefined`).

12.6.4 The function property `name` in the spec

- The spec operation `SetFunctionName()` sets up the property `name`. Search for its name in the spec to find out where that happens.
 - The third parameter of that operation specifies a name prefix. It is used for:
 - [Getters and setters](#) (prefixes `'get'` and `'set'`)
 - `Function.prototype.bind()` (prefix `'bound'`)
- Anonymous function expressions not having a property `name` can be seen by looking at [their runtime semantics](#):
 - The names of named function expressions are set up via `SetFunctionName()`. That operation is not invoked for anonymous function expressions.
 - The names of function declarations are set up when entering a scope (they are hoisted!).
- [When an arrow function is created](#), no name is set up, either (`SetFunctionName()` is not invoked).

12.7 FAQ: callable entities

12.7.1 Why are there “fat” arrow functions (`=>`) in ES6, but no “thin” arrow functions (`->`)?

ECMAScript 6 has syntax for functions with a lexical `this`, so-called *arrow functions*. However, it does not have arrow syntax for functions with dynamic `this`. That omission was deliberate; method definitions cover most of the use cases for thin arrows. If you really need dynamic `this`, you can still use a traditional function expression.

12.7.2 How do I determine whether a function was invoked via `new`?

ES6 has a new protocol for subclassing, which is explained in [the chapter on classes](#). Part of that protocol is the *meta-property* `new.target`, which refers to the first element in a chain of constructor calls (similar to `this` in a chain for supermethod calls). It is `undefined` if there is no constructor call. We can use that to enforce that a function must be invoked via `new` or that it must not be invoked via it. This is an example for the latter:

```
function realFunction() {
  if (new.target !== undefined) {
    throw new Error('Can't be invoked via `new`');
  }
  ...
}
```

In ES5, this was usually checked like this:

```
function realFunction() {
  "use strict";
  if (this !== undefined) {
    throw new Error('Can't be invoked via `new`');
  }
  ...
}
```

13. Arrow functions

13.1 Overview

There are two benefits to arrow functions.

First, they are less verbose than traditional function expressions:

```
const arr = [1, 2, 3];
const squares = arr.map(x => x * x);

// Traditional function expression
const squares = arr.map(function (x) { return x * x });
```

Second, their `this` is picked up from surroundings (*lexical*). Therefore, you don't need `bind()` or `that = this`, anymore.

```
function UiComponent() {
  const button = document.getElementById('myButton');
  button.addEventListener('click', () => {
    console.log('CLICK');
    this.handleClick(); // lexical `this`
  });
}
```

The following variables are all lexical inside arrow functions:

- arguments
- super
- this
- new.target

13.2 Traditional functions are bad non-method functions, due to this

In JavaScript, traditional functions can be used as:

1. Non-method functions
2. Methods
3. Constructors

These roles clash: Due to roles 2 and 3, functions always have their own `this`. But that prevents you from accessing the `this` of, e.g., a surrounding method from inside a callback (role 1).

You can see that in the following ES5 code:

```
function Prefixer(prefix) {
  this.prefix = prefix;
}
Prefixer.prototype.prefixArray = function (arr) { // (A)
  'use strict';
  return arr.map(function (x) { // (B)
    // Doesn't work:
    return this.prefix + x; // (C)
  });
};
```

In line C, we'd like to access `this.prefix`, but can't do that because the `this` of the function from line B shadows the `this` of the method from line A. In strict mode, `this` is undefined in non-method functions, which is why we get an error if we use `Prefixer`:

```
> var pre = new Prefixer('Hi ');
> pre.prefixArray(['Joe', 'Alex'])
TypeError: Cannot read property 'prefix' of undefined
```

There are three ways to work around this problem in ECMAScript 5.

13.2.1 Solution 1: that = this

You can assign `this` to a variable that isn't shadowed. That's what's done in line A, below:

```
function Prefixer(prefix) {
  this.prefix = prefix;
}
Prefixer.prototype.prefixArray = function (arr) {
  var that = this; // (A)
  return arr.map(function (x) {
    return that.prefix + x;
  });
};
```

Now `Prefixer` works as expected:

```
> var pre = new Prefixer('Hi ');
> pre.prefixArray(['Joe', 'Alex'])
[ 'Hi Joe', 'Hi Alex' ]
```

13.2.2 Solution 2: specifying a value for this

A few Array methods have an extra parameter for specifying the value that `this` should have when invoking the callback. That's the last parameter in line A, below.

```
function Prefixer(prefix) {
  this.prefix = prefix;
}
Prefixer.prototype.prefixArray = function (arr) {
  return arr.map(function (x) {
    return this.prefix + x;
  }, this); // (A)
};
```

13.2.3 Solution 3: bind(this)

You can use the method `bind()` to convert a function whose `this` is determined by how it is called (via `call()`, a function call, a method call, etc.) to a function whose `this` is always the same fixed value. That's what we are doing in line A, below.

```
function Prefixer(prefix) {
  this.prefix = prefix;
}
Prefixer.prototype.prefixArray = function (arr) {
  return arr.map(function (x) {
    return this.prefix + x;
  }).bind(this); // (A)
};
```

13.2.4 ECMAScript 6 solution: arrow functions

Arrow functions are basically solution 3, with a more convenient syntax. With an arrow function, the code looks as follows.

```
function Prefixer(prefix) {
  this.prefix = prefix;
}
Prefixer.prototype.prefixArray = function (arr) {
  return arr.map((x) => {
    return this.prefix + x;
  });
};
```

To fully ES6-ify the code, you'd use a class and a more compact variant of arrow functions:

```
class Prefixer {
  constructor(prefix) {
    this.prefix = prefix;
  }
  prefixArray(arr) {
    return arr.map(x => this.prefix + x); // (A)
  }
}
```

In line A we save a few characters by tweaking two parts of the arrow function:

- If there is only one parameter and that parameter is an identifier then the parentheses can be omitted.
- An expression following the arrow leads to that expression being returned.

In the code, you can also see that the methods `constructor` and `prefixArray` are defined using new, more compact ES6 syntax that also works in object literals.

13.3 Arrow function syntax

The “fat” arrow `=>` (as opposed to the thin arrow `-->`) was chosen to be compatible with CoffeeScript, whose fat arrow functions are very similar.

Specifying parameters:

```
() => { ... } // no parameter
x => { ... } // one parameter, an identifier
(x, y) => { ... } // several parameters
```

Specifying a body:

```
x => { return x * x; } // block
x => x * x // expression, equivalent to previous line
```

The statement block behaves like a normal function body. For example, you need `return` to give back a value. With an expression body, the expression is always implicitly returned.

Note how much an arrow function with an expression body can reduce verbosity. Compare:

```
const squares = [1, 2, 3].map(function (x) { return x * x; });
const squares = [1, 2, 3].map(x => x * x);
```

13.3.1 Omitting parentheses around single parameters

Omitting the parentheses around the parameters is only possible if they consist of a single identifier:

```
> [1,2,3].map(x => 2 * x)
[ 2, 4, 6 ]
```

As soon as there is anything else, you have to type the parentheses, even if there is only a single parameter. For example, you need parens if you destructure a single parameter:

```
> [[1,2], [3,4]].map(([a,b]) => a + b)
[ 3, 7 ]
```

And you need parens if a single parameter has a default value (`undefined` triggers the default value!):

```
> [1, undefined, 3].map((x='yes') => x)
[ 1, 'yes', 3 ]
```

13.4 Lexical variables

13.4.1 Sources of variable values: static versus dynamic

The following are two ways in which a variable can receive its value.

First, statically (lexically): Its value is determined by the structure of the program; it receives its value from a surrounding scope. For example:

```
const x = 123;

function foo(y) {
  return x; // value received statically
}
```

Second, dynamically: It receives its value via a function call. For example:

```
function bar(arg) {
  return arg; // value received dynamically
}
```

13.4.2 Variables that are lexical in arrow functions

The source of `this` is an important distinguishing aspect of arrow functions:

- Traditional functions have a *dynamic* `this`; its value is determined by how they are called.
- Arrow functions have a *lexical* `this`; its value is determined by the surrounding scope.

The [complete list](#) of variables whose values are determined lexically is:

- arguments
- super
- this
- new.target

13.5 Syntax pitfalls

There are a few syntax-related details that can sometimes trip you up.

13.5.1 Arrow functions bind very loosely

Syntactically, arrow functions bind very loosely. The reason is that you want every expression that can appear in an expression body to “stick together”; it should bind more tightly than the arrow function:

```
const f = x => (x % 2) === 0 ? x : 0;
```

As a consequence, you often have to wrap arrow functions in parentheses if they appear somewhere else. For example:

```
console.log(typeof () => {}); // SyntaxError
console.log(typeof (() => {})); // OK
```

On the flip side, you can use `typeof` as an expression body without putting it in parens:

```
const f = x => typeof x;
```

13.5.2 No line break after arrow function parameters

ES6 forbids a line break between the parameter definition and the arrow of an arrow function:

```
const func1 = (x, y) // SyntaxError
=> {
  return x + y;
};
const func2 = (x, y) => // OK
{
  return x + y;
};
const func3 = (x, y) => { // OK
  return x + y;
};

const func4 = (x, y) // SyntaxError
=> x + y;
const func5 = (x, y) => // OK
x + y;
```

Line breaks *inside* parameter definitions are OK:

```
const func6 = ( // OK
  x,
  y
) => {
  return x + y;
};
```

The rationale for this restriction is that it keeps the options open w.r.t. to “headless” arrow functions in the future: if there are zero parameters, you’d be able to omit the parentheses.

13.5.3 You can’t use statements as expression bodies

13.5.3.1 Expressions versus statements

Quick review ([consult “Speaking JavaScript” for more information](#)):

Expressions produce (are evaluated to) values. Examples:

```
3 + 4
foo(7)
'abc'.length
```

Statements do things. Examples:

```
while (true) { ... }
return 123;
```

Most expressions¹ can be used as statements, simply by mentioning them in statement positions:

```
function bar() {
  3 + 4;
  foo(7);
  'abc'.length;
}
```

13.5.3.2 The bodies of arrow functions

If an expression is the body of an arrow function, you don't need braces:

```
asyncFunc.then(x => console.log(x));
```

However, statements have to be put in braces:

```
asyncFunc.catch(x => { throw x });
```

13.5.4 Returning object literals

Having a block body in addition to an expression body means that if you want the expression body to be an object literal, you have to put it in parentheses.

The body of this arrow function is a block with the label `bar` and the expression statement `123`.

```
const f = x => { bar: 123 }
```

The body of this arrow function is an expression, an object literal:

```
const f = x => ({ bar: 123 })
```

13.6 Immediately-invoked arrow functions

Remember [Immediately Invoked Function Expressions \(IIFEs\)](#)? They look as follows and are used to simulate block-scoping and value-returning blocks in ECMAScript 5:

```
(function () { // open IIFE
  // inside IIFE
})(); // close IIFE
```

You can save a few characters if you use an Immediately Invoked Arrow Function (IIAF):

```
((() => {
  return 123
})());
```

Similarly to IIFEs, you should terminate IIAFs with semicolons (or use [an equivalent measure](#)), to avoid two consecutive IIAFs being interpreted as a function call (the first one as the function, the second one as the parameter).

Even if the IIAF has a block body, you must wrap it in parentheses, because it can't be (directly) function-called, due to how loosely it binds. Note that the parentheses must be around the arrow function. With IIFEs you have a choice: you can either put the parentheses around the whole statement or just around the function expression.

As mentioned in the previous section, arrow functions binding loosely is useful for expression bodies, where you want this expression:

```
const value = () => foo()
```

to be interpreted as:

```
const value = () => (foo())
```

and not as:

```
const value = (() => foo)()
```

[A section in the chapter on callable entities](#) has more information on using IIFEs and IIAFs in ES6.

13.7 Arrow functions vs. `bind()`

ES6 arrow functions are often a compelling alternative to `Function.prototype.bind()`.

13.7.1 Extracting methods

If an extracted method is to work as a callback, you must specify a fixed `this`, otherwise it will be invoked as a function (and `this` will be `undefined` or the global object). For example:

```
obj.on('anEvent', console.log.bind(console))
```

An alternative is to use an arrow function:

```
obj.on('anEvent', x => console.log(x))
```

13.7.2 `this` via parameters

The following code demonstrates a neat trick: For some methods, you don't need `bind()` for a callback, because they let you specify the value of `this`, via an additional parameter. `filter()` is one such method:

```
const as = new Set([1, 2, 3]);
const bs = new Set([3, 2, 4]);
const intersection = [...as].filter(bs.has, bs);
// [2, 3]
```

However, this code is easier to understand if you use an arrow function:

```
const as = new Set([1, 2, 3]);
const bs = new Set([3, 2, 4]);
const intersection = [...as].filter(a => bs.has(a));
// [2, 3]
```

13.7.3 Partial evaluation

`bind()` enables you to do [partial evaluation](#), you can create new functions by filling in parameters of an existing function:

```
function add(x, y) {
  return x + y;
}
const plus1 = add.bind(undefined, 1);
```

Again, I find an arrow function easier to understand:

```
const plus1 = y => add(1, y);
```

13.8 Arrow functions versus normal functions

An arrow function is different from a normal function in only two ways:

- The following constructs are lexical: `arguments`, `super`, `this`, `new.target`
- It can't be used as a constructor: Normal functions support `new` via the internal method `[[Construct]]` and the property `prototype`. Arrow functions have neither, which is why `new (() => {})` throws an error.

Apart from that, there are no observable differences between an arrow function and a normal function. For example, `typeof` and `instanceof` produce the same results:

```
> typeof (() => {})
'function'
> () => {} instanceof Function
true

> typeof function () {}
'function'
> function () {} instanceof Function
true
```

Consult [the chapter on callable entities](#) for more information on when to use arrow functions and when to use traditional functions.

14. New OOP features besides classes

Classes (which are explained in [the next chapter](#)) are *the* major new OOP feature in ECMAScript 6. However, it also includes new features for object literals and new utility methods in `Object`. This chapter describes them.

14.1 Overview

14.1.1 New object literal features

Method definitions:

```
const obj = {
  myMethod(x, y) {
    ...
  }
};
```

Property value shorthands:


```
const first = 'Jane';
const last = 'Doe';

const obj = { first, last };
// Same as:
const obj = { first: first, last: last };
```

Computed property keys:

```
const propKey = 'foo';
const obj = {
  [propKey]: true,
  ['b'+ 'ar']: 123
};
```

This new syntax can also be used for method definitions:

```
const obj = {
  ['h'+ 'ello']() {
    return 'hi';
  }
};
console.log(obj.hello()); // hi
```

The main use case for computed property keys is to make it easy to use symbols as property keys.

14.1.2 New methods in Object

The most important new method of `Object` is `assign()`. Traditionally, this functionality was called `extend()` in the JavaScript world. In contrast to how this classic operation works, `Object.assign()` only considers *own* (non-inherited) properties.

```
const obj = { foo: 123 };
Object.assign(obj, { bar: true });
console.log(JSON.stringify(obj));
// {"foo":123,"bar":true}
```

14.2 New features of object literals

14.2.1 Method definitions

In ECMAScript 5, methods are properties whose values are functions:

```
var obj = {
  myMethod: function (x, y) {
    ...
  }
};
```

In ECMAScript 6, methods are still function-valued properties, but there is now a more compact way of defining them:

```
const obj = {
  myMethod(x, y) {
    ...
  }
};
```

Getters and setters continue to work as they did in ECMAScript 5 (note how syntactically similar they are to method definitions):

```
const obj = {
  get foo() {
    console.log('GET foo');
    return 123;
  },
  set bar(value) {
    console.log('SET bar to '+value);
    // return value is ignored
  }
};
```

Let's use `obj`:

```
> obj.foo
GET foo
123
> obj.bar = true
SET bar to true
true
```

There is also a way to concisely define properties whose values are generator functions:

```
const obj = {
  * myGeneratorMethod() {
    ...
  }
};
```

This code is equivalent to:

```
const obj = {
  myGeneratorMethod: function* () {
    ...
  }
};
```

```
    }
  };
}
```

14.2.2 Property value shorthands

Property value shorthands let you abbreviate the definition of a property in an object literal: If the name of the variable that specifies the property value is also the property key then you can omit the key. This looks as follows.

```
const x = 4;
const y = 1;
const obj = { x, y };
```

The last line is equivalent to:

```
const obj = { x: x, y: y };
```

Property value shorthands work well together with destructuring:

```
const obj = { x: 4, y: 1 };
const {x,y} = obj;
console.log(x); // 4
console.log(y); // 1
```

One use case for property value shorthands are multiple return values (which are explained in [the chapter on destructuring](#)).

14.2.3 Computed property keys

Remember that there are two ways of specifying a key when you set a property.

1. Via a fixed name: `obj.foo = true;`
2. Via an expression: `obj['b'+ 'ar'] = 123;`

In object literals, you only have option #1 in ECMAScript 5. ECMAScript 6 additionally provides option #2:

```
const propKey = 'foo';
const obj = {
  [propKey]: true,
  ['b'+ 'ar']: 123
};
```

This new syntax can also be used for method definitions:

```
const obj = {
  ['h'+ 'ello']() {
    return 'hi';
  }
};
console.log(obj.hello()); // hi
```

The main use case for computed property keys are symbols: you can define a public symbol and use it as a special property key that is always unique. One prominent example is the symbol stored in `Symbol.iterator`. If an object has a method with that key, it becomes *iterable*: The method must return an iterator, which is used by constructs such as the `for-of` loop to iterate over the object. The following code demonstrates how that works.

```
const obj = {
  * [Symbol.iterator]() { // (A)
    yield 'hello';
    yield 'world';
  }
};
for (const x of obj) {
  console.log(x);
}
// Output:
// hello
// world
```

Line A starts a generator method definition with a computed key (the symbol stored in `Symbol.iterator`).

14.3 New methods of Object

14.3.1 Object.assign(target, source_1, source_2, ...)

This method merges the sources into the target: It modifies `target`, first copies all enumerable *own* (non-inherited) properties of `source_1` into it, then all own properties of `source_2`, etc. At the end, it returns the target.

```
const obj = { foo: 123 };
Object.assign(obj, { bar: true });
console.log(JSON.stringify(obj));
// {"foo":123,"bar":true}
```

Let's look more closely at how `Object.assign()` works:

- Both kinds of property keys: `Object.assign()` is aware of both strings and symbols as property keys.

- Only enumerable own properties: `Object.assign()` ignores inherited properties and properties that are not enumerable.
- Reading a value from a source: normal “get” operation (`const value = source[propKey]`). That means that if the source has a getter whose key is `propKey` then it will be invoked. All properties created by `Object.assign()` are data properties, it won’t transfer getters to the target.
- Writing a value to the target: normal “set” operation (`target[propKey] = value`). That means that if the target has a setter whose key is `propKey` then it will be invoked with `value`.

This is how you’d copy all properties (not just enumerable ones), while correctly transferring getters and setters, without invoking setters on the target:

```
function copyAllProperties(target, ...sources) {
  for (const source of sources) {
    for (const key of Reflect.ownKeys(source)) {
      const desc = Object.getOwnPropertyDescriptor(source, key);
      Object.defineProperty(target, key, desc);
    }
  }
  return target;
}
```

14.3.1.1 Caveat: `Object.assign()` doesn’t work well for moving methods

On one hand, you can’t move a method that uses `super`: Such a method has an internal property `[[HomeObject]]` that ties it to the object it was created in. If you move it via `Object.assign()`, it will continue to refer to the super-properties of the original object. Details are explained in [a section in the chapter on classes](#).

On the other hand, enumerability is wrong if you move methods created by an object literal into the prototype of a class. The former methods are all enumerable (otherwise `Object.assign()` wouldn’t see them, anyway), but the prototype only has non-enumerable methods by default.

14.3.1.2 Use cases for `Object.assign()`

Let’s look at a few use cases.

14.3.1.2.1 Adding properties to `this`

You can use `Object.assign()` to add properties to `this` in a constructor:

```
class Point {
  constructor(x, y) {
    Object.assign(this, {x, y});
  }
}
```

14.3.1.2.2 Providing default values for object properties

`Object.assign()` is also useful for filling in defaults for missing properties. In the following example, we have an object `DEFAULTS` with default values for properties and an object `options` with data.

```
const DEFAULTS = {
  logLevel: 0,
  outputFormat: 'html'
};
function processContent(options) {
  options = Object.assign({}, DEFAULTS, options); // (A)
  ...
}
```

In line A, we created a fresh object, copied the defaults into it and then copied `options` into it, overriding the defaults. `Object.assign()` returns the result of these operations, which we assign to `options`.

14.3.1.2.3 Adding methods to objects

Another use case is adding methods to objects:

```
Object.assign(SomeClass.prototype, {
  someMethod(arg1, arg2) {
    ...
  },
  anotherMethod() {
    ...
  }
});
```

You could also manually assign functions, but then you don’t have the nice method definition syntax and need to mention `SomeClass.prototype` each time:

```
SomeClass.prototype.someMethod = function (arg1, arg2) {
  ...
};
SomeClass.prototype.anotherMethod = function () {
  ...
};
```

14.3.1.2.4 Cloning objects

One last use case for `Object.assign()` is a quick way of cloning objects:

```
function clone(orig) {  
  return Object.assign({}, orig);  
}
```

This way of cloning is also somewhat dirty, because it doesn't preserve the property attributes of `orig`. If that is what you need, you have to use [property descriptors](#).

If you want the clone to have the same prototype as the original, you can use `Object.getPrototypeOf()` and `Object.create()`:

```
function clone(orig) {  
  const origProto = Object.getPrototypeOf(orig);  
  return Object.assign(Object.create(origProto), orig);  
}
```

14.3.2 `Object.getOwnPropertySymbols(obj)`

`Object.getOwnPropertySymbols(obj)` retrieves all *own* (non-inherited) symbol-valued property keys of `obj`. It complements `Object.getOwnPropertyNames()`, which retrieves all string-valued own property keys. Consult [a later section](#) for more details on traversing properties.

14.3.3 `Object.is(value1, value2)`

The strict equals operator (`===`) treats two values differently than one might expect.

First, `NaN` is not equal to itself.

```
> NaN === NaN  
false
```

That is unfortunate, because it often prevents us from detecting `NaN`:

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

Second, JavaScript has [two zeros](#), but strict equals treats them as if they were the same value:

```
> -0 === +0  
true
```

Doing this is normally a good thing.

`Object.is()` provides a way of comparing values that is a bit more precise than `===`. It works as follows:

```
> Object.is(NaN, NaN)  
true  
> Object.is(-0, +0)  
false
```

Everything else is compared as with `===`.

14.3.3.1 Using `Object.is()` to find Array elements

If we combine `Object.is()` with the new ES6 Array method `findIndex()`, we can find `NaN` in Arrays:

```
function myIndexOf(arr, elem) {  
  return arr.findIndex(x => Object.is(x, elem));  
}  
  
myIndexOf([0, NaN, 2], NaN); // 1
```

In contrast, `indexOf()` does not handle `NaN` well:

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

14.3.4 `Object.setPrototypeOf(obj, proto)`

This method sets the prototype of `obj` to `proto`. The non-standard way of doing so in ECMAScript 5, that is supported by many engines, is via assigning to [the special property `__proto__`](#). The recommended way of setting the prototype remains the same as in ECMAScript 5: during the creation of an object, via `Object.create()`. That will always be faster than first creating an object and then setting its prototype. Obviously, it doesn't work if you want to change the prototype of an existing object.

14.4 Traversing properties in ES6

14.4.1 Five operations that traverse properties

In ECMAScript 6, the key of a property can be either a string or a symbol. The following are five operations that traverse the property keys of an object `obj`:

- `Object.keys(obj) : Array<string>`
retrieves all string keys of all enumerable *own* (non-inherited) properties.
- `Object.getOwnPropertyNames(obj) : Array<string>`
retrieves all string keys of all own properties.
- `Object.getOwnPropertySymbols(obj) : Array<symbol>`
retrieves all symbol keys of all own properties.
- `Reflect.ownKeys(obj) : Array<string|symbol>`
retrieves all keys of all own properties.
- `for (const key in obj)`
retrieves all string keys of all enumerable properties (inherited and own).

14.4.2 Traversal order of properties

ES6 defines two traversal orders for properties.

Own Property Keys:

- Retrieves the keys of all own properties of an object, in the following order:
 - First, the string keys that are integer indices (what these are is explained in the next section), in ascending numeric order.
 - Then all other string keys, in the order in which they were added to the object.
 - Lastly, all symbol keys, in the order in which they were added to the object.
- Used by: `Object.assign()`, `Object.defineProperties()`, `Object.getOwnPropertyNames()`, `Object.getOwnPropertySymbols()`, `Reflect.ownKeys()`

Enumerable Own Names:

- Retrieves the string keys of all enumerable own properties of an object. The order is not defined by ES6, but it must be the same order in which `for-in` traverses properties.
- Used by: `JSON.parse()`, `JSON.stringify()`, `Object.keys()`

The order in which `for-in` traverses properties is not defined. [Quoting Allen Wirfs-Brock](#):

Historically, the `for-in` order was not defined and there has been variation among browser implementations in the order they produce (and other specifics). ES5 added `Object.keys` and the requirement that it should order the keys identically to `for-in`. During development of both ES5 and ES6, the possibility of defining a specific `for-in` order was considered but not adopted because of web legacy compatibility concerns and uncertainty about the willingness of browsers to make changes in the ordering they currently produce.

14.4.2.1 Integer indices

Many engines treat integer indices specially, even though they are still strings (at least as far as the ES6 spec is concerned). Therefore, it makes sense to treat them as a separate category of keys.

Roughly, an integer index is a string that, if converted to a 53-bit non-negative integer and back is the same value. Therefore:

- `'10'` and `'2'` are integer indices.
- `'02'` is not an integer index. Converting it to an integer and back results in the different string `'2'`.
- `'3.141'` is not an integer index, because 3.141 is not an integer.

In ES6, instances of `String` and Typed Arrays have integer indices. The indices of normal Arrays are a subset of integer indices: they have a smaller range of 32 bits. For more information on Array indices, consult [“Array Indices in Detail”](#) in “Speaking JavaScript”.



Integer indices have a 53-bit range, because that's the largest range of integers that JavaScript can handle. For details, see Sect. [“Safe integers”](#).

14.4.2.2 Example

The following code demonstrates the traversal order “Own Property Keys”:

```
const obj = {
  [Symbol('first')]: true,
  '02': true,
  '10': true,
  '01': true,
  '2': true,
  [Symbol('second')]: true,
};
Reflect.ownKeys(obj);
// [ '2', '10', '02', '01',
//   Symbol('first'), Symbol('second') ]
```

Explanation:

- '2' and '10' are integer indices, come first and are sorted numerically (not in the order in which they were added).
- '02' and '01' are normal string keys, come next and appear in the order in which they were added to `obj`.
- `Symbol('first')` and `Symbol('second')` are symbols and come last, in the order in which they were added to `obj`.

14.4.2.3 Why does the spec standardize in which order property keys are returned?

Answer by [Tab Atkins Jr.](#):

Because, for objects at least, all implementations used approximately the same order (matching the current spec), and lots of code was inadvertently written that depended on that ordering, and would break if you enumerated it in a different order. Since browsers have to implement this particular ordering to be web-compatible, it was specced as a requirement.

There was some discussion about breaking from this in Maps/Sets, but doing so would require us to specify an order that is *impossible* for code to depend on; in other words, we'd have to mandate that the ordering be *random*, not just unspecified. This was deemed too much effort, and creation-order is reasonable valuable (see `OrderedDict` in Python, for example), so it was decided to have Maps and Sets match Objects.

14.4.2.4 The order of properties in the spec

The following parts of the spec are relevant for this section:

- The section on [Array exotic objects](#) has a note on what Array indices are.
- The internal method `[[OwnPropertyKeys]]` is used by `Reflect.ownKeys()` and others.
- The operation `EnumerableOwnNames` is used by `Object.keys()` and others.
- The internal method `[[Enumerate]]` is used by `for-in`.

14.5 Assigning versus defining properties



This section provides background knowledge that is needed in later sections.

There are two similar ways of adding a property `prop` to an object `obj`:

- Assigning: `obj.prop = 123`
- Defining: `Object.defineProperty(obj, 'prop', 123)`

There are three cases in which assigning does not create an own property `prop`, even if it doesn't exist, yet:

1. A read-only property `prop` exists in the prototype chain. Then the assignment causes a `TypeError` in strict mode.
2. A setter for `prop` exists in the prototype chain. Then that setter is called.
3. A getter for `prop` without a setter exists in the prototype chain. Then a `TypeError` is thrown in strict mode. This case is similar to the first one.

None of these cases prevent `Object.defineProperty()` from creating an own property. The next section looks at case #3 in more detail.

14.5.1 Overriding inherited read-only properties

If an object `obj` inherits a property `prop` that is read-only then you can't assign to that property:

```
const proto = Object.defineProperty({}, 'prop', {
  writable: false,
  configurable: true,
  value: 123,
});
const obj = Object.create(proto);
obj.prop = 456;
// TypeError: Cannot assign to read-only property
```

This is similar to how an inherited property works that has a getter, but no setter. It is in line with viewing assignment as changing the value of an inherited property. It does so non-destructively: the original is not modified, but overridden by a newly created own property. Therefore, an inherited read-only property and an inherited setter-less property both prevent changes via assignment. You can, however, force the creation of an own property by defining a property:

```
const proto = Object.defineProperty({}, 'prop', {
  writable: false,
  configurable: true,
  value: 123,
```

```
});
const obj = Object.create(proto);
Object.defineProperty(obj, 'prop', {value: 456});
console.log(obj.prop); // 456
```

14.6 __proto__ in ECMAScript 6

The property `__proto__` (pronounced “dunder proto”) has existed for a while in most JavaScript engines. This section explains how it worked prior to ECMAScript 6 and what changes with ECMAScript 6.

For this section, it helps if you know what prototype chains are. Consult Sect. “[Layer 2: The Prototype Relationship Between Objects](#)” in “Speaking JavaScript”, if necessary.

14.6.1 __proto__ prior to ECMAScript 6

14.6.1.1 Prototypes

Each object in JavaScript starts a chain of one or more objects, a so-called *prototype chain*. Each object points to its successor, its *prototype* via the internal property `[[Prototype]]` (which is `null` if there is no successor). That property is called *internal*, because it only exists in the language specification and cannot be directly accessed from JavaScript. In ECMAScript 5, the standard way of getting the prototype `p` of an object `obj` is:

```
var p = Object.getPrototypeOf(obj);
```

There is no standard way to change the prototype of an existing object, but you can create a new object `obj` that has the given prototype `p`:

```
var obj = Object.create(p);
```

14.6.1.2 __proto__

A long time ago, Firefox got the non-standard property `__proto__`. Other browsers eventually copied that feature, due to its popularity.

Prior to ECMAScript 6, `__proto__` worked in obscure ways:

- You could use it to get or set the prototype of any object:

```
var obj = {};
var p = {};

console.log(obj.__proto__ === p); // false
obj.__proto__ = p;
console.log(obj.__proto__ === p); // true
```

- However, it was never an actual property:

```
> var obj = {};
> '.__proto__' in obj
false
```

14.6.1.3 Subclassing Array via __proto__

The main reason why `__proto__` became popular was because it enabled the only way to create a subclass `MyArray` of `Array` in ES5: `Array` instances were exotic objects that couldn't be created by ordinary constructors. Therefore, the following trick was used:

```
function MyArray() {
  var instance = new Array(); // exotic object
  instance.__proto__ = MyArray.prototype;
  return instance;
}
MyArray.prototype = Object.create(Array.prototype);
MyArray.prototype.customMethod = function (...) { ... };
```

[Subclassing in ES6](#) works differently than in ES5 and supports subclassing builtins out of the box.

14.6.1.4 Why __proto__ is problematic in ES5

The main problem is that `__proto__` mixes two levels: the object level (normal properties, holding data) and the meta level.

If you accidentally use `__proto__` as a normal property (object level!), to store data, you get into trouble, because the two levels clash. The situation is compounded by the fact that you have to abuse objects as maps in ES5, because it has no built-in data structure for that purpose. Maps should be able to hold arbitrary keys, but you can't use the key `'__proto__'` with objects-as-maps.

In theory, one could fix the problem by using a symbol instead of the special name `__proto__`, but keeping meta-operations completely separate (as done via `Object.getPrototypeOf()`) is the best approach.

14.6.2 The two kinds of __proto__ in ECMAScript 6

Because `__proto__` was so widely supported, it was decided that its behavior should be standardized for ECMAScript 6. However, due to its problematic nature, it was added as a deprecated feature. These features reside in [Annex B in the ECMAScript specification](#), which is described as follows:

The ECMAScript language syntax and semantics defined in this annex are required when the ECMAScript host is a web browser. The content of this annex is normative but optional if the ECMAScript host is not a web browser.

JavaScript has several undesirable features that are required by a significant amount of code on the web. Therefore, web browsers must implement them, but other JavaScript engines don't have to.

In order to explain the magic behind `__proto__`, two mechanisms were introduced in ES6:

- A getter and a setter implemented via `Object.prototype.__proto__`.
- In an object literal, you can consider the property key `'__proto__'` a special operator for specifying the prototype of the created objects.

14.6.2.1 `Object.prototype.__proto__`

ECMAScript 6 enables getting and setting the property `__proto__` via a getter and a setter stored in `Object.prototype`. If you were to implement them manually, this is roughly what it would look like:

```
Object.defineProperty(Object.prototype, '__proto__', {
  get() {
    const _thisObj = Object(this);
    return Object.getPrototypeOf(_thisObj);
  },
  set(proto) {
    if (this === undefined || this === null) {
      throw new TypeError();
    }
    if (!isObject(this)) {
      return undefined;
    }
    if (!isObject(proto)) {
      return undefined;
    }
    const status = Reflect.setPrototypeOf(this, proto);
    if (!status) {
      throw new TypeError();
    }
  },
});
function isObject(value) {
  return Object(value) === value;
}
```

The getter and the setter for `__proto__` in the ES6 spec:

- [get `Object.prototype.__proto__`](#)
- [set `Object.prototype.__proto__`](#)

14.6.2.2 The property key `__proto__` as an operator in an object literal

If `__proto__` appears as an unquoted or quoted property key in an object literal, the prototype of the object created by that literal is set to the property value:

```
> Object.getPrototypeOf({ __proto__: null })
null
> Object.getPrototypeOf({ '__proto__': null })
null
```

Using the string value `'__proto__'` as a computed property key does not change the prototype, it creates an own property:

```
> const obj = { ['__proto__']: null };
> Object.getPrototypeOf(obj) === Object.prototype
true
> Object.keys(obj)
[ '__proto__' ]
```

The special property key `'__proto__'` in the ES6 spec:

- [__proto__ Property Names in Object Initializers](#)

14.6.3 Avoiding the magic of `__proto__`

14.6.3.1 Defining (not assigning) `__proto__`

In ECMAScript 6, if you [define](#) the own property `__proto__`, no special functionality is triggered and the getter/setter `Object.prototype.__proto__` is overridden:

```
const obj = {};
Object.defineProperty(obj, '__proto__', { value: 123 })

Object.keys(obj); // [ '__proto__' ]
console.log(obj.__proto__); // 123
```


14.6.3.2 Objects that don't have `Object.prototype` as a prototype

The `__proto__` getter/setter is provided via `Object.prototype`. Therefore, an object without `Object.prototype` in its prototype chain doesn't have the getter/setter, either. In the following code, `dict` is an example of such an object – it does not have a prototype. As a result, `__proto__` now works like any other property:

```
> const dict = Object.create(null);
> ' __proto__ ' in dict
false
> dict.__proto__ = 'abc';
> dict.__proto__
'abc'
```

14.6.3.3 `__proto__` and dict objects

If you want to use an object as a dictionary then it is best if it doesn't have a prototype. That's why prototype-less objects are also called *dict objects*. In ES6, you don't even have to escape the property key `'__proto__'` for dict objects, because it doesn't trigger any special functionality.

`__proto__` as an operator in an object literal lets you create dict objects more concisely:

```
const dictObj = {
  __proto__: null,
  yes: true,
  no: false,
};
```

Note that in ES6, you should normally prefer the built-in data structure `Map` to dict objects, especially if keys are not fixed.

14.6.3.4 `__proto__` and JSON

Prior to ES6, the following could happen in a JavaScript engine:

```
> JSON.parse('{"__proto__": []}') instanceof Array
true
```

With `__proto__` being a getter/setter in ES6, `JSON.parse()` works fine, because it defines properties, it doesn't assign them (if implemented properly, [an older version of V8 did assign](#)).

`JSON.stringify()` isn't affected by `__proto__`, either, because it only considers own properties. Objects that have an own property whose name is `__proto__` work fine:

```
> JSON.stringify({'__proto__': true})
'{"__proto__":true}'
```

14.6.4 Detecting support for ES6-style `__proto__`

Support for ES6-style `__proto__` varies from engine to engine. Consult kangax' ECMAScript 6 compatibility table for information on the status quo:

- `Object.prototype.__proto__`
- `__proto__` in object literals

The following two sections describe how you can programmatically detect whether an engine supports either of the two kinds of `__proto__`.

14.6.4.1 Feature: `__proto__` as getter/setter

A simple check for the getter/setter:

```
var supported = {}.hasOwnProperty.call(Object.prototype, '__proto__');
```

A more sophisticated check:

```
var desc = Object.getOwnPropertyDescriptor(Object.prototype, '__proto__');
var supported = (
  typeof desc.get === 'function' && typeof desc.set === 'function'
);
```

14.6.4.2 Feature: `__proto__` as an operator in an object literal

You can use the following check:

```
var supported = Object.getPrototypeOf({__proto__: null}) === null;
```

14.6.5 `__proto__` is pronounced “dunder proto”

Bracketing names with double underscores is a common practice in Python to avoid name clashes between meta-data (such as `__proto__`) and data (user-defined properties). That practice will never become common in JavaScript, because it now has symbols for this purpose. However, we can look to the Python community for ideas on how to pronounce double underscores.

The following pronunciation has been [suggested](#) by Ned Batchelder:

An awkward thing about programming in Python: there are lots of double underscores. For example, the standard method names beneath the syntactic sugar have names like `__getattr__`, constructors are `__init__`, built-in operators can be overloaded with `__add__`, and so on. [...]

My problem with the double underscore is that it's hard to say. How do you pronounce `__init__`? "underscore underscore init underscore underscore"? "under under init under under"? Just plain "init" seems to leave out something important.

I have a solution: double underscore should be pronounced "dunder". So `__init__` is "dunder init dunder", or just "dunder init".

Thus, `__proto__` is pronounced "dunder proto". The chances for this pronunciation catching on are good, JavaScript creator Brendan Eich uses it.

14.6.6 Recommendations for `__proto__`

It is nice how well ES6 turns `__proto__` from something obscure into something that is easy to understand.

However, I still recommend not to use it. It is effectively a deprecated feature and not part of the core standard. You can't rely on it being there for code that must run on all engines.

More recommendations:

- Use `Object.getPrototypeOf()` to get the prototype of an object.
- Prefer `Object.create()` to create a new object with a given prototype. Avoid `Object.setPrototypeOf()`, which hampers performance on many engines.
- I actually like `__proto__` as an operator in an object literal. It is useful for demonstrating prototypal inheritance and for creating dict objects. However, the previously mentioned caveats do apply.

14.7 Enumerability in ECMAScript 6

Enumerability is an *attribute* of object properties. This section explains how it works in ECMAScript 6. Let's first explore what attributes are.

14.7.1 Property attributes

Each object has zero or more *properties*. Each property has a key and three or more *attributes*, named slots that store the data of the property (in other words, a property is itself much like a JavaScript object or like a record with fields in a database).

ECMAScript 6 supports the following attributes (as does ES5):

- All properties have the attributes:
 - `enumerable`: Setting this attribute to `false` hides the property from some operations.
 - `configurable`: Setting this attribute to `false` prevents several changes to a property (attributes except `value` can't be change, property can't be deleted, etc.).
- Normal properties (data properties, methods) have the attributes:
 - `value`: holds the value of the property.
 - `writable`: controls whether the property's value can be changed.
- Accessors (getters/setters) have the attributes:
 - `get`: holds the getter (a function).
 - `set`: holds the setter (a function).

You can retrieve the attributes of a property via `Object.getOwnPropertyDescriptor()`, which returns the attributes as a JavaScript object:

```
> const obj = { foo: 123 };
> Object.getOwnPropertyDescriptor(obj, 'foo')
{ value: 123,
  writable: true,
  enumerable: true,
  configurable: true }
```

This section explains how the attribute `enumerable` works in ES6. All other attributes and how to change attributes is explained in Sect. "[Property Attributes and Property Descriptors](#)" in "Speaking JavaScript".

14.7.2 Constructs affected by enumerability

ECMAScript 5:

- `for-in` loop: traverses the string keys of own and inherited enumerable properties.
- `Object.keys()`: returns the string keys of enumerable own properties.
- `JSON.stringify()`: only stringifies enumerable own properties with string keys.

ECMAScript 6:

- `Object.assign()`: only copies enumerable own properties (both string keys and symbol keys are considered).

`for-in` is the only built-in operations where enumerability matters for inherited properties. All other operations only work with own properties.

14.7.3 Use cases for enumerability

Unfortunately, enumerability is quite an idiosyncratic feature. This section presents several use cases for it and argues that, apart from protecting legacy code from breaking, its usefulness is limited.

14.7.3.1 Use case: Hiding properties from the `for-in` loop

The `for-in` loop traverses *all* enumerable properties of an object, own and inherited ones. Therefore, the attribute `enumerable` is used to hide properties that should not be traversed. That was the reason for introducing enumerability in ECMAScript 1.

14.7.3.1.1 Non-enumerability in the language

Non-enumerable properties occur in the following locations in the language:

- All prototype properties of built-in classes are non-enumerable:

```
> const desc = Object.getOwnPropertyDescriptor.bind(Object);
> desc(Object.prototype, 'toString').enumerable
false
```

- All prototype properties of classes are non-enumerable:

```
> desc(class {foo() {}}.prototype, 'foo').enumerable
false
```

- In Arrays, `length` is not enumerable, which means that `for-in` only traverses indices. (However, that can easily change if you add a property via assignment, which makes it enumerable.)

```
> desc([], 'length').enumerable
false
> desc(['a'], '0').enumerable
true
```

The main reason for making all of these properties non-enumerable is to hide them (especially the inherited ones) from legacy code that uses the `for-in` loop or `$.extend()` (and similar operations that copy both inherited and own properties; see next section). Both operations should be avoided in ES6. Hiding them ensures that the legacy code doesn't break.

14.7.3.2 Use case: Marking properties as not to be copied

14.7.3.2.1 Historical precedents

When it comes to copying properties, there are two important historical precedents that take enumerability into consideration:

- [Prototype's `Object.extend\(destination, source\)`](#)

```
const obj1 = Object.create({ foo: 123 });
Object.extend({}, obj1); // { foo: 123 }

const obj2 = Object.defineProperty({}, 'foo', {
  value: 123,
  enumerable: false
});
Object.extend({}, obj2) // {}
```

- [jQuery's `\$.extend\(target, source1, source2, ...\)`](#) copies all enumerable own and inherited properties of `source1` etc. into own properties of `target`.

```
const obj1 = Object.create({ foo: 123 });
$.extend({}, obj1); // { foo: 123 }

const obj2 = Object.defineProperty({}, 'foo', {
  value: 123,
  enumerable: false
});
$.extend({}, obj2) // {}
```

Problems with this way of copying properties:

- Turning inherited source properties into own target properties is rarely what you want. That's why enumerability is used to hide inherited properties.
- Which properties to copy and which not often depends on the task at hand, it rarely makes sense to have a single flag for everything. A better choice is to provide the copying operation with a *predicate* (a callback that returns a boolean) that tells it when to consider a property.

The only instance property that is non-enumerable in the standard library is property `length` of Arrays. However, that property only needs to be hidden due to it magically

updating itself via other properties. You can't create that kind of magic property for your own objects (short of using a Proxy).

14.7.3.2.2 ES6: `Object.assign()`

In ES6, `Object.assign(target, source_1, source_2, ...)` can be used to merge the sources into the target. All own enumerable properties of the sources are considered (that is, keys can be either strings or symbols). `Object.assign()` uses a "get" operation to read a value from a source and a "set" operation to write a value to the target.

With regard to enumerability, `Object.assign()` continues the tradition of `Object.extend()` and `$.extend()`. [Quoting Yehuda Katz](#):

Object.assign would pave the cowpath of all of the extend() APIs already in circulation. We thought the precedent of not copying enumerable methods in those cases was enough reason for Object.assign to have this behavior.

In other words: `Object.assign()` was created with an upgrade path from `$.extend()` (and similar) in mind. Its approach is cleaner than `$.extend`'s, because it ignores inherited properties.

14.7.3.3 Marking properties as private

If you make a property non-enumerable, it can't be seen by `Object.keys()` and the `for-in` loop, anymore. With regard to those mechanisms, the property is private.

However, there are several problems with this approach:

- When copying an object, you normally want to copy private properties. That clashes making properties non-enumerable that shouldn't be copied (see previous section).
- The property isn't really private. Getting, setting and several other mechanisms make no distinction between enumerable and non-enumerable properties.
- When working with code either as source or interactively, you can't immediately see whether a property is enumerable or not. A naming convention (such as prefixing property names with an underscore) is easier to discover.
- You can't use enumerability to distinguish between public and private methods, because methods in prototypes are non-enumerable by default.

14.7.3.4 Hiding own properties from `JSON.stringify()`

`JSON.stringify()` does not include properties in its output that are non-enumerable. You can therefore use enumerability to determine which own properties should be exported to JSON. This use case is similar to marking properties as private, the previous use case. But it is also different, because this is more about exporting and slightly different considerations apply. For example: Can an object be completely reconstructed from JSON?

An alternative for specifying how an object should be converted to JSON is to use `toJSON()`:

```
const obj = {
  foo: 123,
  toJSON() {
    return { bar: 456 };
  },
};
JSON.stringify(obj); // '{"bar":456}'
```

I find `toJSON()` cleaner than enumerability for the current use case. It also gives you more control, because you can export properties that don't exist on the object.

14.7.4 Naming inconsistencies

In general, a shorter name means that only enumerable properties are considered:

- `Object.keys()` ignores non-enumerable properties
- `Object.getOwnPropertyNames()` lists all property names

However, `Reflect.ownKeys()` deviates from that rule, it ignores enumerability and returns the keys of all properties. Additionally, starting with ES6, the following distinction is made:

- *Property keys* are either strings or symbols.
- *Property names* are only strings.

Therefore, a better name for `Object.keys()` would now be `Object.names()`.

14.7.5 Looking ahead

It seems to me that enumerability is only suited for hiding properties from the `for-in` loop and `$.extend()` (and similar operations). Both are legacy features, you should avoid them in new code. As for the other use cases:

- I don't think there is a need for a general flag specifying whether or not to copy a property.
- Non-enumerability does not work well as a way to keep properties private.
- The `toJSON()` method is more powerful and explicit than enumerability when it comes to controlling how to convert an object to JSON.

I'm not sure what the best strategy is for enumerability going forward. If, with ES6, we had started to pretend that it didn't exist (except for making prototype properties non-enumerable so that old code doesn't break), we might eventually have been able to deprecate enumerability. However, `Object.assign()` considering enumerability runs counter that strategy (but it does so for a valid reason, backward compatibility).

In my own ES6 code, I'm not using enumerability, except (implicitly) for classes whose `prototype` methods are non-enumerable.

Lastly, when using an interactive command line, I occasionally miss an operation that returns *all* property keys of an object, not just the own ones (`Reflect.ownKeys`). Such an operation would provide a nice overview of the contents of an object.

14.8 Customizing basic language operations via well-known symbols

This section explains how you can customize basic language operations by using the following well-known symbols as property keys:

- `Symbol.hasInstance` (method)
Lets an object `C` customize the behavior of `x instanceof C`.
- `Symbol.toPrimitive` (method)
Lets an object customize how it is converted to a primitive value. This is the first step whenever something is coerced to a primitive type (via operators etc.).
- `Symbol.toStringTag` (string)
Called by `Object.prototype.toString()` to compute the default string description of an object `obj`: `'[object '+obj[Symbol.toStringTag]+'']'`.
- `Symbol.unscopables` (Object)
Lets an object hide some properties from the `with` statement.

14.8.1 Property key `Symbol.hasInstance` (method)

An object `C` can customize the behavior of the `instanceof` operator via a method with the key `Symbol.hasInstance` that has the following signature:

```
[Symbol.hasInstance](potentialInstance : any)
```

`x instanceof C` works as follows in ES6:

- If `C` is not an object, throw a `TypeError`.
- If the method exists, call `C[Symbol.hasInstance](x)`, coerce the result to boolean and return it.
- Otherwise, compute and return the result according to the traditional algorithm (`C` must be callable, `C.prototype` in the prototype chain of `x`, etc.).

14.8.1.1 Uses in the standard library

The only method in the standard library that has this key is:

- `Function.prototype[Symbol.hasInstance]()`

This is the implementation of `instanceof` that all functions (including classes) use by default. [Quoting the spec](#):

This property is non-writable and non-configurable to prevent tampering that could be used to globally expose the target function of a bound function.

The tampering is possible because the traditional `instanceof` algorithm, [OrdinaryHasInstance\(\)](#), applies `instanceof` to the target function if it encounters a bound function.

Given that this property is read-only, you can't use assignment to override it, [as mentioned earlier](#).

14.8.1.2 Example: checking whether a value is an object

As an example, let's implement an object `ReferenceType` whose "instances" are all objects, not just objects that are instances of `Object` (and therefore have `Object.prototype` in their prototype chains).

```
const ReferenceType = {
  [Symbol.hasInstance](value) {
    return (value !== null
      && (typeof value === 'object'
        || typeof value === 'function'));
  }
};
const obj1 = {};
console.log(obj1 instanceof ReferenceType); // true
```

```
console.log(obj1 instanceof ReferenceType); // true

const obj2 = Object.create(null);
console.log(obj2 instanceof Object); // false
console.log(obj2 instanceof ReferenceType); // true
```

14.8.2 Property key `Symbol.toPrimitive` (method)

`Symbol.toPrimitive` lets an object customize how it is *coerced* (converted automatically) to a primitive value.

Many JavaScript operations coerce values to the types that they need.

- The multiplication operator (`*`) coerces its operands to numbers.
- `new Date(year, month, date)` coerces its parameters to numbers.
- `parseInt(string, radix)` coerces its first parameter to a string.

The following are the most common types that values are coerced to:

- **Boolean:** Coercion returns `true` for truthy values, `false` for falsy values. Objects are always truthy (even `new Boolean(false)`).
- **Number:** Coercion converts objects to primitives first. Primitives are then converted to numbers (`null` \rightarrow `0`, `true` \rightarrow `1`, `'123'` \rightarrow `123`, etc.).
- **String:** Coercion converts objects to primitives first. Primitives are then converted to strings (`null` \rightarrow `'null'`, `true` \rightarrow `'true'`, `123` \rightarrow `'123'`, etc.).
- **Object:** The coercion *wraps* primitive values (booleans `b` via `new Boolean(b)`, numbers `n` via `new Number(n)`, etc.).

Thus, for numbers and strings, the first step is to ensure that a value is any kind of primitive. That is handled by the spec-internal operation `ToPrimitive()`, which has three modes:

- **Number:** the caller needs a number.
- **String:** the caller needs a string.
- **Default:** the caller needs either a number or a string.

The default mode is only used by:

- Equality operator (`==`)
- Addition operator (`+`)
- `new Date(value)` (exactly one parameter!)

If the value is a primitive then `ToPrimitive()` is already done. Otherwise, the value is an object `obj`, which is converted to a primitive as follows:

- **Number mode:** Return the result of `obj.valueOf()` if it is primitive. Otherwise, return the result of `obj.toString()` if it is primitive. Otherwise, throw a `TypeError`.
- **String mode:** works like Number mode, but `toString()` is called first, `valueOf()` second.
- **Default mode:** works exactly like Number mode.

This normal algorithm can be overridden by giving an object a method with the following signature:

```
[Symbol.toPrimitive](hint : 'default' | 'string' | 'number')
```

In the standard library, there are two such methods:

- `Symbol.prototype[Symbol.toPrimitive](hint)` prevents `toString()` from being called (which throws an exception).
- `Date.prototype[Symbol.toPrimitive](hint)` This method implements behavior that deviates from the default algorithm. Quoting the specification: “Date objects are unique among built-in ECMAScript object in that they treat ‘default’ as being equivalent to ‘string’. All other built-in ECMAScript objects treat ‘default’ as being equivalent to ‘number’.”

14.8.2.1 Example

The following code demonstrates how coercion affects the object `obj`.

```
const obj = {
  [Symbol.toPrimitive](hint) {
    switch (hint) {
      case 'number':
        return 123;
      case 'string':
        return 'str';
      case 'default':
        return 'default';
      default:
        throw new Error();
    }
  }
};

console.log(2 * obj); // 246
console.log(3 + obj); // '3default'
console.log(obj == 'default'); // true
console.log(String(obj)); // 'str'
```

14.8.3 Property key `Symbol.toStringTag` (string)

In ES5 and earlier, each object had the internal own property `[[Class]]` whose value hinted at its type. You could not access it directly, but its value was part of the string returned by `Object.prototype.toString()`, which is why that method was used for type checks, as an alternative to `typeof`.

In ES6, there is no internal property `[[Class]]`, anymore, and using `Object.prototype.toString()` for type checks is discouraged. In order to ensure the backwards-compatibility of that method, the public property with the key `Symbol.toStringTag` was introduced. You could say that it replaces `[[Class]]`.

`Object.prototype.toString()` now works as follows:

- Convert `this` to an object `obj`.
- Determine the *toString tag* `tst` of `obj`.
- Return `'[object ' + tst + ']'`.

14.8.3.1 Default toString tags

The default values for various kinds of objects are shown in the following table.

| Value | toString tag |
|-----------------------------|--------------|
| undefined | 'Undefined' |
| null | 'Null' |
| An Array object | 'Array' |
| A string object | 'String' |
| arguments | 'Arguments' |
| Something callable | 'Function' |
| An error object | 'Error' |
| A boolean object | 'Boolean' |
| A number object | 'Number' |
| A date object | 'Date' |
| A regular expression object | 'RegExp' |
| (Otherwise) | 'Object' |

Most of the checks in the left column are performed by looking at internal properties. For example, if an object has the internal property `[[Call]]`, it is callable.

The following interaction demonstrates the default toString tags.

```
> Object.prototype.toString.call(null)
'[object Null]'
> Object.prototype.toString.call([])
'[object Array]'
> Object.prototype.toString.call({})
'[object Object]'
> Object.prototype.toString.call(Object.create(null))
'[object Object]'
```

14.8.3.2 Overriding the default toString tag

If an object has an (own or inherited) property whose key is `Symbol.toStringTag` then its value overrides the default toString tag. For example:

```
> ({}).toString()
'[object Object]'
> ({[Symbol.toStringTag]: 'Foo'}).toString()
'[object Foo]'
```

Instances of user-defined classes get the default toString tag (of objects):

```
class Foo { }
console.log(new Foo().toString()); // [object Object]
```

One option for overriding the default is via a getter:

```
class Bar {
  get [Symbol.toStringTag]() {
    return 'Bar';
  }
}
console.log(new Bar().toString()); // [object Bar]
```

In the JavaScript standard library, there are the following custom toString tags. Objects that have no global names are quoted with percent symbols (for example:

`%TypedArray%`).

- Module-like objects:
 - `JSON[Symbol.toStringTag] → 'JSON'`
 - `Math[Symbol.toStringTag] → 'Math'`
- Actual module objects `M`: `M[Symbol.toStringTag] → 'Module'`
- Built-in classes
 - `ArrayBuffer.prototype[Symbol.toStringTag] → 'ArrayBuffer'`
 - `DataView.prototype[Symbol.toStringTag] → 'DataView'`
 - `Map.prototype[Symbol.toStringTag] → 'Map'`
 - `Promise.prototype[Symbol.toStringTag] → 'Promise'`
 - `Set.prototype[Symbol.toStringTag] → 'Set'`
 - `get %TypedArray%.prototype[Symbol.toStringTag] → 'Uint8Array' etc.`

- WeakMap.prototype[Symbol.toStringTag] → 'WeakMap'
- WeakSet.prototype[Symbol.toStringTag] → 'WeakSet'
- Iterators
 - %MapIteratorPrototype[Symbol.toStringTag] → 'Map Iterator'
 - %SetIteratorPrototype[Symbol.toStringTag] → 'Set Iterator'
 - %StringIteratorPrototype[Symbol.toStringTag] → 'String Iterator'
- Miscellaneous
 - Symbol.prototype[Symbol.toStringTag] → 'Symbol'
 - Generator.prototype[Symbol.toStringTag] → 'Generator'
 - GeneratorFunction.prototype[Symbol.toStringTag] → 'GeneratorFunction'

All of the built-in properties whose keys are `Symbol.toStringTag` have the following property descriptor:

```
{
  writable: false,
  enumerable: false,
  configurable: true,
}
```

As mentioned earlier, you can't use assignment to override those properties, because they are read-only.

14.8.4 Property key `Symbol.unscopables` (Object)

`Symbol.unscopables` lets an object hide some properties from the `with` statement.

The reason for doing so is that it allows TC39 to add new methods to `Array.prototype` without breaking old code. Note that current code rarely uses `with`, which is forbidden in strict mode and therefore ES6 modules (which are implicitly in strict mode).

Why would adding methods to `Array.prototype` break code that uses `with` (such as the widely deployed [Ext JS 4.2.1](#))? Take a look at the following code. The existence of a property `Array.prototype.values` breaks `foo()`, if you call it with an `Array`:

```
function foo(values) {
  with (values) {
    console.log(values.length); // abc (*)
  }
}
Array.prototype.values = { length: 'abc' };
foo([]);
```

Inside the `with` statement, all properties of `values` become local variables, shadowing even `values` itself. Therefore, if `values` has a property `values` then the statement in line * logs `values.values.length` and not `values.length`.

`Symbol.unscopables` is used only once in the standard library:

- `Array.prototype[Symbol.unscopables]`
 - Holds an object with the following properties (which are therefore hidden from the `with` statement): `copyWithin`, `entries`, `fill`, `find`, `findIndex`, `keys`, `values`

14.9 FAQ: object literals

14.9.1 Can I use `super` in object literals?

Yes you can! Details are explained in [the chapter on classes](#).

15. Classes

15.1 Overview

A class and a subclass:

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  toString() {
    return `${this.x}, ${this.y}`;
  }
}

class ColorPoint extends Point {
  constructor(x, y, color) {
    super(x, y);
    this.color = color;
  }
  toString() {
    return super.toString() + ' in ' + this.color;
  }
}
```


Using the classes:

```
> const cp = new ColorPoint(25, 8, 'green');
> cp.toString();
'(25, 8) in green'
> cp instanceof ColorPoint
true
> cp instanceof Point
true
```

Under the hood, ES6 classes are not something that is radically new: They mainly provide more convenient syntax to create old-school constructor functions. You can see that if you use `typeof`:

```
> typeof Point
'function'
```

15.2 The essentials

15.2.1 Base classes

A class is defined like this in ECMAScript 6:

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  toString() {
    return `${this.x}, ${this.y}`;
  }
}
```

You use this class just like an ES5 constructor function:

```
> var p = new Point(25, 8);
> p.toString()
'(25, 8)'
```

In fact, the result of a class definition is a function:

```
> typeof Point
'function'
```

However, you can only invoke a class via `new`, not via a function call (the rationale behind this [is explained later](#)):

```
> Point()
TypeError: Classes can't be function-called
```



In the spec, function-calling classes is prevented in [the internal method](#) `[[Call]]` of function objects.

15.2.1.1 No separators between members of class definitions

There is no separating punctuation between the members of a class definition. For example, the members of an object literal are separated by commas, which are illegal at the top levels of class definitions. Semicolons are allowed, but ignored:

```
class MyClass {
  foo() {}
  ; // OK, ignored
  , // SyntaxError
  bar() {}
}
```

Semicolons are allowed in preparation for future syntax which may include semicolon-terminated members. Commas are forbidden to emphasize that class definitions are different from object literals.

15.2.1.2 Class declarations are not hoisted

Function declarations are *hoisted*: When entering a scope, the functions that are declared in it are immediately available – independently of where the declarations happen. That means that you can call a function that is declared later:

```
foo(); // works, because `foo` is hoisted
function foo() {}
```

In contrast, class declarations are not hoisted. Therefore, a class only exists after execution reached its definition and it was evaluated. Accessing it beforehand leads to a `ReferenceError`:

```
new Foo(); // ReferenceError
```

```
class Foo {}
```

The reason for this limitation is that classes can have an `extends` clause whose value is an arbitrary expression. That expression must be evaluated in the proper “location”, its evaluation can’t be hoisted.

Not having hoisting is less limiting than you may think. For example, a function that comes before a class declaration can still refer to that class, but you have to wait until the class declaration has been evaluated before you can call the function.

```
function functionThatUsesBar() {
  new Bar();
}

functionThatUsesBar(); // ReferenceError
class Bar {}
functionThatUsesBar(); // OK
```

15.2.1.3 Class expressions

Similarly to functions, there are two kinds of *class definitions*, two ways to define a class: *class declarations* and *class expressions*.

Similarly to function expressions, class expressions can be anonymous:

```
const MyClass = class {
  ...
};
const inst = new MyClass();
```

Also similarly to function expressions, class expressions can have names that are only visible inside them:

```
const MyClass = class Me {
  getClassName() {
    return Me.name;
  }
};
const inst = new MyClass();

console.log(inst.getClassName()); // Me
console.log(Me.name); // ReferenceError: Me is not defined
```

The last two lines demonstrate that `Me` does not become a variable outside of the class, but can be used inside it.

15.2.2 Inside the body of a class definition

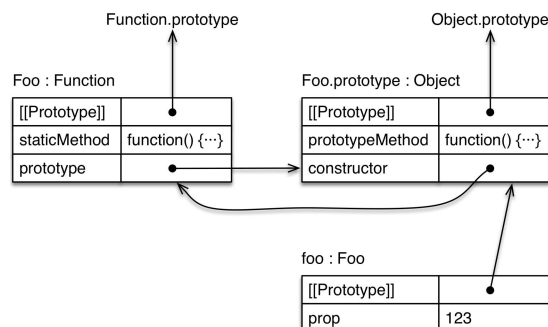
A class body can only contain methods, but not data properties. Prototypes having data properties is generally considered an anti-pattern, so this just enforces a best practice.

15.2.2.1 constructor, static methods, prototype methods

Let’s examine three kinds of methods that you often find in class definitions.

```
class Foo {
  constructor(prop) {
    this.prop = prop;
  }
  staticMethod() {
    return 'classy';
  }
  prototypeMethod() {
    return 'prototypical';
  }
}
const foo = new Foo(123);
```

The object diagram for this class declaration looks as follows. Tip for understanding it: `[[Prototype]]` is an inheritance relationship between objects, while `prototype` is a normal property whose value is an object. The property `prototype` is only special w.r.t. the `new` operator using its value as the prototype for instances it creates.



First, the pseudo-method `constructor`. This method is special, as it defines the function that represents the class:

```
> Foo === Foo.prototype.constructor
true
```

```
> typeof Foo
'function'
```

It is sometimes called a `class constructor`. It has features that normal constructor functions don't have (mainly the ability to constructor-call its superconstructor via `super()`, which is explained later).

Second, static methods. *Static properties* (or *class properties*) are properties of `Foo` itself. If you prefix a method definition with `static`, you create a class method:

```
> typeof Foo.staticMethod
'function'
> Foo.staticMethod()
'classy'
```

Third, prototype methods. The *prototype properties* of `Foo` are the properties of `Foo.prototype`. They are usually methods and inherited by instances of `Foo`.

```
> typeof Foo.prototype.prototypeMethod
'function'
> foo.prototypeMethod()
'prototypical'
```

15.2.2.2 Static data properties

For the sake of finishing ES6 classes in time, they were deliberately designed to be “maximally minimal”. That’s why you can currently only create static methods, getters, and setters, but not static data properties. There is a proposal for adding them to the language. Until that proposal is accepted, there are two work-arounds that you can use.

First, you can manually add a static property:

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
}
Point.ZERO = new Point(0, 0);
```

You could use `Object.defineProperty()` to create a read-only property, but I like the simplicity of an assignment.

Second, you can create a static getter:

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  static get ZERO() {
    return new Point(0, 0);
  }
}
```

In both cases, you get a property `Point.ZERO` that you can read. In the first case, the same instance is returned every time. In the second case, a new instance is returned every time.

15.2.2.3 Getters and setters

The syntax for getters and setters is just like in [ECMAScript 5 object literals](#):

```
class MyClass {
  get prop() {
    return 'getter';
  }
  set prop(value) {
    console.log('setter: '+value);
  }
}
```

You use `MyClass` as follows.

```
> const inst = new MyClass();
> inst.prop = 123;
setter: 123
> inst.prop
'getter'
```

15.2.2.4 Computed method names

You can define the name of a method via an expression, if you put it in square brackets. For example, the following ways of defining `Foo` are all equivalent.

```
class Foo() {
  myMethod() {}
}

class Foo() {
  ['my'+ 'Method']() {}
}

const m = 'myMethod';
class Foo() {
```

```

    [m]() {}
  }
}

```

Several special methods in ECMAScript 6 have keys that are symbols. Computed method names allow you to define such methods. For example, if an object has a method whose key is `Symbol.iterator`, it is *iterable*. That means that its contents can be iterated over by the `for-of` loop and other language mechanisms.

```

class IterableClass {
  [Symbol.iterator]() {
    ...
  }
}

```

15.2.2.5 Generator methods

If you prefix a method definition with an asterisk (*), it becomes a *generator method*. Among other things, a generator is useful for defining the method whose key is `Symbol.iterator`. The following code demonstrates how that works.

```

class IterableArguments {
  constructor(...args) {
    this.args = args;
  }
  * [Symbol.iterator]() {
    for (const arg of this.args) {
      yield arg;
    }
  }
}

for (const x of new IterableArguments('hello', 'world')) {
  console.log(x);
}

// Output:
// hello
// world

```

15.2.3 Subclassing

The `extends` clause lets you create a subclass of an existing constructor (which may or may not have been defined via a class):

```

class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  toString() {
    return `${this.x}, ${this.y}`;
  }
}

class ColorPoint extends Point {
  constructor(x, y, color) {
    super(x, y); // (A)
    this.color = color;
  }
  toString() {
    return super.toString() + ' in ' + this.color; // (B)
  }
}

```

Again, this class is used like you'd expect:

```

> const cp = new ColorPoint(25, 8, 'green');
> cp.toString()
' (25, 8) in green '

> cp instanceof ColorPoint
true
> cp instanceof Point
true

```

There are two kinds of classes:

- `Point` is a *base class*, because it doesn't have an `extends` clause.
- `ColorPoint` is a *derived class*.

There are two ways of using `super`:

- A *class constructor* (the pseudo-method `constructor` in a class definition) uses it like a function call (`super(...)`), in order to make a superconstructor call (line A).
- Method definitions (in object literals or classes, with or without `static`) use it like property references (`super.prop`) or method calls (`super.method(...)`), in order to refer to superproperties (line B).

15.2.3.1 The prototype of a subclass is the superclass

The prototype of a subclass is the superclass in ECMAScript 6:

```

> Object.getPrototypeOf(ColorPoint) === Point
true

```

That means that static properties are inherited:

```

class Foo {
  static classMethod() {
    return 'hello';
  }
}

class Bar extends Foo {
}
Bar.classMethod(); // 'hello'

```

You can even super-call static methods:

```

class Foo {
  static classMethod() {
    return 'hello';
  }
}

class Bar extends Foo {
  static classMethod() {
    return super.classMethod() + ', too';
  }
}
Bar.classMethod(); // 'hello, too'

```

15.2.3.2 Superconstructor calls

In a derived class, you must call `super()` before you can use `this`:

```

class Foo {}

class Bar extends Foo {
  constructor(num) {
    const tmp = num * 2; // OK
    this.num = num; // ReferenceError
    super();
    this.num = num; // OK
  }
}

```

Implicitly leaving a derived constructor without calling `super()` also causes an error:

```

class Foo {}

class Bar extends Foo {
  constructor() {
  }
}

const bar = new Bar(); // ReferenceError

```

15.2.3.3 Overriding the result of a constructor

Just like in ES5, you can override the result of a constructor by explicitly returning an object:

```

class Foo {
  constructor() {
    return Object.create(null);
  }
}

console.log(new Foo() instanceof Foo); // false

```

If you do so, it doesn't matter whether `this` has been initialized or not. In other words: you don't have to call `super()` in a derived constructor if you override the result in this manner.

15.2.3.4 Default constructors for classes

If you don't specify a constructor for a base class, the following definition is used:

```

constructor() {}

```

For derived classes, the following default constructor is used:

```

constructor(...args) {
  super(...args);
}

```

15.2.3.5 Subclassing built-in constructors

In ECMAScript 6, you can finally subclass all built-in constructors (there are [work-arounds for ES5](#), but these have significant limitations).

For example, you can now create your own exception classes (that will inherit the feature of having a stack trace in most engines):

```

class MyError extends Error {
}
throw new MyError('Something happened!');

```

You can also create subclasses of `Array` whose instances properly handle `length`:

```

class Stack extends Array {
  get top() {
    return this[this.length - 1];
  }
}

```

```

}

var stack = new Stack();
stack.push('world');
stack.push('hello');
console.log(stack.top); // hello
console.log(stack.length); // 2

```

Note that subclassing `Array` is usually not the best solution. It's often better to create your own class (whose interface you control) and to delegate to an `Array` in a private property.



Subclassing built-in constructors is something that engines have to support natively, you won't get this feature via transpilers.

15.3 Private data for classes

This section explains four approaches for managing private data for ES6 classes:

1. Keeping private data in the environment of a class `constructor`
2. Marking private properties via a naming convention (e.g. a prefixed underscore)
3. Keeping private data in WeakMaps
4. Using symbols as keys for private properties

Approaches #1 and #2 were already common in ES5, for constructors. Approaches #3 and #4 are new in ES6. Let's implement the same example four times, via each of the approaches.

15.3.1 Private data via constructor environments

Our running example is a class `Countdown` that invokes a callback `action` once a counter (whose initial value is `counter`) reaches zero. The two parameters `action` and `counter` should be stored as private data.

In the first implementation, we store `action` and `counter` in the *environment* of the class constructor. An environment is the internal data structure, in which a JavaScript engine stores the parameters and local variables that come into existence whenever a new scope is entered (e.g. via a function call or a constructor call). This is the code:

```

class Countdown {
  constructor(counter, action) {
    Object.assign(this, {
      dec() {
        if (counter < 1) return;
        counter--;
        if (counter === 0) {
          action();
        }
      }
    });
  }
}

```

Using `Countdown` looks like this:

```

> const c = new Countdown(2, () => console.log('DONE'));
> c.dec();
> c.dec();
DONE

```

Pros:

- The private data is completely safe
- The names of private properties won't clash with the names of other private properties (of superclasses or subclasses).

Cons:

- The code becomes less elegant, because you need to add all methods to the instance, inside the constructor (at least those methods that need access to the private data).
- Due to the instance methods, the code wastes memory. If the methods were prototype methods, they would be shared.

More information on this technique: Sect. "[Private Data in the Environment of a Constructor \(Crockford Privacy Pattern\)](#)" in "Speaking JavaScript".

15.3.2 Private data via a naming convention

The following code keeps private data in properties whose names are marked via a prefixed underscore:

```

class Countdown {
  constructor(counter, action) {
    this._counter = counter;
    this._action = action;
  }
}

```

```

dec() {
  if (this._counter < 1) return;
  this._counter--;
  if (this._counter === 0) {
    this._action();
  }
}
}

```

Pros:

- Code looks nice.
- We can use prototype methods.

Cons:

- Not safe, only a guideline for client code.
- The names of private properties can clash.

15.3.3 Private data via WeakMaps

There is a neat technique involving WeakMaps that combines the advantage of the first approach (safety) with the advantage of the second approach (being able to use prototype methods). This technique is demonstrated in the following code: we use the WeakMaps `_counter` and `_action` to store private data.

```

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);
    if (counter < 1) return;
    counter--;
    _counter.set(this, counter);
    if (counter === 0) {
      _action.get(this)();
    }
  }
}

```

Each of the two WeakMaps `_counter` and `_action` maps objects to their private data. Due to how WeakMaps work that won't prevent objects from being garbage-collected. As long as you keep the WeakMaps hidden from the outside world, the private data is safe.

If you want to be even safer, you can store `WeakMap.prototype.get` and `WeakMap.prototype.set` in variables and invoke those (instead of the methods, dynamically):

```

const set = WeakMap.prototype.set;
...
set.call(_counter, this, counter);
// _counter.set(this, counter);

```

Then your code won't be affected if malicious code replaces those methods with ones that snoop on our private data. However, you are only protected against code that runs after your code. There is nothing you can do if it runs before yours.

Pros:

- We can use prototype methods.
- Safer than a naming convention for property keys.
- The names of private properties can't clash.
- Relatively elegant.

Con:

- Code is not as elegant as a naming convention.

15.3.4 Private data via symbols

Another storage location for private data are properties whose keys are symbols:

```

const _counter = Symbol('counter');
const _action = Symbol('action');
class Countdown {
  constructor(counter, action) {
    this[_counter] = counter;
    this[_action] = action;
  }
  dec() {
    if (this[_counter] < 1) return;
    this[_counter]--;
    if (this[_counter] === 0) {
      this[_action]();
    }
  }
}

```

Each symbol is unique, which is why a symbol-valued property key will never clash with any other property key. Additionally, symbols are somewhat hidden from the outside world, but not completely:

```
const c = new Countdown(2, () => console.log('DONE'));

console.log(Object.keys(c));
// []
console.log(Reflect.ownKeys(c));
// [ Symbol(counter), Symbol(action) ]
```

Pros:

- We can use prototype methods.
- The names of private properties can't clash.

Cons:

- Code is not as elegant as a naming convention.
- Not safe: you can list all property keys (including symbols!) of an object via `Reflect.ownKeys()`.

15.3.5 Further reading

- Sect. “[Keeping Data Private](#)” in “[Speaking JavaScript](#)” (covers ES5 techniques)

15.4 Simple mixins

Subclassing in JavaScript is used for two reasons:

- **Interface inheritance:** Every object that is an instance of a subclass (as tested by `instanceof`) is also an instance of the superclass. The expectation is that subclass instances behave like superclass instances, but may do more.
- **Implementation inheritance:** Superclasses pass on functionality to their subclasses.

The usefulness of classes for implementation inheritance is limited, because they only support single inheritance (a class can have at most one superclass). Therefore, it is impossible to inherit tool methods from multiple sources – they must all come from the superclass.

So how can we solve this problem? Let's explore a solution via an example. Consider a management system for an enterprise where `Employee` is a subclass of `Person`.

```
class Person { ... }
class Employee extends Person { ... }
```

Additionally, there are tool classes for storage and for data validation:

```
class Storage {
  save(database) { ... }
}
class Validation {
  validate(schema) { ... }
}
```

It would be nice if we could include the tool classes like this:

```
// Invented ES6 syntax:
class Employee extends Storage, Validation, Person { ... }
```

That is, we want `Employee` to be a subclass of `Storage` which should be a subclass of `Validation` which should be a subclass of `Person`. `Employee` and `Person` will only be used in one such chain of classes. But `Storage` and `Validation` will be used multiple times. We want them to be templates for classes whose superclasses we fill in. Such templates are called *abstract subclasses* or *mixins*.

One way of implementing a mixin in ES6 is to view it as a function whose input is a superclass and whose output is a subclass extending that superclass:

```
const Storage = Sup => class extends Sup {
  save(database) { ... }
};
const Validation = Sup => class extends Sup {
  validate(schema) { ... }
};
```

Here, we profit from the operand of the `extends` clause not being a fixed identifier, but an arbitrary expression. With these mixins, `Employee` is created like this:

```
class Employee extends Storage(Validation(Person)) { ... }
```

Acknowledgement. The first occurrence of this technique that I'm aware of is [a Gist by Sebastian Markbåge](#).

15.5 The details of classes

What we have seen so far are the essentials of classes. You only need to read on if you are interested how things happen under the hood. Let's start with the syntax of classes.

The following is a slightly modified version of the syntax shown in [Sect. A.4 of the ECMAScript 6 specification](#).

```
ClassDeclaration:
  "class" BindingIdentifier ClassTail
ClassExpression:
  "class" BindingIdentifier? ClassTail

ClassTail:
  ClassHeritage? "{" ClassBody? "}"
ClassHeritage:
  "extends" AssignmentExpression
ClassBody:
  ClassElement+
ClassElement:
  MethodDefinition
  "static" MethodDefinition
  ","

MethodDefinition:
  PropName "(" FormalParams ")" "{" FuncBody "}"
  "*" PropName "(" FormalParams ")" "{" GeneratorBody "}"
  "get" PropName "(" ")" "{" FuncBody "}"
  "set" PropName "(" PropSetParams ")" "{" FuncBody "}"

PropertyName:
  LiteralPropertyName
  ComputedPropertyName
LiteralPropertyName:
  IdentifierName /* foo */
  StringLiteral /* "foo" */
  NumericLiteral /* 123.45, 0xFF */
ComputedPropertyName:
  "[" Expression "]"
```

Two observations:

- The value to be extended can be produced by an arbitrary expression. Which means that you'll be able to write code such as the following:

```
class Foo extends combine(MyMixin, MySuperClass) {}
```

- Semicolons are allowed between methods.

15.5.1 Various checks

- Error checks: the class name cannot be `eval` or arguments; duplicate class element names are not allowed; the name `constructor` can only be used for a normal method, not for a getter, a setter or a generator method.
- Classes can't be function-called. They throw a `TypeError` if they are.
- Prototype methods cannot be used as constructors:

```
class C {
  m() {}
}
new C.prototype.m(); // TypeError
```

15.5.2 Attributes of properties

Class declarations create (mutable) let bindings. The following table describes the attributes of properties related to a given class `Foo`:

| | writable | enumerable | configurable |
|--|----------|------------|--------------|
| Static properties <code>Foo.*</code> | true | false | true |
| <code>Foo.prototype</code> | false | false | false |
| <code>Foo.prototype.constructor</code> | false | false | true |
| Prototype properties | | | |
| <code>Foo.prototype.*</code> | true | false | true |

Notes:

- Many properties are writable, to allow for dynamic patching.
- A constructor and the object in its property prototype have an immutable bidirectional link.
- Method definitions in object literals produce enumerable properties.



The properties shown in the table are created in [Sect. "Runtime Semantics: ClassDefinitionEvaluation"](#) in the spec.

15.5.3 Classes have inner names

Classes have lexical inner names, just like named function expressions.

15.5.3.1 The inner names of named function expressions

You may know that named function expressions have lexical inner names:

```

const fac = function me(n) {
  if (n > 0) {
    // Use inner name `me` to
    // refer to function
    return n * me(n-1);
  } else {
    return 1;
  }
};
console.log(fac(3)); // 6

```

The name `me` of the named function expression becomes a lexically bound variable that is unaffected by which variable currently holds the function.

15.5.3.2 The inner names of classes

Interestingly, ES6 classes also have lexical inner names that you can use in methods (constructor methods and regular methods):

```

class C {
  constructor() {
    // Use inner name C to refer to class
    console.log(`constructor: ${C.prop}`);
  }
  logProp() {
    // Use inner name C to refer to class
    console.log(`logProp: ${C.prop}`);
  }
}
C.prop = 'Hi!';

const D = C;
C = null;

// C is not a class, anymore:
new C().logProp();
// TypeError: C is not a function

// But inside the class, the identifier C
// still works
new D().logProp();
// constructor: Hi!
// logProp: Hi!

```

(In the ES6 spec the inner name is set up by [the dynamic semantics of ClassDefinitionEvaluation](#).)

Acknowledgement: Thanks to Michael Ficarra for pointing out that classes have inner names.

15.6 The details of subclassing

In ECMAScript 6, subclassing looks as follows.

```

class Person {
  constructor(name) {
    this.name = name;
  }
  toString() {
    return `Person named ${this.name}`;
  }
  static logNames(persons) {
    for (const person of persons) {
      console.log(person.name);
    }
  }
}

class Employee extends Person {
  constructor(name, title) {
    super(name);
    this.title = title;
  }
  toString() {
    return `${super.toString()} (${this.title})`;
  }
}

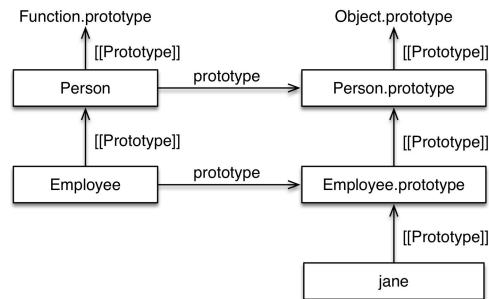
const jane = new Employee('Jane', 'CTO');
console.log(jane.toString()); // Person named Jane (CTO)

```

The next section examines the structure of the objects that were created by the previous example. The section after that examines how `jane` is allocated and initialized.

15.6.1 Prototype chains

The previous example creates the following objects.



Prototype chains are objects linked via the `[[Prototype]]` relationship (which is an inheritance relationship). In the diagram, you can see two prototype chains:

15.6.1.1 Left column: classes (functions)

The prototype of a derived class is the class it extends. The reason for this setup is that you want a subclass to inherit all properties of its superclass:

```
> Employee.logNames === Person.logNames
true
```

The prototype of a base class is `Function.prototype`, which is also the prototype of functions:

```
> const getProto = Object.getPrototypeOf.bind(Object);

> getProto(Person) === Function.prototype
true
> getProto(function () {}) === Function.prototype
true
```

That means that base classes and all their derived classes (their prototypees) are functions. Traditional ES5 functions are essentially base classes.

15.6.1.2 Right column: the prototype chain of the instance

The main purpose of a class is to set up this prototype chain. The prototype chain ends with `Object.prototype` (whose prototype is `null`). That makes `Object` an implicit superclass of every base class (as far as instances and the `instanceof` operator are concerned).

The reason for this setup is that you want the instance prototype of a subclass to inherit all properties of the superclass instance prototype.

As an aside, objects created via object literals also have the prototype

`Object.prototype`:

```
> Object.getPrototypeOf({}) === Object.prototype
true
```

15.6.2 Allocating and initializing instances

The data flow between class constructors is different from the canonical way of subclassing in ES5. Under the hood, it roughly looks as follows.

```
// Base class: this is where the instance is allocated
function Person(name) {
  // Performed before entering this constructor:
  this = Object.create(new.target.prototype);

  this.name = name;
}

function Employee(name, title) {
  // Performed before entering this constructor:
  this = uninitialized;

  this = Reflect.construct(Person, [name], new.target); // (A)
  // super(name);

  this.title = title;
}
Object.setPrototypeOf(Employee, Person);

const jane = Reflect.construct( // (B)
  Employee, ['Jane', 'CTO'],
  Employee);
// const jane = new Employee('Jane', 'CTO')
```

The instance object is created in different locations in ES6 and ES5:

- In ES6, it is created in the base constructor, the last in a chain of constructor calls. The superconstructor is invoked via `super()`, which triggers a constructor call.
- In ES5, it is created in the operand of `new`, the first in a chain of constructor calls. The superconstructor is invoked via a function call.

The previous code uses two new ES6 features:

- `new.target` is an implicit parameter that all functions have. In a chain of constructor calls, its role is similar to `this` in a chain of supermethod calls.
 - If a constructor is directly invoked via `new` (as in line B), the value of `new.target` is that constructor.
 - If a constructor is called via `super()` (as in line A), the value of `new.target` is the `new.target` of the constructor that makes the call.
 - During a normal function call, it is `undefined`. That means that you can use `new.target` to determine whether a function was function-called or constructor-called (via `new`).
 - Inside an arrow function, `new.target` refers to the `new.target` of the surrounding non-arrow function.
- `Reflect.construct()` lets you make constructor calls while specifying `new.target` via the last parameter.

The advantage of this way of subclassing is that it enables normal code to subclass built-in constructors (such as `Error` and `Array`). A later section explains why a different approach was necessary.

As a reminder, here is how you do subclassing in ES5:

```
function Person(name) {
  this.name = name;
}
...

function Employee(name, title) {
  Person.call(this, name);
  this.title = title;
}
Employee.prototype = Object.create(Person.prototype);
Employee.prototype.constructor = Employee;
...
```

15.6.2.1 Safety checks

- `this` originally being uninitialized in derived constructors means that an error is thrown if they access `this` in any way before they have called `super()`.
- Once `this` is initialized, calling `super()` produces a `ReferenceError`. This protects you against calling `super()` twice.
- If a constructor returns implicitly (without a `return` statement), the result is `this`. If `this` is uninitialized, a `ReferenceError` is thrown. This protects you against forgetting to call `super()`.
- If a constructor explicitly returns a non-object (including `undefined` and `null`), the result is `this` (this behavior is required to remain compatible with ES5 and earlier). If `this` is uninitialized, a `TypeError` is thrown.
- If a constructor explicitly returns an object, it is used as its result. Then it doesn't matter whether `this` is initialized or not.

15.6.2.2 The `extends` clause

Let's examine how the `extends` clause influences how a class is set up ([Sect. 14.5.14 of the spec](#)).

The value of an `extends` clause must be "constructible" (invocable via `new`). `null` is allowed, though.

```
class C {
}
```

- Constructor kind: base
- Prototype of `C`: `Function.prototype` (like a normal function)
- Prototype of `C.prototype`: `Object.prototype` (which is also the prototype of objects created via object literals)

```
class C extends B {
}
```

- Constructor kind: derived
- Prototype of `C`: `B`
- Prototype of `C.prototype`: `B.prototype`

```
class C extends Object {
}
```

- Constructor kind: derived
- Prototype of `C`: `Object`
- Prototype of `C.prototype`: `Object.prototype`

Note the following subtle difference with the first case: If there is no `extends` clause, the class is a base class and allocates instances. If a class `extends` `Object`, it is a derived class and `Object` allocates the instances. The resulting instances (including their prototype chains) are the same, but you get there differently.

```
class C extends null {
}
```

- Constructor kind: base (as of ES2016)
- Prototype of `C`: `Function.prototype`

- Prototype of `C.prototype: null`

Such a class lets you avoid `Object.prototype` in the prototype chain.

15.6.3 Why can't you subclass built-in constructors in ES5?

In ECMAScript 5, most built-in constructors can't be subclassed ([several work-arounds exist](#)).

To understand why, let's use the canonical ES5 pattern to subclass `Array`. As we shall soon find out, this doesn't work.

```
function MyArray(len) {
  Array.call(this, len); // (A)
}
MyArray.prototype = Object.create(Array.prototype);
```

Unfortunately, if we instantiate `MyArray`, we find out that it doesn't work properly: The instance property `length` does not change in reaction to us adding `Array` elements:

```
> var myArr = new MyArray(0);
> myArr.length
0
> myArr[0] = 'foo';
> myArr.length
0
```

There are two obstacles that prevent `myArr` from being a proper `Array`.

First obstacle: initialization. The `this` you hand to the constructor `Array` (in line A) is completely ignored. That means you can't use `Array` to set up the instance that was created for `MyArray`.

```
> var a = [];
> var b = Array.call(a, 3);
> a !== b // a is ignored, b is a new object
true
> b.length // set up correctly
3
> a.length // unchanged
0
```

Second obstacle: allocation. The instance objects created by `Array` are *exotic* (a term used by the ECMAScript specification for objects that have features that normal objects don't have): Their property `length` tracks and influences the management of `Array` elements. In general, exotic objects can be created from scratch, but you can't convert an existing normal object into an exotic one. Unfortunately, that is what `Array` would have to do, when called in line A: It would have to turn the normal object created for `MyArray` into an exotic `Array` object.

15.6.3.1 The solution: ES6 subclassing

In ECMAScript 6, subclassing `Array` looks as follows:

```
class MyArray extends Array {
  constructor(len) {
    super(len);
  }
}
```

This works:

```
> const myArr = new MyArray(0);
> myArr.length
0
> myArr[0] = 'foo';
> myArr.length
1
```

Let's examine how the ES6 approach to subclassing removes the previously mentioned obstacles:

- The first obstacle, `Array` not being able to set up an instance, is removed by `Array` returning a fully configured instance. In contrast to ES5, this instance has the prototype of the subclass.
- The second obstacle, subclass constructors not creating exotic instances, is removed by derived classes relying on base classes for allocating instances.

15.6.4 Referring to superproperties in methods

The following ES6 code makes a supermethod call in line B.

```
class Person {
  constructor(name) {
    this.name = name;
  }
  toString() { // (A)
    return `Person named ${this.name}`;
  }
}

class Employee extends Person {
  constructor(name, title) {
```

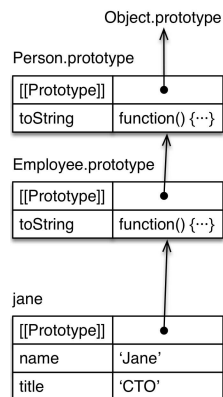
```

    super(name);
    this.title = title;
  }
  toString() {
    return `${super.toString()} (${this.title})`; // (B)
  }
}

const jane = new Employee('Jane', 'CTO');
console.log(jane.toString()); // Person named Jane (CTO)

```

To understand how super-calls work, let's look at the object diagram of `jane`:



In line B, `Employee.prototype.toString` makes a super-call (line B) to the method (starting in line A) that it has overridden. Let's call the object, in which a method is stored, the *home object* of that method. For example, `Employee.prototype` is the home object of `Employee.prototype.toString()`.

The super-call in line B involves three steps:

1. Start your search in the prototype of the home object of the current method.
2. Look for a method whose name is `toString`. That method may be found in the object where the search started or later in the prototype chain.
3. Call that method with the current `this`. The reason for doing so is: the super-called method must be able to access the same instance properties (in our example, the own properties of `jane`).

Note that even if you are only getting (`super.prop`) or setting (`super.prop = 123`) a superproperty (versus making a method call), `this` may still (internally) play a role in step #3, because a getter or a setter may be invoked.

Let's express these steps in three different – but equivalent – ways:

```

// Variation 1: supermethod calls in ES5
var result = Person.prototype.toString.call(this) // steps 1,2,3

// Variation 2: ES5, refactored
var superObject = Person.prototype; // step 1
var superMethod = superObject.toString; // step 2
var result = superMethod.call(this) // step 3

// Variation 3: ES6
var homeObject = Employee.prototype;
var superObject = Object.getPrototypeOf(homeObject); // step 1
var superMethod = superObject.toString; // step 2
var result = superMethod.call(this) // step 3

```

Variation 3 is how ECMAScript 6 handles super-calls. This approach is supported by [two internal bindings](#) that the *environments* of functions have (*environments* provide storage space, so-called *bindings*, for the variables in a scope):

- `[[thisValue]]`: This internal binding also exists in ECMAScript 5 and stores the value of `this`.
- `[[HomeObject]]`: Refers to the home object of the environment's function. Filled in via an internal property `[[HomeObject]]` that all methods have that use `super`. Both the binding and the property are new in ECMAScript 6.



Methods are a special kind of function now

In a class, a method definition that uses `super` creates a special kind of function: It is still a function, but it has the internal property `[[HomeObject]]`. That property is set up by the method definition and can't be changed in JavaScript. Therefore, you can't meaningfully move such a method to a different object. (But maybe it'll be possible in a future version of ECMAScript.)

15.6.4.1 Where can you use `super`?

Referring to superproperties is handy whenever prototype chains are involved, which is why you can use it in method definitions (incl. generator method definitions, getters and

setters) inside object literals and class definitions. The class can be derived or not, the method can be static or not.

Using `super` to refer to a property is not allowed in function declarations, function expressions and generator functions.

15.6.4.2 Pitfall: A method that uses `super` can't be moved

You can't move a method that uses `super`: Such a method has an internal property `[[HomeObject]]` that ties it to the object it was created in. If you move it via an assignment, it will continue to refer to the superproperties of the original object. In future ECMAScript versions, there may be a way to transfer such a method, too.

15.7 The species pattern

One more mechanism of built-in constructors has been made extensible in ECMAScript 6: Sometimes a method creates new instances of its class. If you create a subclass – should the method return an instance of its class or an instance of the subclass? A few built-in ES6 methods let you configure how they create instances via the so-called *species pattern*.

As an example, consider a subclass `SortedArray` of `Array`. If we invoke `map()` on instances of that class, we want it to return instances of `Array`, to avoid unnecessary sorting. By default, `map()` returns instances of the receiver (`this`), but the species patterns lets you change that.

15.7.1 Helper methods for examples

In the following three sections, I'll use two helper functions in the examples:

```
function isObject(value) {
  return (value !== null
    && (typeof value === 'object'
      || typeof value === 'function'));
}

/**
 * Spec-internal operation that determines whether `x`
 * can be used as a constructor.
 */
function isConstructor(x) {
  ...
}
```

15.7.2 The standard species pattern

The standard species pattern is used by `Promise.prototype.then()`, the `filter()` method of Typed Arrays and other operations. It works as follows:

- If `this.constructor[Symbol.species]` exists, use it as a constructor for the new instance.
- Otherwise, use a default constructor (e.g. `Array` for `Arrays`).

Implemented in JavaScript, the pattern would look like this:

```
function SpeciesConstructor(O, defaultConstructor) {
  const C = O.constructor;
  if (C === undefined) {
    return defaultConstructor;
  }
  if (!isObject(C)) {
    throw new TypeError();
  }
  const S = C[Symbol.species];
  if (S === undefined || S === null) {
    return defaultConstructor;
  }
  if (!isConstructor(S)) {
    throw new TypeError();
  }
  return S;
}
```



The standard species pattern is implemented in the spec via the operation `SpeciesConstructor()`.

15.7.3 The species pattern for Arrays

Normal Arrays implement the species pattern slightly differently:

```
function ArraySpeciesCreate(self, length) {
  let C = undefined;
  // If the receiver `self` is an Array,
  // we use the species pattern
  if (Array.isArray(self)) {
    C = self.constructor;
    if (isObject(C)) {
      C = C[Symbol.species];
    }
  }
}
```

```

    }
  }
  // Either `self` is not an Array or the species
  // pattern didn't work out:
  // create and return an Array
  if (C === undefined || C === null) {
    return new Array(length);
  }
  if (!IsConstructor(C)) {
    throw new TypeError();
  }
  return new C(length);
}

```

`Array.prototype.map()` creates the Array it returns via `ArraySpeciesCreate(this, this.length)`.



The species pattern for Arrays is implemented in the spec via the operation `ArraySpeciesCreate()`.

15.7.4 The species pattern in static methods

Promises use a variant of the species pattern for static methods such as

`Promise.all()`:

```

let C = this; // default
if (!isObject(C)) {
  throw new TypeError();
}
// The default can be overridden via the property `C[Symbol.species]`
const S = C[Symbol.species];
if (S !== undefined && S !== null) {
  C = S;
}
if (!IsConstructor(C)) {
  throw new TypeError();
}
const instance = new C(...);

```

15.7.5 Overriding the default species in subclasses

This is the default getter for the property `[Symbol.species]`:

```

static get [Symbol.species]() {
  return this;
}

```

This default getter is implemented by the built-in classes `Array`, `ArrayBuffer`, `Map`, `Promise`, `RegExp`, `Set` and `%TypedArray%`. It is automatically inherited by subclasses of these built-in classes.

There are two ways in which you can override the default species: with a constructor of your choosing or with `null`.

15.7.5.1 Setting the species to a constructor of your choosing

You can override the default species via a static getter (line A):

```

class MyArray1 extends Array {
  static get [Symbol.species]() { // (A)
    return Array;
  }
}

```

As a result, `map()` returns an instance of `Array`:

```

const result1 = new MyArray1().map(x => x);
console.log(result1 instanceof Array); // true

```

If you don't override the default species, `map()` returns an instance of the subclass:

```

class MyArray2 extends Array { }

const result2 = new MyArray2().map(x => x);
console.log(result2 instanceof MyArray2); // true

```

15.7.5.1.1 Specifying the species via a data property

If you don't want to use a static getter, you need to use `Object.defineProperty()`. You can't use assignment, as there is already a property with that key that only has a getter. That means that it is read-only and can't be assigned to.

For example, here we set the species of `MyArray1` to `Array`:

```

Object.defineProperty(
  MyArray1, Symbol.species, {
    value: Array
  });

```

15.7.5.2 Setting the species to null

If you set the species to `null` then the default constructor is used (which one that is depends on which variant of the species pattern is used, consult the previous sections for more information).

```
class MyArray3 extends Array {
  static get [Symbol.species]() {
    return null;
  }
}

const result3 = new MyArray3().map(x => x);
console.log(result3 instanceof Array); // true
```

15.8 The pros and cons of classes

Classes are controversial within the JavaScript community: On one hand, people coming from class-based languages are happy that they don't have to deal with JavaScript's unconventional inheritance mechanisms, anymore. On the other hand, there are many JavaScript programmers who argue that what's complicated about JavaScript is not prototypal inheritance, but constructors.

ES6 classes provide a few clear benefits:

- They are backward-compatible with much of the current code.
- Compared to constructors and constructor inheritance, classes make it easier for beginners to get started.
- Subclassing is supported within the language.
- Built-in constructors are subclassable.
- No library for inheritance is needed, anymore; code will become more portable between frameworks.
- They provide a foundation for advanced features in the future: traits (or mixins), immutable instances, etc.
- They help tools that statically analyze code (IDEs, type checkers, style checkers, etc.).

Let's look at a few common complaints about ES6 classes. You will see me agree with most of them, but I also think that they benefits of classes much outweigh their disadvantages. I'm glad that they are in ES6 and I recommend to use them.

15.8.1 Complaint: ES6 classes obscure the true nature of JavaScript inheritance

Yes, ES6 classes do obscure the true nature of JavaScript inheritance. There is an unfortunate disconnect between what a class looks like (its syntax) and how it behaves (its semantics): It looks like an object, but it is a function. My preference would have been for classes to be *constructor objects*, not constructor functions. I explore that approach in [the Proto.js project](#), via a tiny library (which proves how good a fit this approach is).

However, backwards-compatibility matters, which is why classes being constructor functions also makes sense. That way, ES6 code and ES5 are more interoperable.

The disconnect between syntax and semantics will cause some friction in ES6 and later. But you can lead a comfortable life by simply taking ES6 classes at face value. I don't think the illusion will ever bite you. Newcomers can get started more quickly and later read up on what goes on behind the scenes (after they are more comfortable with the language).

15.8.2 Complaint: Classes provide only single inheritance

Classes only give you single inheritance, which severely limits your freedom of expression w.r.t. object-oriented design. However, the plan has always been for them to be the foundation of a multiple-inheritance mechanism such as traits.



traits.js: traits library for JavaScript

Check out [traits.js](#) if you are interested in how traits work (they are similar to mixins, which you may be familiar with).

Then a class becomes an instantiable entity and a location where you assemble traits. Until that happens, you will need to resort to libraries if you want multiple inheritance.

15.8.3 Complaint: Classes lock you in, due to mandatory `new`

If you want to instantiate a class, you are forced to use `new` in ES6. That means that you can't switch from a class to a factory function without changing the call sites. That is indeed a limitation, but there are two mitigating factors:

- You can override the default result returned by the `new` operator, by returning an object from the `constructor` method of a class.
- Due to its built-in modules and classes, ES6 makes it easier for IDEs to refactor code. Therefore, going from `new` to a function call will be simple. Obviously that

doesn't help you if you don't control the code that calls your code, as is the case for libraries.

Therefore, classes do *somewhat* limit you syntactically, but, once JavaScript has traits, they won't limit you *conceptually* (w.r.t. object-oriented design).

15.9 FAQ: classes

15.9.1 Why can't classes be function-called?

Function-calling classes is currently forbidden. That was done to keep options open for the future, to eventually add a way to handle function calls via classes.

15.9.2 How do I instantiate a class, given an Array of arguments?

What is the analog of `Function.prototype.apply()` for classes? That is, if I have a class `TheClass` and an Array `args` of arguments, how do I instantiate `TheClass`?

One way of doing so is via the spread operator (`...`):

```
function instantiate(TheClass, args) {  
  return new TheClass(...args);  
}
```

Another option is to use `Reflect.construct()`:

```
function instantiate(TheClass, args) {  
  return Reflect.construct(TheClass, args);  
}
```

15.10 What is next for classes?

The design motto for classes was "maximally minimal". Several advanced features were discussed, but ultimately discarded in order to get a design that would be unanimously accepted by TC39.

Upcoming versions of ECMAScript can now extend this minimal design – classes will provide a foundation for features such as traits (or mixins), value objects (where different objects are equal if they have the same content) and const classes (that produce immutable instances).

15.11 Further reading

The following document is an important source of this chapter:

- ["Instantiation Reform: One last time"](#), slides by Allen Wirfs-Brock.

16. Modules

16.1 Overview

JavaScript has had modules for a long time. However, they were implemented via libraries, not built into the language. ES6 is the first time that JavaScript has built-in modules.

ES6 modules are stored in files. There is exactly one module per file and one file per module. You have two ways of exporting things from a module. [These two ways can be mixed](#), but it is usually better to use them separately.

16.1.1 Multiple named exports

There can be multiple *named exports*:

```
//----- lib.js -----  
export const sqrt = Math.sqrt;  
export function square(x) {  
  return x * x;  
}  
export function diag(x, y) {  
  return sqrt(square(x) + square(y));  
}  
  
//----- main.js -----  
import { square, diag } from 'lib';  
console.log(square(11)); // 121  
console.log(diag(4, 3)); // 5
```

You can also import the complete module:

```
//----- main.js -----  
import * as lib from 'lib';  
console.log(lib.square(11)); // 121  
console.log(lib.diag(4, 3)); // 5
```

16.1.2 Single default export

There can be a single *default export*. For example, a function:

```
//----- myFunc.js -----  
export default function () { ... } // no semicolon!  
  
//----- main1.js -----  
import myFunc from 'myFunc';  
myFunc();
```

Or a class:

```
//----- MyClass.js -----  
export default class { ... } // no semicolon!  
  
//----- main2.js -----  
import MyClass from 'MyClass';  
const inst = new MyClass();
```

Note that there is no semicolon at the end if you default-export a function or a class (which are anonymous declarations).

16.1.3 Browsers: scripts versus modules

| | Scripts | Modules |
|--|---------------|------------------------|
| HTML element | <script> | <script type="module"> |
| Default mode | non-strict | strict |
| Top-level variables are | global | local to module |
| Value of <code>this</code> at top level | window | undefined |
| Executed | synchronously | asynchronously |
| Declarative imports (import statement) | no | yes |
| Programmatic imports (Promise-based API) | yes | yes |
| File extension | .js | .js |

16.2 Modules in JavaScript

Even though JavaScript never had built-in modules, the community has converged on a simple style of modules, which is supported by libraries in ES5 and earlier. This style has also been adopted by ES6:

- Each module is a piece of code that is executed once it is loaded.
- In that code, there may be declarations (variable declarations, function declarations, etc.).
 - By default, these declarations stay local to the module.
 - You can mark some of them as exports, then other modules can import them.
- A module can import things from other modules. It refers to those modules via *module specifiers*, strings that are either:
 - Relative paths ('../model/user'): these paths are interpreted relative to the location of the importing module. The file extension .js can usually be omitted.
 - Absolute paths ('/lib/js/helpers'): point directly to the file of the module to be imported.
 - Names ('util'): What modules names refer to has to be configured.
- Modules are singletons. Even if a module is imported multiple times, only a single "instance" of it exists.

This approach to modules avoids global variables, the only things that are global are module specifiers.

16.2.1 ECMAScript 5 module systems

It is impressive how well ES5 module systems work without explicit support from the language. The two most important (and unfortunately incompatible) standards are:

- **CommonJS Modules:** The dominant implementation of this standard is in [Node.js](#) (Node.js modules have a few features that go beyond CommonJS).
Characteristics:
 - Compact syntax
 - Designed for synchronous loading and servers
- **Asynchronous Module Definition (AMD):** The most popular implementation of this standard is [RequireJS](#). Characteristics:
 - Slightly more complicated syntax, enabling AMD to work without eval() (or a compilation step)

- Designed for asynchronous loading and browsers

The above is but a simplified explanation of ES5 modules. If you want more in-depth material, take a look at [“Writing Modular JavaScript With AMD, CommonJS & ES Harmony”](#) by Addy Osmani.

16.2.2 ECMAScript 6 modules

The goal for ECMAScript 6 modules was to create a format that both users of CommonJS and of AMD are happy with:

- Similarly to CommonJS, they have a compact syntax, a preference for single exports and support for cyclic dependencies.
- Similarly to AMD, they have direct support for asynchronous loading and configurable module loading.

Being built into the language allows ES6 modules to go beyond CommonJS and AMD (details are explained later):

- Their syntax is even more compact than CommonJS's.
- Their structure can be statically analyzed (for static checking, optimization, etc.).
- Their support for cyclic dependencies is better than CommonJS's.

The ES6 module standard has two parts:

- Declarative syntax (for importing and exporting)
- Programmatic loader API: to configure how modules are loaded and to conditionally load modules

16.3 The basics of ES6 modules

There are two kinds of exports: named exports (several per module) and default exports (one per module). [As explained later](#), it is possible use both at the same time, but usually best to keep them separate.

16.3.1 Named exports (several per module)

A module can export multiple things by prefixing its declarations with the keyword `export`. These exports are distinguished by their names and are called *named exports*.

```
//----- lib.js -----
export const sqrt = Math.sqrt;
export function square(x) {
  return x * x;
}
export function diag(x, y) {
  return sqrt(square(x) + square(y));
}

//----- main.js -----
import { square, diag } from 'lib';
console.log(square(11)); // 121
console.log(diag(4, 3)); // 5
```

There are other ways to specify named exports (which are explained later), but I find this one quite convenient: simply write your code as if there were no outside world, then label everything that you want to export with a keyword.

If you want to, you can also import the whole module and refer to its named exports via property notation:

```
//----- main.js -----
import * as lib from 'lib';
console.log(lib.square(11)); // 121
console.log(lib.diag(4, 3)); // 5
```

The same code in CommonJS syntax: For a while, I tried several clever strategies to be less redundant with my module exports in Node.js. Now I prefer the following simple but slightly verbose style that is reminiscent of the [revealing module pattern](#):

```
//----- lib.js -----
var sqrt = Math.sqrt;
function square(x) {
  return x * x;
}
function diag(x, y) {
  return sqrt(square(x) + square(y));
}
module.exports = {
  sqrt: sqrt,
  square: square,
  diag: diag,
};

//----- main.js -----
var square = require('lib').square;
var diag = require('lib').diag;
console.log(square(11)); // 121
console.log(diag(4, 3)); // 5
```

16.3.2 Default exports (one per module)

Modules that only export single values are very popular in the Node.js community. But they are also common in frontend development where you often have classes for models and components, with one class per module. An ES6 module can pick a *default export*, the main exported value. Default exports are especially easy to import.

The following ECMAScript 6 module “is” a single function:

```
//----- myFunc.js -----
export default function () {} // no semicolon!

//----- main1.js -----
import myFunc from 'myFunc';
myFunc();
```

An ECMAScript 6 module whose default export is a class looks as follows:

```
//----- MyClass.js -----
export default class {} // no semicolon!

//----- main2.js -----
import MyClass from 'MyClass';
const inst = new MyClass();
```

There are two styles of default exports:

1. Labeling declarations
2. Default-exporting values directly

16.3.2.1 Default export style 1: labeling declarations

You can prefix any function declaration (or generator function declaration) or class declaration with the keywords `export default` to make it the default export:

```
export default function foo() {} // no semicolon!
export default class Bar {} // no semicolon!
```

You can also omit the name in this case. That makes default exports the only place where JavaScript has anonymous function declarations and anonymous class declarations:

```
export default function () {} // no semicolon!
export default class {} // no semicolon!
```

16.3.2.1.1 Why anonymous function declarations and not anonymous function expressions?

When you look at the previous two lines of code, you’d expect the operands of `export default` to be expressions. They are only declarations for reasons of consistency: operands can be named declarations, interpreting their anonymous versions as expressions would be confusing (even more so than introducing new kinds of declarations).

If you want the operands to be interpreted as expressions, you need to use parentheses:

```
export default (function () {});
export default (class {});
```

16.3.2.2 Default export style 2: default-exporting values directly

The values are produced via expressions:

```
export default 'abc';
export default foo();
export default /^xyz$/;
export default 5 * 7;
export default { no: false, yes: true };
```

Each of these default exports has the following structure.

```
export default «expression»;
```

That is equivalent to:

```
const __default__ = «expression»;
export { __default__ as default }; // (A)
```

The statement in line A is an *export clause* (which is explained in [a later section](#)).

16.3.2.2.1 Why two default export styles?

The second default export style was introduced because variable declarations can’t be meaningfully turned into default exports if they declare multiple variables:

```
export default const foo = 1, bar = 2, baz = 3; // not legal JavaScript!
```

Which one of the three variables `foo`, `bar` and `baz` would be the default export?

16.3.3 Imports and exports must be at the top level

As explained in more detail later, the structure of ES6 modules is *static*, you can't conditionally import or export things. That brings a variety of benefits.

This restriction is enforced syntactically by only allowing imports and exports at the top level of a module:

```
if (Math.random()) {
  import 'foo'; // SyntaxError
}

// You can't even nest `import` and `export`
// inside a simple block:
{
  import 'foo'; // SyntaxError
}
```

16.3.4 Imports are hoisted

Module imports are hoisted (internally moved to the beginning of the current scope). Therefore, it doesn't matter where you mention them in a module and the following code works without any problems:

```
foo();

import { foo } from 'my_module';
```

16.3.5 Imports are read-only views on exports

The imports of an ES6 module are read-only views on the exported entities. That means that the connections to variables declared inside module bodies remain live, as demonstrated in the following code.

```
//----- lib.js -----
export let counter = 3;
export function incCounter() {
  counter++;
}

//----- main.js -----
import { counter, incCounter } from './lib';

// The imported value `counter` is live
console.log(counter); // 3
incCounter();
console.log(counter); // 4
```

How that works under the hood is explained [in a later section](#).

Imports as views have the following advantages:

- They enable cyclic dependencies, even for unqualified imports (as explained in the next section).
- Qualified and unqualified imports work the same way (they are both indirections).
- You can split code into multiple modules and it will continue to work (as long as you don't try to change the values of imports).

16.3.6 Support for cyclic dependencies

Two modules A and B are [cyclically dependent](#) on each other if both A (possibly indirectly/transitively) imports B and B imports A. If possible, cyclic dependencies should be avoided, they lead to A and B being *tightly coupled* – they can only be used and evolved together.

Why support cyclic dependencies, then? Occasionally, you can't get around them, which is why support for them is an important feature. [A later section](#) has more information.

Let's see how CommonJS and ECMAScript 6 handle cyclic dependencies.

16.3.6.1 Cyclic dependencies in CommonJS

The following CommonJS code correctly handles two modules `a` and `b` cyclically depending on each other.

```
//----- a.js -----
var b = require('b');
function foo() {
  b.bar();
}
exports.foo = foo;

//----- b.js -----
var a = require('a'); // (i)
function bar() {
  if (Math.random()) {
    a.foo(); // (ii)
  }
}
exports.bar = bar;
```

If module `a` is imported first then, in line `i`, module `b` gets `a`'s exports object before the exports are added to it. Therefore, `b` cannot access `a.foo` in its top level, but that

property exists once the execution of `a` is finished. If `bar()` is called afterwards then the method call in line ii works.

As a general rule, keep in mind that with cyclic dependencies, you can't access imports in the body of the module. That is inherent to the phenomenon and doesn't change with ECMAScript 6 modules.

The limitations of the CommonJS approach are:

- Node.js-style single-value exports don't work. There, you export single values instead of objects:

```
module.exports = function () { ... };
```

If module `a` did that then module `b`'s variable `a` would not be updated once the assignment is made. It would continue to refer to the original exports object.

- You can't use named exports directly. That is, module `b` can't import `foo` like this:

```
var foo = require('a').foo;
```

`foo` would simply be undefined. In other words, you have no choice but to refer to `foo` via `a.foo`.

These limitations mean that both exporter and importers must be aware of cyclic dependencies and support them explicitly.

16.3.6.2 Cyclic dependencies in ECMAScript 6

ES6 modules support cyclic dependencies automatically. That is, they do not have the two limitations of CommonJS modules that were mentioned in the previous section: default exports work, as do unqualified named imports (lines i and iii in the following example). Therefore, you can implement modules that cyclically depend on each other as follows.

```
//----- a.js -----
import {bar} from 'b'; // (i)
export function foo() {
  bar(); // (ii)
}

//----- b.js -----
import {foo} from 'a'; // (iii)
export function bar() {
  if (Math.random()) {
    foo(); // (iv)
  }
}
```

This code works, because, as explained in the previous section, imports are views on exports. That means that even unqualified imports (such as `bar` in line ii and `foo` in line iv) are indirections that refer to the original data. Thus, in the face of cyclic dependencies, it doesn't matter whether you access a named export via an unqualified import or via its module: There is an indirection involved in either case and it always works.

16.4 Importing and exporting in detail

16.4.1 Importing styles

ECMAScript 6 provides several styles of importing²:

- Default import:

```
import localName from 'src/my_lib';
```

- Namespace import: imports the module as an object (with one property per named export).

```
import * as my_lib from 'src/my_lib';
```

- Named imports:

```
import { name1, name2 } from 'src/my_lib';
```

You can rename named imports:

```
// Renaming: import `name1` as `localName1`
import { name1 as localName1, name2 } from 'src/my_lib';

// Renaming: import the default export as `foo`
import { default as foo } from 'src/my_lib';
```

- Empty import: only loads the module, doesn't import anything. The first such import in a program executes the body of the module.

```
import 'src/my_lib';
```

There are only two ways to combine these styles and the order in which they appear is fixed; the default export always comes first.

- Combining a default import with a namespace import:

```
import theDefault, * as my_lib from 'src/my_lib';
```

- Combining a default import with named imports

```
import theDefault, { name1, name2 } from 'src/my_lib';
```

16.4.2 Named exporting styles: inline versus clause

There are [two ways](#) in which you can export named things inside modules.

On one hand, you can mark declarations with the keyword `export`.

```
export var myVar1 = ...;
export let myVar2 = ...;
export const MY_CONST = ...;

export function myFunc() {
  ...
}
export function* myGeneratorFunc() {
  ...
}
export class MyClass {
  ...
}
```

On the other hand, you can list everything you want to export at the end of the module (which is similar in style to the revealing module pattern).

```
const MY_CONST = ...;
function myFunc() {
  ...
}

export { MY_CONST, myFunc };
```

You can also export things under different names:

```
export { MY_CONST as FOO, myFunc };
```

16.4.3 Re-exporting

Re-exporting means adding another module's exports to those of the current module. You can either add all of the other module's exports:

```
export * from 'src/other_module';
```

Default exports are ignored³ by `export *`.

Or you can be more selective (optionally while renaming):

```
export { foo, bar } from 'src/other_module';

// Renaming: export other_module's foo as myFoo
export { foo as myFoo, bar } from 'src/other_module';
```

16.4.3.1 Making a re-export the default export

The following statement makes the default export of another module `foo` the default export of the current module:

```
export { default } from 'foo';
```

The following statement makes the named export `myFunc` of module `foo` the default export of the current module:

```
export { myFunc as default } from 'foo';
```

16.4.4 All exporting styles

ECMAScript 6 provides several styles of exporting⁴:

- Re-exporting:
 - Re-export everything (except for the default export):

```
export * from 'src/other_module';
```

- Re-export via a clause:

```
export { foo as myFoo, bar } from 'src/other_module';

export { default } from 'src/other_module';
export { default as foo } from 'src/other_module';
export { foo as default } from 'src/other_module';
```

- Named exporting via a clause:

```
export { MY_CONST as FOO, myFunc };
export { foo as default };
```

- Inline named exports:

- Variable declarations:

```
export var foo;
export let foo;
export const foo;
```

- Function declarations:

```
export function myFunc() {}
export function* myGenFunc() {}
```

- Class declarations:

```
export class MyClass() {}
```

- Default export:

- Function declarations (can be anonymous here):

```
export default function myFunc() {}
export default function () {}

export default function* myGenFunc() {}
export default function* () {}
```

- Class declarations (can be anonymous here):

```
export default class MyClass() {}
export default class () {}
```

- Expressions: export values. Note the semicolons at the end.

```
export default foo;
export default 'Hello world!';
export default 3 * 7;
export default (function () {});
```

16.4.5 Having both named exports and a default export in a module

The following pattern is surprisingly common in JavaScript: A library is a single function, but additional services are provided via properties of that function. Examples include jQuery and Underscore.js. The following is a sketch of Underscore as a CommonJS module:

```
//----- underscore.js -----
var _ = function (obj) {
  ...
};
var each = _.each = _.forEach =
  function (obj, iterator, context) {
    ...
  };
module.exports = _;

//----- main.js -----
var _ = require('underscore');
var each = _.each;
```

With ES6 glasses, the function `_` is the default export, while `each` and `forEach` are named exports. As it turns out, you can actually have named exports and a default export at the same time. As an example, the previous CommonJS module, rewritten as an ES6 module, looks like this:

```
//----- underscore.js -----
export default function (obj) {
  ...
}
export function each(obj, iterator, context) {
  ...
}
export { each as forEach };

//----- main.js -----
import _, { each } from 'underscore';
...
```

Note that the CommonJS version and the ECMAScript 6 version are only roughly similar. The latter has a flat structure, whereas the former is nested.

16.4.5.1 Recommendation: avoid mixing default exports and named exports

I generally recommend to keep the two kinds of exporting separate: per module, either only have a default export or only have named exports.

However, that is not a very strong recommendation; it occasionally may make sense to mix the two kinds. One example is a module that default-exports an entity. For unit tests, one could additionally make some of the internals available via named exports.

16.4.5.2 The default export is just another named export

The default export is actually just a named export with the special name `default`. That is, the following two statements are equivalent:

```
import { default as foo } from 'lib';
import foo from 'lib';
```

Similarly, the following two modules have the same default export:

```
//----- module1.js -----  
export default function foo() {} // function declaration!  
  
//----- module2.js -----  
function foo() {}  
export { foo as default };
```

16.4.5.3 default: OK as export name, but not as variable name

You can't use reserved words (such as `default` and `new`) as variable names, but you can use them as names for exports (you can also use them as property names in ECMAScript 5). If you want to directly import such named exports, you have to rename them to proper variable names.

That means that `default` can only appear on the left-hand side of a renaming import:

```
import { default as foo } from 'some_module';
```

And it can only appear on the right-hand side of a renaming export:

```
export { foo as default };
```

In re-exporting, both sides of the `as` are export names:

```
export { myFunc as default } from 'foo';  
export { default as otherFunc } from 'foo';  
  
// The following two statements are equivalent:  
export { default } from 'foo';  
export { default as default } from 'foo';
```

16.5 The ECMAScript 6 module loader API

In addition to the declarative syntax for working with modules, there is also a programmatic API. It allows you to:

- Programmatically work with modules
- Configure module loading



The module loader API is not part of the ES6 standard

It will be specified in a separate document, the "JavaScript Loader Standard", that will be evolved more dynamically than the language specification. [The repository for that document](#) states:

[The JavaScript Loader Standard] consolidates work on the ECMAScript module loading semantics with the integration points of Web browsers, as well as Node.js.



The module loader API is work in progress

As you can see in [the repository of the JavaScript Loader Standard](#), the module loader API is still work in progress. Everything you read about it in this book is tentative. To get an impression of what the API may look like, you can take a look at [the ES6 Module Loader Polyfill](#) on GitHub.

16.5.1 Loaders

Loaders handle resolving *module specifiers* (the string IDs at the end of `import-from`), loading modules, etc. Their constructor is `Reflect.Loader`. Each platform keeps a default instance in the global variable `System` (the *system loader*), which implements its specific style of module loading.

16.5.2 Loader method: importing modules

You can programmatically import a module, via an API based on [Promises](#):

```
System.import('some_module')  
  .then(some_module => {  
    // Use some_module  
  })  
  .catch(error => {  
    ...  
  });
```

`System.import()` enables you to:

- Use modules inside `<script>` elements (where module syntax is not supported, consult [the section on modules versus scripts](#) for details).
- Load modules conditionally.

`System.import()` retrieves a single module, you can use `Promise.all()` to import several modules:

```
Promise.all([
  'module1', 'module2', 'module3'
].map(x => System.import(x)))
.then([module1, module2, module3]) => {
  // Use module1, module2, module3
};
```

16.5.3 More loader methods

Loaders have more methods. Three important ones are:

- `System.module(source, options?)`
evaluates the JavaScript code in `source` to a module (which is delivered asynchronously via a Promise).
- `System.set(name, module)`
is for registering a module (e.g. one you have created via `System.module()`).
- `System.define(name, source, options?)`
both evaluates the module code in `source` and registers the result.

16.5.4 Configuring module loading

The module loader API will have various hooks for configuring the loading process. Use cases include:

1. Lint modules on import (e.g. via JSLint or JSHint).
2. Automatically translate modules on import (they could contain CoffeeScript or TypeScript code).
3. Use legacy modules (AMD, Node.js).

Configurable module loading is an area where Node.js and CommonJS are limited.

16.6 Using ES6 modules in browsers

Let's look at how ES6 modules are supported in browsers.



Support for ES6 modules in browsers is work in progress

Similarly to module loading, other aspects of support for modules in browsers are still being worked on. Everything you read here may change.

16.6.1 Browsers: asynchronous modules versus synchronous scripts

In browsers, there are two different kinds of entities: scripts and modules. They have slightly different syntax and work differently.

This is an overview of the differences, details are explained later:

| | Scripts | Modules |
|--|-----------------------------|---|
| HTML element | <code><script></code> | <code><script type="module"></code> |
| Default mode | non-strict | strict |
| Top-level variables are | global | local to module |
| Value of <code>this</code> at top level | window | undefined |
| Executed | synchronously | asynchronously |
| Declarative imports (import statement) | no | yes |
| Programmatic imports (Promise-based API) | yes | yes |
| File extension | .js | .js |

16.6.1.1 Scripts

Scripts are the traditional browser way to embed JavaScript and to refer to external JavaScript files. Scripts have an [internet media type](#) that is used as:

- The content type of JavaScript files delivered via a web server.
- The value of the attribute `type` of `<script>` elements. Note that for HTML5, the recommendation is to omit the `type` attribute in `<script>` elements if they contain or refer to JavaScript.

The following are the most important values:

- `text/javascript`: is a legacy value and used as the default if you omit the `type` attribute in a script tag. It is [the safest choice](#) for Internet Explorer 8 and earlier.
- `application/javascript`: is [recommended](#) for current browsers.

Scripts are normally loaded or executed synchronously. The JavaScript thread stops until the code has been loaded or executed.

16.6.1.2 Modules

To be in line with JavaScript's usual run-to-completion semantics, the body of a module must be executed without interruption. That leaves two options for importing modules:

1. Load modules synchronously, while the body is executed. That is what Node.js does.
2. Load all modules asynchronously, before the body is executed. That is how AMD modules are handled. It is the best option for browsers, because modules are loaded over the internet and execution doesn't have to pause while they are. As an added benefit, this approach allows one to load multiple modules in parallel.

ECMAScript 6 gives you the best of both worlds: The synchronous syntax of Node.js plus the asynchronous loading of AMD. To make both possible, ES6 modules are syntactically less flexible than Node.js modules: Imports and exports must happen at the top level. That means that they can't be conditional, either. This restriction allows an ES6 module loader to analyze statically what modules are imported by a module and load them before executing its body.

The synchronous nature of scripts prevents them from becoming modules. Scripts cannot even import modules declaratively (you have to use the programmatic module loader API if you want to do so).

Modules can be used from browsers via a new variant of the `<script>` element that is completely asynchronous:

```
<script type="module">
  import $ from 'lib/jquery';
  var x = 123;

  // The current scope is not global
  console.log('$' in window); // false
  console.log('x' in window); // false

  // `this` still refers to the global object
  console.log(this === window); // true
</script>
```

As you can see, the element has its own scope and variables "inside" it are local to that scope. Note that module code is implicitly in strict mode. This is great news – no more `'use strict'`.

Similar to normal `<script>` elements, `<script type="module">` can also be used to load external modules. For example, the following tag starts a web application via a `main` module (the attribute name `import` is my invention, it isn't yet clear what name will be used).

```
<script type="module" import="impl/main"></script>
```

The advantage of supporting modules in HTML via a custom `<script>` type is that it is easy to bring that support to older engines via a polyfill (a library). There may or may not eventually be a dedicated element for modules (e.g. `<module>`).

16.6.1.3 Module or script – a matter of context

Whether a file is a module or a script is only determined by how it is imported or loaded. Most modules have either imports or exports and can thus be detected. But if a module has neither then it is indistinguishable from a script. For example:

```
var x = 123;
```

The semantics of this piece of code differs depending on whether it is interpreted as a module or as a script:

- As a module, the variable `x` is created in module scope.
- As a script, the variable `x` becomes a global variable and a property of the global object (`window` in browsers).

More realistic example is a module that installs something, e.g. a polyfill in global variables or a global event listener. Such a module neither imports nor exports anything and is activated via an empty import:

```
import './my_module';
```



Sources of this section

- ["Modules: Status Update"](#), slides by David Herman.
- ["Modules vs Scripts"](#), an email by David Herman.

16.7 Details: imports as views on exports



The code in this section is available [on GitHub](#).

Imports work differently in CommonJS and ES6:

- In CommonJS, imports are copies of exported values.
- In ES6, imports are live read-only views on exported values.

The following sections explain what that means.

16.7.1 In CommonJS, imports are copies of exported values

With CommonJS (Node.js) modules, things work in relatively familiar ways.

If you import a value into a variable, the value is copied twice: once when it is exported (line A) and once it is imported (line B).

```
//----- lib.js -----
var counter = 3;
function incCounter() {
  counter++;
}
module.exports = {
  counter: counter, // (A)
  incCounter: incCounter,
};

//----- main1.js -----
var counter = require('./lib').counter; // (B)
var incCounter = require('./lib').incCounter;

// The imported value is a (disconnected) copy of a copy
console.log(counter); // 3
incCounter();
console.log(counter); // 3

// The imported value can be changed
counter++;
console.log(counter); // 4
```

If you access the value via the exports object, it is still copied once, on export:

```
//----- main2.js -----
var lib = require('./lib');

// The imported value is a (disconnected) copy
console.log(lib.counter); // 3
lib.incCounter();
console.log(lib.counter); // 3

// The imported value can be changed
lib.counter++;
console.log(lib.counter); // 4
```

16.7.2 In ES6, imports are live read-only views on exported values

In contrast to CommonJS, imports are views on exported values. In other words, every import is a live connection to the exported data. Imports are read-only:

- Unqualified imports (`import x from 'foo'`) are like `const`-declared variables.
- The properties of a module object `foo` (`import * as foo from 'foo'`) are like the properties of a [frozen object](#).

The following code demonstrates how imports are like views:

```
//----- lib.js -----
export let counter = 3;
export function incCounter() {
  counter++;
}

//----- main1.js -----
import { counter, incCounter } from './lib';

// The imported value `counter` is live
console.log(counter); // 3
incCounter();
console.log(counter); // 4

// The imported value can't be changed
counter++; // TypeError
```

If you import the module object via the asterisk (*), you get the same results:

```
//----- main2.js -----
import * as lib from './lib';

// The imported value `counter` is live
console.log(lib.counter); // 3
lib.incCounter();
console.log(lib.counter); // 4
```

```
// The imported value can't be changed
lib.counter++; // TypeError
```

Note that while you can't change the values of imports, you can change the objects that they are referring to. For example:

```
//----- lib.js -----
export let obj = {};

//----- main.js -----
import { obj } from './lib';

obj.prop = 123; // OK
obj = {}; // TypeError
```

16.7.2.1 Why a new approach to importing?

Why introduce such a relatively complicated mechanism for importing that deviates from established practices?

- Cyclic dependencies: The main advantage is that it supports [cyclic dependencies](#) even for unqualified imports.
- Qualified and unqualified imports work the same. In CommonJS, they don't: a qualified import provides direct access to a property of a module's export object, an unqualified import is a copy of it.
- You can split code into multiple modules and it will continue to work (as long as you don't try to change the values of imports).
- On the flip side, *module folding*, combining multiple modules into a single module becomes simpler, too.

In my experience, ES6 imports just work, you rarely have to think about what's going on under the hood.

16.7.3 Implementing views

How do imports work as views of exports under the hood? Exports are managed via the data structure *export entry*. All export entries (except those for re-exports) have the following two names:

- Local name: is the name under which the export is stored inside the module.
- Export name: is the name that importing modules need to use to access the export.

After you have imported an entity, that entity is always accessed via a pointer that has the two components *module* and *local name*. In other words, that pointer refers to a *binding* (the storage space of a variable) inside a module.

Let's examine the export names and local names created by various kinds of exporting. The following table ([adapted from the ES6 spec](#)) gives an overview, subsequent sections have more details.

| Statement | Local name | Export name |
|--------------------------------|-------------|-------------|
| export {v}; | 'v' | 'v' |
| export {v as x}; | 'v' | 'x' |
| export const v = 123; | 'v' | 'v' |
| export function f() {} | 'f' | 'f' |
| export default function f() {} | 'f' | 'default' |
| export default function () {} | '*default*' | 'default' |
| export default 123; | '*default*' | 'default' |

16.7.3.1 Export clause

```
function foo() {}
export { foo };
```

- Local name: `foo`
- Export name: `foo`

```
function foo() {}
export { foo as bar };
```

- Local name: `foo`
- Export name: `bar`

16.7.3.2 Inline exports

This is an inline export:

```
export function foo() {}
```

It is equivalent to the following code:

```
function foo() {}  
export { foo };
```

Therefore, we have the following names:

- Local name: `foo`
- Export name: `foo`

16.7.3.3 Default exports

There are two kinds of default exports:

- Default exports of *hoistable declarations* (function declarations, generator function declarations) and class declarations are similar to normal inline exports in that named local entities are created and tagged.
- All other default exports are about exporting the results of expressions.

16.7.3.3.1 Default-exporting expressions

The following code default-exports the result of the expression `123`:

```
export default 123;
```

It is equivalent to:

```
const *default* = 123; // *not* legal JavaScript  
export { *default* as default };
```

If you default-export an expression, you get:

- Local name: `*default*`
- Export name: `default`

The local name was chosen so that it wouldn't clash with any other local name.

Note that a default export still leads to a binding being created. But, due to `*default*` not being a legal identifier, you can't access that binding from inside the module.

16.7.3.3.2 Default-exporting hoistable declarations and class declarations

The following code default-exports a function declaration:

```
export default function foo() {}
```

It is equivalent to:

```
function foo() {}  
export { foo as default };
```

The names are:

- Local name: `foo`
- Export name: `default`

That means that you can change the value of the default export from within the module, by assigning a different value to `foo`.

(Only) for default exports, you can also omit the name of a function declaration:

```
export default function () {}
```

That is equivalent to:

```
function *default* () {} // *not* legal JavaScript  
export { *default* as default };
```

The names are:

- Local name: `*default*`
- Export name: `default`

Default-exporting generator declarations and class declarations works similarly to default-exporting function declarations.

16.7.4 Imports as views in the spec

This section gives pointers into the ECMAScript 2015 (ES6) language specification.

Managing imports:

- `CreateImportBinding()` creates local bindings for imports.
- `GetBindingValue()` is used to access them.
- `ModuleDeclarationInstantiation()` sets up the environment of a module (compare: `FunctionDeclarationInstantiation()`, `BlockDeclarationInstantiation()`).

The export names and local names created by the various kinds of exports are shown in [table 42](#) in the section “[Source Text Module Records](#)”. The section “[Static Semantics: ExportEntries](#)” has more details. You can see that export entries are set up statically (before evaluating the module), evaluating export statements is described in the section “[Runtime Semantics: Evaluation](#)”.

16.8 Design goals for ES6 modules

If you want to make sense of ECMAScript 6 modules, it helps to understand what goals influenced their design. The major ones are:

- Default exports are favored
- Static module structure
- Support for both synchronous and asynchronous loading
- Support for cyclic dependencies between modules

The following subsections explain these goals.

16.8.1 Default exports are favored

The module syntax suggesting that the default export “is” the module may seem a bit strange, but it makes sense if you consider that one major design goal was to make default exports as convenient as possible. Quoting [David Herman](#):

ECMAScript 6 favors the single/default export style, and gives the sweetest syntax to importing the default. Importing named exports can and even should be slightly less concise.

16.8.2 Static module structure

Current JavaScript module formats have a dynamic structure: What is imported and exported can change at runtime. One reason why ES6 introduced its own module format is to enable a static structure, which has several benefits. But before we go into those, let’s examine what the structure being static means.

It means that you can determine imports and exports at compile time (statically) – you only need to look at the source code, you don’t have to execute it. ES6 enforces this syntactically: You can only import and export at the top level (never nested inside a conditional statement). And import and export statements have no dynamic parts (no variables etc. are allowed).

The following are two examples of CommonJS modules that don’t have a static structure. In the first example, you have to run the code to find out what it imports:

```
var my_lib;
if (Math.random()) {
  my_lib = require('foo');
} else {
  my_lib = require('bar');
}
```

In the second example, you have to run the code to find out what it exports:

```
if (Math.random()) {
  exports.baz = ...;
}
```

ECMAScript 6 modules are less flexible and force you to be static. As a result, you get several benefits, which are described next.

16.8.2.1 Benefit: dead code elimination during bundling

In frontend development, modules are usually handled as follows:

- During development, code exists as many, often small, modules.
- For deployment, these modules are *bundled* into a few, relatively large, files.

The reasons for bundling are:

1. Fewer files need to be retrieved in order to load all modules.
2. Compressing the bundled file is slightly more efficient than compressing separate files.
3. During bundling, unused exports can be removed, potentially resulting in significant space savings.

Reason #1 is important for HTTP/1, where the cost for requesting a file is relatively high. That will change with HTTP/2, which is why this reason doesn’t matter there.

Reason #3 will remain compelling. It can only be achieved with a module format that has a static structure.

16.8.2.2 Benefit: compact bundling, no custom bundle format

The module bundler [Rollup](#) proved that ES6 modules can be combined efficiently, because they all fit into a single scope (after renaming variables to eliminate name clashes). This is possible due to two characteristics of ES6 modules:

- Their static structure means that the bundle format does not have to account for conditionally loaded modules (a common technique for doing so is putting module code in functions).
- Imports being read-only views on exports means that you don't have to copy exports, you can refer to them directly.

As an example, consider the following two ES6 modules.

```
// lib.js
export function foo() {}
export function bar() {}

// main.js
import {foo} from './lib.js';
console.log(foo());
```

Rollup can bundle these two ES6 modules into the following single ES6 module (note the eliminated unused export `bar`):

```
function foo() {}

console.log(foo());
```

Another benefit of Rollup's approach is that the bundle does not have a custom format, it is just an ES6 module.

16.8.2.3 Benefit: faster lookup of imports

If you require a library in CommonJS, you get back an object:

```
var lib = require('lib');
lib.someFunc(); // property lookup
```

Thus, accessing a named export via `lib.someFunc` means you have to do a property lookup, which is slow, because it is dynamic.

In contrast, if you import a library in ES6, you statically know its contents and can optimize accesses:

```
import * as lib from 'lib';
lib.someFunc(); // statically resolved
```

16.8.2.4 Benefit: variable checking

With a static module structure, you always statically know which variables are visible at any location inside the module:

- Global variables: increasingly, the only completely global variables will come from the language proper. Everything else will come from modules (including functionality from the standard library and the browser). That is, you statically know all global variables.
- Module imports: You statically know those, too.
- Module-local variables: can be determined by statically examining the module.

This helps tremendously with checking whether a given identifier has been spelled properly. This kind of check is a popular feature of linters such as JSLint and JSHint; in ECMAScript 6, most of it can be performed by JavaScript engines.

Additionally, any access of named imports (such as `lib.foo`) can also be checked statically.

16.8.2.5 Benefit: ready for macros

Macros are still on the roadmap for JavaScript's future. If a JavaScript engine supports macros, you can add new syntax to it via a library. [Sweet.js](#) is an experimental macro system for JavaScript. The following is an example from the Sweet.js website: a macro for classes.

```
// Define the macro
macro class {
  rule {
    $className {
      constructor $params $body
      S($mname $mparams $mbody) ...
    }
  } => {
    function $className $params $body
    S($className.prototype.$mname
      = function $mname $mparams $mbody; ) ...
  }
}

// Use the macro
class Person {
  constructor(name) {
```

```

    this.name = name;
  }
  say(msg) {
    console.log(this.name + " says: " + msg);
  }
}
var bob = new Person("Bob");
bob.say("Macros are sweet!");

```

For macros, a JavaScript engine performs a preprocessing step before compilation: if a sequence of tokens in the token stream produced by the parser matches the pattern part of the macro, it is replaced by tokens generated via the body of macro. The preprocessing step only works if you are able to statically find macro definitions. Therefore, if you want to import macros via modules then they must have a static structure.

16.8.2.6 Benefit: ready for types

Static type checking imposes constraints similar to macros: it can only be done if type definitions can be found statically. Again, types can only be imported from modules if they have a static structure.

Types are appealing because they enable statically typed fast dialects of JavaScript in which performance-critical code can be written. One such dialect is [Low-Level JavaScript](#) (LLJS).

16.8.2.7 Benefit: supporting other languages

If you want to support compiling languages with macros and static types to JavaScript then JavaScript's modules should have a static structure, for the reasons mentioned in the previous two sections.

16.8.2.8 Source of this section

- ["Static module resolution"](#) by David Herman

16.8.3 Support for both synchronous and asynchronous loading

ECMAScript 6 modules must work independently of whether the engine loads modules synchronously (e.g. on servers) or asynchronously (e.g. in browsers). Its syntax is well suited for synchronous loading, asynchronous loading is enabled by its static structure: Because you can statically determine all imports, you can load them before evaluating the body of the module (in a manner reminiscent of AMD modules).

16.8.4 Support for cyclic dependencies between modules

Support for cyclic dependencies was a key goal for ES6 modules. Here is why:

Cyclic dependencies are not inherently evil. Especially for objects, you sometimes even want this kind of dependency. For example, in some trees (such as DOM documents), parents refer to children and children refer back to parents. In libraries, you can usually avoid cyclic dependencies via careful design. In a large system, though, they can happen, especially during refactoring. Then it is very useful if a module system supports them, because the system doesn't break while you are refactoring.

[The Node.js documentation acknowledges the importance of cyclic dependencies](#) and [Rob Sayre provides additional evidence](#):

Data point: I once implemented a system like [ECMAScript 6 modules] for Firefox. I got [asked](#) for cyclic dependency support 3 weeks after shipping.

That system that Alex Fritze invented and I worked on is not perfect, and the syntax isn't very pretty. But [it's still getting used](#) 7 years later, so it must have gotten something right.

16.9 FAQ: modules

16.9.1 Can I use a variable to specify from which module I want to import?

The `import` statement is completely static: its module specifier is always fixed. If you want to dynamically determine what module to load, you need to use [the programmatic loader API](#):

```

const moduleSpecifier = 'module_' + Math.random();
System.import(moduleSpecifier)
  .then(the_module => {
    // Use the_module
  })

```

16.9.2 Can I import a module conditionally or on demand?

Import statements must always be at the top level of modules. That means that you can't nest them inside `if` statements, functions, etc. Therefore, you have to use [the programmatic loader API](#) if you want to load a module conditionally or on demand:

```

if (Math.random()) {
  System.import('some_module')
    .then(some_module => {
      // Use some_module
    })
}

```

16.9.3 Can I use variables in an `import` statement?

No, you can't. Remember – what is imported must not depend on anything that is computed at runtime. Therefore:

```

// Illegal syntax:
import foo from 'some_module'+SUFFIX;

```

16.9.4 Can I use destructuring in an `import` statement?

No you can't. The `import` statement only looks like destructuring, but is completely different (static, imports are views, etc.).

Therefore, you can't do something like this in ES6:

```

// Illegal syntax:
import { foo: { bar } } from 'some_module';

```

16.9.5 Are named exports necessary? Why not default-export objects?

You may be wondering – why do we need named exports if we could simply default-export objects (like in CommonJS)? The answer is that you can't enforce a static structure via objects and lose all of the associated advantages (which are explained in [this chapter](#)).

16.9.6 Can I `eval()` the code of module?

No, you can't. Modules are too high-level a construct for `eval()`. The [module loader API](#) provides the means for creating modules from strings. Syntactically, `eval()` accepts scripts (which don't allow `import` and `export`), not modules.

16.10 Advantages of ECMAScript 6 modules

At first glance, having modules built into ECMAScript 6 may seem like a boring feature – after all, we already have several good module systems. But ECMAScript 6 modules have several new features:

- More compact syntax
- Static module structure (helping with dead code elimination, optimizations, static checking and more)
- Automatic support for cyclic dependencies

ES6 modules will also – hopefully – end the fragmentation between the currently dominant standards CommonJS and AMD. Having a single, native standard for modules means:

- No more UMD ([Universal Module Definition](#)): UMD is a name for patterns that enable the same file to be used by several module systems (e.g. both CommonJS and AMD). Once ES6 is the only module standard, UMD becomes obsolete.
- New browser APIs become modules instead of global variables or properties of `navigator`.
- No more objects-as-namespaces: Objects such as `Math` and `JSON` serve as namespaces for functions in ECMAScript 5. In the future, such functionality can be provided via modules.

16.11 Further reading

- **CommonJS versus ES6:** “[JavaScript Modules](#)” (by [Yehuda Katz](#)) is a quick intro to ECMAScript 6 modules. Especially interesting is a [second page](#) where CommonJS modules are shown side by side with their ECMAScript 6 versions.

IV Collections

17. The `for-of` loop

17.1 Overview

`for-of` is a new loop in ES6 that replaces both `for-in` and `forEach()` and supports the new iteration protocol.

Use it to loop over *iterable* objects (Arrays, strings, Maps, Sets, etc.; see Chap. “[Iterables and iterators](#)”):

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

// Output:
// a
// b
```

`break` and `continue` work inside `for-of` loops:

```
for (const x of ['a', '', 'b']) {
  if (x.length === 0) break;
  console.log(x);
}

// Output:
// a
```

Access both elements and their indices while looping over an Array (the square brackets before `of` mean that we are using [destructuring](#)):

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

// Output:
// 0. a
// 1. b
```

Looping over the `[key, value]` entries in a Map (the square brackets before `of` mean that we are using [destructuring](#)):

```
const map = new Map([
  [false, 'no'],
  [true, 'yes'],
]);
for (const [key, value] of map) {
  console.log(`${key} => ${value}`);
}

// Output:
// false => no
// true => yes
```

17.2 Introducing the `for-of` loop

`for-of` lets you loop over data structures that are *iterable*: Arrays, strings, Maps, Sets and others. How exactly iterability works is explained in Chap. “[Iterables and iterators](#)”. But you don’t have to know the details if you use the `for-of` loop:

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

// Output:
// a
// b
```

`for-of` goes through the items of `iterable` and assigns them, one at a time, to the loop variable `x`, before it executes the body. The scope of `x` is the loop, it only exists inside it.

You can use `break` and `continue`:

```
for (const x of ['a', '', 'b']) {
  if (x.length === 0) break;
  console.log(x);
}

// Output:
// a
```

`for-of` combines the advantages of:

- Normal `for` loops: `break/continue`; usable in generators
- `forEach()` methods: concise syntax

17.3 Pitfall: `for-of` only works with iterable values

The operand of the `of` clause must be iterable. That means that you need a helper function if you want to iterate over plain objects (see “[Plain objects are not iterable](#)”). If a value is Array-like, you can convert it to an Array via [Array.from\(\)](#):

```
// Array-like, but not iterable!
const arrayLike = { length: 2, 0: 'a', 1: 'b' };

for (const x of arrayLike) { // TypeError
  console.log(x);
}
```

```
for (const x of Array.from(arrayLike)) { // OK
  console.log(x);
}
```

17.4 Iteration variables: `const` declarations versus `var` declarations

If you `const`-declare the iteration variable, a fresh *binding* (storage space) will be created for each iteration. That can be seen in the following code snippet where we save the current binding of `elem` for later, via an arrow function. Afterwards, you can see that the arrow functions don't share the same binding for `elem`, they each have a different one.

```
const arr = [];
for (const elem of [0, 1, 2]) {
  arr.push(() => elem); // save `elem` for later
}
console.log(arr.map(f => f())); // [0, 1, 2]

// `elem` only exists inside the loop:
console.log(elem); // ReferenceError: elem is not defined
```

A `let` declaration works the same way as a `const` declaration (but the bindings are mutable).

It is instructive to see how things are different if you `var`-declare the iteration variable. Now all arrow functions refer to the same binding of `elem`.

```
const arr = [];
for (var elem of [0, 1, 2]) {
  arr.push(() => elem);
}
console.log(arr.map(f => f())); // [2, 2, 2]

// `elem` exists in the surrounding function:
console.log(elem); // 2
```

Having one binding per iteration is very helpful whenever you create functions via a loop (e.g. to add event listeners).

You also get per-iteration bindings in `for` loops (via `let`) and `for-in` loops (via `const` or `let`). Details are explained in [the chapter on variables](#).

17.5 Iterating with existing variables, object properties and Array elements

So far, we have only seen `for-of` with a declared iteration variable. But there are several other forms.

You can iterate with an existing variable:

```
let x;
for (x of ['a', 'b']) {
  console.log(x);
}
```

You can also iterate with an object property:

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

And you can iterate with an Array element:

```
const arr = [];
for (arr[0] of ['a', 'b']) {
  console.log(arr[0]);
}
```

17.6 Iterating with a destructuring pattern

Combining `for-of` with destructuring is especially useful for iterables over `[key, value]` pairs (encoded as Arrays). That's what Maps are:

```
const map = new Map().set(false, 'no').set(true, 'yes');
for (const [k,v] of map) {
  console.log(`key = ${k}, value = ${v}`);
}
// Output:
// key = false, value = no
// key = true, value = yes
```

`Array.prototype.entries()` also returns an iterable over `[key, value]` pairs:

```
const arr = ['a', 'b', 'c'];
for (const [k,v] of arr.entries()) {
  console.log(`key = ${k}, value = ${v}`);
}
// Output:
// key = 0, value = a
// key = 1, value = b
// key = 2, value = c
```

Therefore, `entries()` gives you a way to treat iterated items differently, depending on their position:

```
/** Same as arr.join(', ') */
function toString(arr) {
  let result = '';
  for (const [i, elem] of arr.entries()) {
    if (i > 0) {
      result += ', ';
    }
    result += String(elem);
  }
  return result;
}
```

This function is used as follows:

```
> toString(['eeny', 'meeny', 'miny', 'moe'])
'eeny, meeny, miny, moe'
```

18. New Array features

18.1 Overview

New static Array methods:

- `Array.from(arrayLike, mapFunc?, thisArg?)`
- `Array.of(...items)`

New Array.prototype methods:

- Iterating:
 - `Array.prototype.entries()`
 - `Array.prototype.keys()`
 - `Array.prototype.values()`
- Searching for elements:
 - `Array.prototype.find(predicate, thisArg?)`
 - `Array.prototype.findIndex(predicate, thisArg?)`
- `Array.prototype.copyWithin(target, start, end=this.length)`
- `Array.prototype.fill(value, start=0, end=this.length)`

18.2 New static Array methods

The object `Array` has new methods.

18.2.1 `Array.from(arrayLike, mapFunc?, thisArg?)`

`Array.from()`'s basic functionality is to convert two kinds of values to Arrays:

- **Array-like values**, which have a property `length` and indexed elements. Examples include the results of DOM operations such as `document.getElementsByClassName()`.
- **Iterable values**, whose contents can be retrieved one element at a time. Strings and Arrays are iterable, as are ECMAScript's new data structures `Map` and `Set`.

The following is an example of converting an Array-like object to an Array:

```
const arrayLike = { length: 2, 0: 'a', 1: 'b' };

// for-of only works with iterable values
for (const x of arrayLike) { // TypeError
  console.log(x);
}

const arr = Array.from(arrayLike);
for (const x of arr) { // OK, iterable
  console.log(x);
}
// Output:
// a
// b
```

18.2.1.1 Mapping via `Array.from()`

`Array.from()` is also a convenient alternative to using `map()` **generically**:

```
const spans = document.querySelectorAll('span.name');

// map(), generically:
const names1 = Array.prototype.map.call(spans, s => s.textContent);

// Array.from():
const names2 = Array.from(spans, s => s.textContent);
```

In this example, the result of `document.querySelectorAll()` is again an Array-like object, not an Array, which is why we couldn't invoke `map()` on it. Previously, we converted the Array-like object to an Array in order to call `forEach()`. Here, we skipped

that intermediate step via a generic method call and via the two-parameter version of `Array.from()`.

18.2.1.2 `from()` in subclasses of `Array`

Another use case for `Array.from()` is to convert an Array-like or iterable value to an instance of a subclass of `Array`. For example, if you create a subclass `MyArray` of `Array` and want to convert such an object to an instance of `MyArray`, you simply use `MyArray.from()`. The reason that that works is because constructors inherit from each other in ECMAScript 6 (a super-constructor is the prototype of its sub-constructors).

```
class MyArray extends Array {
  ...
}
const instanceOfMyArray = MyArray.from(anIterable);
```

You can also combine this functionality with mapping, to get a map operation where you control the result's constructor:

```
// from() - determine the result's constructor via the receiver
// (in this case, MyArray)
const instanceOfMyArray = MyArray.from([1, 2, 3], x => x * x);

// map(): the result is always an instance of Array
const instanceOfArray = [1, 2, 3].map(x => x * x);
```

The species pattern lets you configure what instances non-static built-in methods (such as `slice()`, `filter()` and `map()`) return. It is explained in Sect. “The species pattern” in Chap. “Classes”.

18.2.2 `Array.of(...items)`

`Array.of(item_0, item_1, ...)` creates an `Array` whose elements are `item_0`, `item_1`, etc.

18.2.2.1 `Array.of()` as an `Array` literal for subclasses of `Array`

If you want to turn several values into an `Array`, you should always use an `Array` literal, especially since the `Array` constructor doesn't work properly if there is a single value that is a number ([more information](#) on this quirk):

```
> new Array(3, 11, 8)
[ 3, 11, 8 ]
> new Array(3)
[ , , ]
> new Array(3.1)
RangeError: Invalid array length
```

But how are you supposed to turn values into an instance of a sub-constructor of `Array` then? This is where `Array.of()` helps (remember that sub-constructors of `Array` inherit all of `Array`'s methods, including `of()`).

```
class MyArray extends Array {
  ...
}
console.log(MyArray.of(3, 11, 8) instanceof MyArray); // true
console.log(MyArray.of(3).length === 1); // true
```

18.3 New `Array`.prototype methods

Several new methods are available for `Array` instances.

18.3.1 Iterating over `Arrays`

The following methods help with iterating over `Arrays`:

- `Array.prototype.entries()`
- `Array.prototype.keys()`
- `Array.prototype.values()`

The result of each of the aforementioned methods is a sequence of values, but they are not returned as an `Array`; they are revealed one by one, via an iterator. Let's look at an example. I'm using `Array.from()` to put the iterators' contents into `Arrays`:

```
> Array.from(['a', 'b'].keys())
[ 0, 1 ]
> Array.from(['a', 'b'].values())
[ 'a', 'b' ]
> Array.from(['a', 'b'].entries())
[ [ 0, 'a' ],
  [ 1, 'b' ] ]
```

I could also have used [the spread operator \(...\)](#) to convert iterators to `Arrays`:

```
> [...['a', 'b'].keys()]
[ 0, 1 ]
```

18.3.1.1 Iterating over `[index, element]` pairs

You can combine `entries()` with ECMAScript 6's `for-of` loop and destructuring to conveniently iterate over `[index, element]` pairs:

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

18.3.2 Searching for Array elements

`Array.prototype.find(predicate, thisArg?)`

Returns the first Array element for which the callback `predicate` returns `true`. If there is no such element, it returns `undefined`. Example:

```
> [6, -5, 8].find(x => x < 0)
-5
> [6, 5, 8].find(x => x < 0)
undefined
```

`Array.prototype.findIndex(predicate, thisArg?)`

Returns the index of the first element for which the callback `predicate` returns `true`. If there is no such element, it returns `-1`. Example:

```
> [6, -5, 8].findIndex(x => x < 0)
1
> [6, 5, 8].findIndex(x => x < 0)
-1
```

The full signature of the callback `predicate` is:

```
predicate(element, index, array)
```

18.3.2.1 Finding NaN via `findIndex()`

A well-known [limitation](#) of `Array.prototype.indexOf()` is that it can't find `NaN`, because it searches for elements via `===`:

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

With `findIndex()`, you can use `Object.is()` (explained in [the chapter on OOP](#)) and will have no such problem:

```
> [NaN].findIndex(y => Object.is(NaN, y))
0
```

You can also adopt a more general approach, by creating a helper function `elemIs()`:

```
> function elemIs(x) { return Object.is.bind(Object, x) }
> [NaN].findIndex(elemIs(NaN))
0
```

18.3.3 `Array.prototype.copyWithin()`

The signature of this method is:

```
Array.prototype.copyWithin(target : number,
  start : number, end = this.length) : This
```

It copies the elements whose indices are in the range `[start,end)` to index `target` and subsequent indices. If the two index ranges overlap, care is taken that all source elements are copied before they are overwritten.

Example:

```
> const arr = [0,1,2,3];
> arr.copyWithin(2, 0, 2)
[ 0, 1, 0, 1 ]
> arr
[ 0, 1, 0, 1 ]
```

18.3.4 `Array.prototype.fill()`

The signature of this method is:

```
Array.prototype.fill(value : any, start=0, end=this.length) : This
```

It fills an Array with the given `value`:

```
> const arr = ['a', 'b', 'c'];
> arr.fill(7)
[ 7, 7, 7 ]
> arr
[ 7, 7, 7 ]
```

Optionally, you can restrict where the filling starts and ends:

```
> ['a', 'b', 'c'].fill(7, 1, 2)
[ 'a', 7, 'c' ]
```

18.4 ES6 and holes in Arrays

Holes are indices “inside” an Array that have no associated element. In other words: An

Array `arr` is said to have a hole at index `i` if:

- $0 \leq i < \text{arr.length}$
- `!(i in arr)`

For example: The following Array has a hole at index 1.

```
> const arr = ['a',, 'b']
'use strict'
> 0 in arr
true
> 1 in arr
false
> 2 in arr
true
> arr[1]
undefined
```

You'll see lots of examples involving holes in this section. Should anything ever be unclear, you can consult Sect. [“Holes in Arrays”](#) in “Speaking JavaScript” for more information.



ES6 pretends that holes don't exist (as much as it can while being backward-compatible). And so should you – especially if you consider that holes can also affect performance negatively. Then you don't have to burden your brain with the numerous and inconsistent rules around holes.

18.4.1 ECMAScript 6 treats holes like `undefined` elements

The general rule for Array methods that are new in ES6 is: each hole is treated as if it were the element `undefined`. Examples:

```
> Array.from(['a',, 'b'])
[ 'a', undefined, 'b' ]
> [, 'a'].findIndex(x => x === undefined)
0
> [...[, 'a']].entries()
[ [ 0, undefined ], [ 1, 'a' ] ]
```

The idea is to steer people away from holes and to simplify long-term. Unfortunately that means that things are even more inconsistent now.

18.4.2 Array operations and holes

18.4.2.1 Iteration

The iterator created by `Array.prototype[Symbol.iterator]` treats each hole as if it were the element `undefined`. Take, for example, the following iterator `iter`:

```
> var arr = [, 'a'];
> var iter = arr[Symbol.iterator]();
```

If we invoke `next()` twice, we get the hole at index 0 and the element `'a'` at index 1. As you can see, the former produces `undefined`:

```
> iter.next()
{ value: undefined, done: false }
> iter.next()
{ value: 'a', done: false }
```

Among others, two operations are based on [the iteration protocol](#). Therefore, these operations also treat holes as `undefined` elements.

First, the spread operator (`...`):

```
> [...[, 'a']]
[ undefined, 'a' ]
```

Second, the `for-of` loop:

```
for (const x of [, 'a']) {
  console.log(x);
}
// Output:
// undefined
// a
```

Note that the Array prototype methods (`filter()` etc.) do not use the iteration protocol.

18.4.2.2 `Array.from()`

If its argument is iterable, `Array.from()` uses iteration to convert it to an Array. Then it works exactly like the spread operator:

```
> Array.from([, 'a'])
[ undefined, 'a' ]
```

But `Array.from()` can also convert [Array-like objects](#) to Arrays. Then holes become

undefined, too:

```
> Array.from({1: 'a', length: 2})
[ undefined, 'a' ]
```

With a second argument, `Array.from()` works mostly like `Array.prototype.map()`.

However, `Array.from()` treats holes as `undefined`:

```
> Array.from(['a'], x => x)
[ undefined, 'a' ]
> Array.from(['a'], (x,i) => i)
[ 0, 1 ]
```

`Array.prototype.map()` skips them, but preserves them:

```
> ['a'].map(x => x)
[ , 'a' ]
> ['a'].map((x,i) => i)
[ , 1 ]
```

18.4.2.3 `Array.prototype` methods

In ECMAScript 5, behavior already varied slightly. For example:

- `forEach()`, `filter()`, `every()` and `some()` ignore holes.
- `map()` skips but preserves holes.
- `join()` and `toString()` treat holes as if they were `undefined` elements, but interprets both `null` and `undefined` as empty strings.

ECMAScript 6 adds new kinds of behaviors:

- `copyWithin()` creates holes when copying holes (i.e., it deletes elements if necessary).
- `entries()`, `keys()`, `values()` treat each hole as if it was the element `undefined`.
- `find()` and `findIndex()` do the same.
- `fill()` doesn't care whether there are elements at indices or not.

The following table describes how `Array.prototype` methods handle holes.

| Method | Holes are | |
|--|-----------|---|
| <code>concat</code> | Preserved | <code>['a',, 'b'].concat(['c',, 'd']) → ['a',, 'b', 'c',, 'd']</code> |
| <code>copyWithin</code> ^{ES6} | Preserved | <code>[, 'a', 'b',,].copyWithin(2, 0) → [, 'a',, 'a']</code> |
| <code>entries</code> ^{ES6} | Elements | <code>[...[, 'a']].entries() → [[0, undefined], [1, 'a']]</code> |
| <code>every</code> | Ignored | <code>[, 'a'].every(x => x === 'a') → true</code> |
| <code>fill</code> ^{ES6} | Filled | <code>new Array(3).fill('a') → ['a', 'a', 'a']</code> |
| <code>filter</code> | Removed | <code>['a',, 'b'].filter(x => true) → ['a', 'b']</code> |
| <code>find</code> ^{ES6} | Elements | <code>[, 'a'].find(x => true) → undefined</code> |
| <code>findIndex</code> ^{ES6} | Elements | <code>[, 'a'].findIndex(x => true) → 0</code> |
| <code>forEach</code> | Ignored | <code>[, 'a'].forEach((x, i) => log(i)); → 1</code> |
| <code>indexOf</code> | Ignored | <code>[, 'a'].indexOf(undefined) → -1</code> |
| <code>join</code> | Elements | <code>[, 'a', undefined, null].join('#') → '#a##'</code> |
| <code>keys</code> ^{ES6} | Elements | <code>[...[, 'a']].keys() → [0, 1]</code> |
| <code>lastIndexOf</code> | Ignored | <code>[, 'a'].lastIndexOf(undefined) → -1</code> |
| <code>map</code> | Preserved | <code>[, 'a'].map(x => 1) → [, 1]</code> |
| <code>pop</code> | Elements | <code>['a',,].pop() → undefined</code> |
| <code>push</code> | Preserved | <code>new Array(1).push('a') → 2</code> |
| <code>reduce</code> | Ignored | <code>['#',, undefined].reduce((x, y) => x+y) → '#undefined'</code> |
| <code>reduceRight</code> | Ignored | <code>['#',, undefined].reduceRight((x, y) => x+y) → 'undefined#'</code> |
| <code>reverse</code> | Preserved | <code>['a',, 'b'].reverse() → ['b',, 'a']</code> |
| <code>shift</code> | Elements | <code>[, 'a'].shift() → undefined</code> |
| <code>slice</code> | Preserved | <code>[, 'a'].slice(0, 1) → [,]</code> |
| <code>some</code> | Ignored | <code>[, 'a'].some(x => x !== 'a') → false</code> |
| <code>sort</code> | Preserved | <code>[, undefined, 'a'].sort() → ['a', undefined,,]</code> |
| <code>splice</code> | Preserved | <code>['a',,].splice(1, 1) → [,]</code> |
| <code>toString</code> | Elements | <code>[, 'a', undefined, null].toString() → ',a,,'</code> |
| <code>unshift</code> | Preserved | <code>[, 'a'].unshift('b') → 3</code> |
| <code>values</code> ^{ES6} | Elements | <code>[...[, 'a']].values() → [undefined, 'a']</code> |

Notes:

- ES6 methods are marked via the superscript “ES6”.
- JavaScript ignores a trailing comma in an Array literal: `['a',,].length → 2`
- Helper function used in the table: `const log = console.log.bind(console);`

18.4.3 Creating Arrays filled with values

Holes being treated as `undefined` elements by the new ES6 operations helps with creating Arrays that are filled with values.

18.4.3.1 Filling with a fixed value

`Array.prototype.fill()` replaces all Array elements (incl. holes) with a fixed value:

```
> new Array(3).fill(7)
[ 7, 7, 7 ]
```

`new Array(3)` creates an Array with three holes and `fill()` replaces each hole with the value 7.

18.4.3.2 Filling with ascending numbers

`Array.prototype.keys()` reports keys even if an Array only has holes. It returns an iterable, which you can convert to an Array via the spread operator:

```
> [...new Array(3).keys()]
[ 0, 1, 2 ]
```

18.4.3.3 Filling with computed values

The mapping function in the second parameter of `Array.from()` is notified of holes. Therefore, you can use `Array.from()` for more sophisticated filling:

```
> Array.from(new Array(5), (x,i) => i*2)
[ 0, 2, 4, 6, 8 ]
```

18.4.3.4 Filling with `undefined`

If you need an Array that is filled with `undefined`, you can use the fact that iteration (as triggered by the spread operator) converts holes to `undefined`s:

```
> [...new Array(3)]
[ undefined, undefined, undefined ]
```

18.4.4 Removing holes from Arrays

The ES5 method `filter()` lets you remove holes:

```
> ['a',, 'c'].filter(() => true)
[ 'a', 'c' ]
```

ES6 iteration (triggered via the spread operator) lets you convert holes to `undefined` elements:

```
> [...['a',, 'c']]
[ 'a', undefined, 'c' ]
```

18.5 Configuring which objects are spread by `concat()` (`Symbol.isConcatSpreadable`)

You can configure how `Array.prototype.concat()` treats objects by adding an (own or inherited) property whose key is the well-known symbol `Symbol.isConcatSpreadable` and whose value is a boolean.

18.5.1 Default for Arrays: spreading

By default, `Array.prototype.concat()` *spreads* Arrays into its result: their indexed elements become elements of the result:

```
const arr1 = ['c', 'd'];
['a', 'b'].concat(arr1, 'e');
// ['a', 'b', 'c', 'd', 'e']
```

With `Symbol.isConcatSpreadable`, you can override the default and avoid spreading for Arrays:

```
const arr2 = ['c', 'd'];
arr2[Symbol.isConcatSpreadable] = false;
['a', 'b'].concat(arr2, 'e');
// ['a', 'b', ['c','d'], 'e']
```

18.5.2 Default for non-Arrays: no spreading

For non-Arrays, the default is not to spread:

```
const arrayLike = {length: 2, 0: 'c', 1: 'd'};
console.log(['a', 'b'].concat(arrayLike, 'e'));
// ['a', 'b', arrayLike, 'e']

console.log(Array.prototype.concat.call(
  arrayLike, ['e','f'], 'g'));
// [arrayLike, 'e', 'f', 'g']
```

You can use `Symbol.isConcatSpreadable` to force spreading:

```
arrayLike[Symbol.isConcatSpreadable] = true;

console.log(['a', 'b'].concat(arrayLike, 'e'));
// ['a', 'b', 'c', 'd', 'e']

console.log(Array.prototype.concat.call(
  arrayLike, ['e', 'f'], 'g'));
// ['c', 'd', 'e', 'f', 'g']
```

18.5.3 Detecting Arrays

How does `concat()` determine if a parameter is an Array? It uses the same algorithm as `Array.isArray()`. Whether or not `Array.prototype` is in the prototype chain makes no difference for that algorithm. That is important, because, in ES5 and earlier, hacks were used to subclass `Array` and those must continue to work (see the section on `__proto__` in this book):

```
> const arr = [];
> Array.isArray(arr)
true

> Object.setPrototypeOf(arr, null);
> Array.isArray(arr)
true
```

18.5.4 `Symbol.isConcatSpreadable` in the standard library

No object in the ES6 standard library has a property with the key `Symbol.isConcatSpreadable`. This mechanism therefore exists purely for browser APIs and user code.

Consequences:

- Subclasses of `Array` are spread by default (because their instances are `Array` objects).
- A subclass of `Array` can prevent its instances from being spread by setting a property to `false` whose key is `Symbol.isConcatSpreadable`. That property can be a prototype property or an instance property.
- Other Array-like objects are spread by `concat()` if property `[Symbol.isConcatSpreadable]` is `true`. That would enable one, for example, to turn on spreading for some Array-like DOM collections.
- Typed Arrays are not spread. They don't have a method `concat()`, either.



`Symbol.isConcatSpreadable` in the ES6 spec

- In the description of `Array.prototype.concat()`, you can see that spreading requires an object to be Array-like (property `length` plus indexed elements).
- Whether or not to spread an object is determined via the spec operation `IsConcatSpreadable()`. The last step is the default (equivalent to `Array.isArray()`) and the property `[Symbol.isConcatSpreadable]` is retrieved via a normal `Get()` operation, meaning that it doesn't matter whether it is own or inherited.

18.6 The numeric range of Array indices

For Arrays, ES6 still has the same rules as ES5:

- Array lengths `l` are in the range $0 \leq l \leq 2^{32}-1$.
- Array indices `i` are in the range $0 \leq i < 2^{32}-1$.

However, Typed Arrays have a larger range of indices: $0 \leq i < 2^{53}-1$ ($2^{53}-1$ is the largest integer that JavaScript's floating point numbers can safely represent). That's why generic Array methods such as `push()` and `unshift()` allow a larger range of indices. Range checks appropriate for Arrays are performed elsewhere, whenever `length` is set.

19. Maps and Sets

19.1 Overview

Among others, the following four data structures are new in ECMAScript 6: `Map`, `WeakMap`, `Set` and `WeakSet`.

19.1.1 Maps

The keys of a Map can be arbitrary values:

```
> const map = new Map(); // create an empty Map
> const KEY = {};

> map.set(KEY, 123);
> map.get(KEY)
123
> map.has(KEY)
true
> map.delete(KEY);
true
> map.has(KEY)
false
```

You can use an Array (or any iterable) with [key, value] pairs to set up the initial data in the Map:

```
const map = new Map([
  [ 1, 'one' ],
  [ 2, 'two' ],
  [ 3, 'three' ], // trailing comma is ignored
]);
```

19.1.2 Sets

A Set is a collection of unique elements:

```
const arr = [5, 1, 5, 7, 7, 5];
const unique = [...new Set(arr)]; // [ 5, 1, 7 ]
```

As you can see, you can initialize a Set with elements if you hand the constructor an iterable (arr in the example) over those elements.

19.1.3 WeakMaps

A WeakMap is a Map that doesn't prevent its keys from being garbage-collected. That means that you can associate data with objects without having to worry about memory leaks. For example:

```
//----- Manage listeners

const _objToListeners = new WeakMap();

function addListener(obj, listener) {
  if (!_objToListeners.has(obj)) {
    _objToListeners.set(obj, new Set());
  }
  _objToListeners.get(obj).add(listener);
}

function triggerListeners(obj) {
  const listeners = _objToListeners.get(obj);
  if (listeners) {
    for (const listener of listeners) {
      listener();
    }
  }
}

//----- Example: attach listeners to an object

const obj = {};
addListener(obj, () => console.log('hello'));
addListener(obj, () => console.log('world'));

//----- Example: trigger listeners

triggerListeners(obj);

// Output:
// hello
// world
```

19.2 Map

JavaScript has always had a very spartan standard library. Sorely missing was a data structure for mapping values to values. The best you can get in ECMAScript 5 is a Map from strings to arbitrary values, by abusing objects. Even then there are [several pitfalls](#) that can trip you up.

The Map data structure in ECMAScript 6 lets you use arbitrary values as keys and is highly welcome.

19.2.1 Basic operations

Working with single entries:

```
> const map = new Map();

> map.set('foo', 123);
> map.get('foo')
123

> map.has('foo')
true
> map.delete('foo')
```

```
true
> map.has('foo')
false
```

Determining the size of a Map and clearing it:

```
> const map = new Map();
> map.set('foo', true);
> map.set('bar', false);

> map.size
2
> map.clear();
> map.size
0
```

19.2.2 Setting up a Map

You can set up a Map via an iterable over key-value “pairs” (Arrays with 2 elements). One possibility is to use an Array (which is iterable):

```
const map = new Map([
  [ 1, 'one' ],
  [ 2, 'two' ],
  [ 3, 'three' ], // trailing comma is ignored
]);
```

Alternatively, the `set()` method is chainable:

```
const map = new Map()
.set(1, 'one')
.set(2, 'two')
.set(3, 'three');
```

19.2.3 Keys

Any value can be a key, even an object:

```
const map = new Map();

const KEY1 = {};
map.set(KEY1, 'hello');
console.log(map.get(KEY1)); // hello

const KEY2 = {};
map.set(KEY2, 'world');
console.log(map.get(KEY2)); // world
```

19.2.3.1 What keys are considered equal?

Most Map operations need to check whether a value is equal to one of the keys. They do so via the internal operation [SameValueZero](#), which works like `===`, but considers `NaN` to be equal to itself.

Let’s first see how `===` handles `NaN`:

```
> NaN === NaN
false
```

Conversely, you can use `NaN` as a key in Maps, just like any other value:

```
> const map = new Map();

> map.set(NaN, 123);
> map.get(NaN)
123
```

Like `===`, `-0` and `+0` are considered the same value. That is normally the best way to handle the two zeros ([details are explained in “Speaking JavaScript”](#)).

```
> map.set(-0, 123);
> map.get(+0)
123
```

Different objects are always considered different. That is something that can’t be configured (yet), [as explained later, in the FAQ](#).

```
> new Map().set({}, 1).set({}, 2).size
2
```

Getting an unknown key produces `undefined`:

```
> new Map().get('asfddfsasadf')
undefined
```

19.2.4 Iterating over Maps

Let’s set up a Map to demonstrate how one can iterate over it.

```
const map = new Map([
  [false, 'no'],
  [true, 'yes'],
]);
```

Maps record the order in which elements are inserted and honor that order when iterating over keys, values or entries.

19.2.4.1 Iterables for keys and values

`keys()` returns an **iterable** over the keys in the Map:

```
for (const key of map.keys()) {
  console.log(key);
}
// Output:
// false
// true
```

`values()` returns an iterable over the values in the Map:

```
for (const value of map.values()) {
  console.log(value);
}
// Output:
// no
// yes
```

19.2.4.2 Iterables for entries

`entries()` returns the entries of the Map as an iterable over [key,value] pairs (Arrays).

```
for (const entry of map.entries()) {
  console.log(entry[0], entry[1]);
}
// Output:
// false no
// true yes
```

Destructuring enables you to access the keys and values directly:

```
for (const [key, value] of map.entries()) {
  console.log(key, value);
}
```

The default way of iterating over a Map is `entries()`:

```
> map[Symbol.iterator] === map.entries
true
```

Thus, you can make the previous code snippet even shorter:

```
for (const [key, value] of map) {
  console.log(key, value);
}
```

19.2.4.3 Converting iterables (incl. Maps) to Arrays

The **spread operator** (`...`) can turn an iterable into an Array. That lets us convert the result of `Map.prototype.keys()` (an iterable) into an Array:

```
> const map = new Map().set(false, 'no').set(true, 'yes');
> [...map.keys()]
[ false, true ]
```

Maps are also iterable, which means that the spread operator can turn Maps into Arrays:

```
> const map = new Map().set(false, 'no').set(true, 'yes');
> [...map]
[ [ false, 'no' ],
  [ true, 'yes' ] ]
```

19.2.5 Looping over Map entries

The Map method `forEach` has the following signature:

```
Map.prototype.forEach((value, key, map) => void, thisArg?) : void
```

The signature of the first parameter mirrors the signature of the callback of `Array.prototype.forEach`, which is why the value comes first.

```
const map = new Map([
  [false, 'no'],
  [true, 'yes'],
]);
map.forEach((value, key) => {
  console.log(key, value);
});
// Output:
// false no
// true yes
```

19.2.6 Mapping and filtering Maps

You can `map()` and `filter()` Arrays, but there are no such operations for Maps. The solution is:

1. Convert the Map into an Array of [key,value] pairs.

2. Map or filter the Array.
3. Convert the result back to a Map.

I'll use the following Map to demonstrate how that works.

```
const originalMap = new Map()
.set(1, 'a')
.set(2, 'b')
.set(3, 'c');
```

Mapping originalMap:

```
const mappedMap = new Map( // step 3
  [...originalMap] // step 1
  .map(([k, v]) => [k * 2, '_' + v]) // step 2
);
// Resulting Map: {2 => '_a', 4 => '_b', 6 => '_c'}
```

Filtering originalMap:

```
const filteredMap = new Map( // step 3
  [...originalMap] // step 1
  .filter(([k, v]) => k < 3) // step 2
);
// Resulting Map: {1 => 'a', 2 => 'b'}
```

Step 1 is performed by the spread operator (...) which I have explained previously.

19.2.7 Combining Maps

There are no methods for combining Maps, which is why the approach from the previous section must be used to do so.

Let's combine the following two Maps:

```
const map1 = new Map()
.set(1, 'a1')
.set(2, 'b1')
.set(3, 'c1');

const map2 = new Map()
.set(2, 'b2')
.set(3, 'c2')
.set(4, 'd2');
```

To combine map1 and map2, I turn them into Arrays via the spread operator (...) and concatenate those Arrays. Afterwards, I convert the result back to a Map. All of that is done in the first line.

```
> const combinedMap = new Map([...map1, ...map2])
> [...combinedMap] // convert to Array to display
[ [ 1, 'a1' ],
  [ 2, 'b2' ],
  [ 3, 'c2' ],
  [ 4, 'd2' ] ]
```

19.2.8 Arbitrary Maps as JSON via Arrays of pairs

If a Map contains arbitrary (JSON-compatible) data, we can convert it to JSON by encoding it as an Array of key-value pairs (2-element Arrays). Let's examine first how to achieve that encoding.

19.2.8.1 Converting Maps to and from Arrays of pairs

The spread operator lets you convert a Map to an Array of pairs:

```
> const myMap = new Map().set(true, 7).set({foo: 3}, ['abc']);
> [...myMap]
[ [ true, 7 ], [ { foo: 3 }, [ 'abc' ] ] ]
```

The Map constructor lets you convert an Array of pairs to a Map:

```
> new Map([true, 7], [{foo: 3}, ['abc']])
Map {true => 7, Object {foo: 3} => ['abc']}
```

19.2.8.2 The conversion to and from JSON

Let's use this knowledge to convert any Map with JSON-compatible data to JSON and back:

```
function mapToJson(map) {
  return JSON.stringify([...map]);
}

function jsonToMap(jsonStr) {
  return new Map(JSON.parse(jsonStr));
}
```

The following interaction demonstrates how these functions are used:

```
> const myMap = new Map().set(true, 7).set({foo: 3}, ['abc']);
> mapToJson(myMap)
'[[true,7],[{"foo":3},["abc"]]]'
```



```
> jsonToMap('[[true,7],[{"foo":3},{"abc"}]]')
Map {true => 7, Object {foo: 3} => ['abc']}
```

19.2.9 String Maps as JSON via objects

Whenever a Map only has strings as keys, you can convert it to JSON by encoding it as an object. Let's examine first how to achieve that encoding.

19.2.9.1 Converting a string Map to and from an object

The following two function convert string Maps to and from objects:

```
function strMapToObj(strMap) {
  const obj = Object.create(null);
  for (const [k,v] of strMap) {
    // We don't escape the key '__proto__'
    // which can cause problems on older engines
    obj[k] = v;
  }
  return obj;
}
function objToStrMap(obj) {
  const strMap = new Map();
  for (const k of Object.keys(obj)) {
    strMap.set(k, obj[k]);
  }
  return strMap;
}
```

Let's use these two functions:

```
> const myMap = new Map().set('yes', true).set('no', false);
> strMapToObj(myMap)
{ yes: true, no: false }
> objToStrMap({yes: true, no: false})
[ [ 'yes', true ], [ 'no', false ] ]
```

19.2.9.2 The conversion to and from JSON

With these helper functions, the conversion to JSON works as follows:

```
function strMapToJson(strMap) {
  return JSON.stringify(strMapToObj(strMap));
}
function jsonToStrMap(jsonStr) {
  return objToStrMap(JSON.parse(jsonStr));
}
```

This is an example of using these functions:

```
> const myMap = new Map().set('yes', true).set('no', false);
> strMapToJson(myMap)
'{"yes":true,"no":false}'
> jsonToStrMap('{"yes":true,"no":false}');
Map {'yes' => true, 'no' => false}
```

19.2.10 Map API

Constructor:

- new Map(entries? : Iterable<[any,any]>)
If you don't provide the parameter `iterable` then an empty Map is created. If you do provide an iterable over [key, value] pairs then those pairs are used to add entries to the Map. For example:

```
const map = new Map([
  [ 1, 'one' ],
  [ 2, 'two' ],
  [ 3, 'three' ], // trailing comma is ignored
]);
```

Handling single entries:

- Map.prototype.get(key) : any
Returns the value that key is mapped to in this Map. If there is no key key in this Map, undefined is returned.
- Map.prototype.set(key, value) : this
Maps the given key to the given value. If there is already an entry whose key is key, it is updated. Otherwise, a new entry is created. This method returns this, which means that you can chain it.
- Map.prototype.has(key) : boolean
Returns whether the given key exists in this Map.
- Map.prototype.delete(key) : boolean
If there is an entry whose key is key, it is removed and true is returned. Otherwise, nothing happens and false is returned.

Handling all entries:

- `get Map.prototype.size` : number
Returns how many entries there are in this Map.
- `Map.prototype.clear()` : void
Removes all entries from this Map.

Iterating and looping: happens in the order in which entries were added to a Map.

- `Map.prototype.entries()` : `Iterable<[any,any]>`
Returns an iterable with one `[key,value]` pair for each entry in this Map. The pairs are Arrays of length 2.
- `Map.prototype.forEach((value, key, collection) => void, thisArg?)` : void
The first parameter is a callback that is invoked once for each entry in this Map. If `thisArg` is provided, `this` is set to it for each invocation. Otherwise, `this` is set to `undefined`.
- `Map.prototype.keys()` : `Iterable<any>`
Returns an iterable over all keys in this Map.
- `Map.prototype.values()` : `Iterable<any>`
Returns an iterable over all values in this Map.
- `Map.prototype[Symbol.iterator]()` : `Iterable<[any,any]>`
The default way of iterating over Maps. Refers to `Map.prototype.entries`.

19.3 WeakMap

WeakMaps work mostly like Maps, with the following differences:

- WeakMap keys are objects (values can be arbitrary values)
- WeakMap keys are weakly held
- You can't get an overview of the contents of a WeakMap
- You can't clear a WeakMap

The following sections explain each of these differences.

19.3.1 WeakMap keys are objects

If you add an entry to a WeakMap then the key must be an object:

```
const wm = new WeakMap()

wm.set('abc', 123); // TypeError
wm.set({}, 123); // OK
```

19.3.2 WeakMap keys are weakly held

The keys in a WeakMap are *weakly held*: Normally, an object that isn't referred to by any storage location (variable, property, etc.) can be garbage-collected. WeakMap keys do not count as storage locations in that sense. In other words: an object being a key in a WeakMap does not prevent the object being garbage-collected.

Additionally, once a key is gone, its entry will also disappear (eventually, but there is no way to detect when, anyway).

19.3.3 You can't get an overview of a WeakMap or clear it

It is impossible to inspect the innards of a WeakMap, to get an overview of them. That includes not being able to iterate over keys, values or entries. Put differently: to get content out of a WeakMap, you need a key. There is no way to clear a WeakMap, either (as a work-around, you can create a completely new instance).

These restrictions enable a security property. Quoting [Mark Miller](#): "The mapping from weakmap/key pair value can only be observed or affected by someone who has both the weakmap and the key. With `clear()`, someone with only the WeakMap would've been able to affect the WeakMap-and-key-to-value mapping."

Additionally, iteration would be difficult to implement, because you'd have to guarantee that keys remain weakly held.

19.3.4 Use cases for WeakMaps

WeakMaps are useful for associating data with objects whose life cycle you can't (or don't want to) control. In this section, we look at two examples:

- Caching computed results
- Managing listeners
- Keeping private data

19.3.4.1 Caching computed results via WeakMaps

With WeakMaps, you can associate previously computed results with objects, without having to worry about memory management. The following function `countOwnKeys` is an example: it caches previous results in the WeakMap `cache`.

```
const cache = new WeakMap();
function countOwnKeys(obj) {
  if (cache.has(obj)) {
```

```

        console.log('Cached');
        return cache.get(obj);
    } else {
        console.log('Computed');
        const count = Object.keys(obj).length;
        cache.set(obj, count);
        return count;
    }
}

```

If we use this function with an object `obj`, you can see that the result is only computed for the first invocation, while a cached value is used for the second invocation:

```

> const obj = { foo: 1, bar: 2};
> countOwnKeys(obj)
Computed
2
> countOwnKeys(obj)
Cached
2

```

19.3.4.2 Managing listeners

Let's say we want to attach listeners to objects without changing the objects. You'd be able to add listeners to an object `obj`:

```

const obj = {};
addListener(obj, () => console.log('hello'));
addListener(obj, () => console.log('world'));

```

And you'd be able to trigger the listeners:

```

triggerListeners(obj);

// Output:
// hello
// world

```

The two functions `addListener()` and `triggerListeners()` can be implemented as follows.

```

const _objToListeners = new WeakMap();

function addListener(obj, listener) {
    if (! _objToListeners.has(obj)) {
        _objToListeners.set(obj, new Set());
    }
    _objToListeners.get(obj).add(listener);
}

function triggerListeners(obj) {
    const listeners = _objToListeners.get(obj);
    if (listeners) {
        for (const listener of listeners) {
            listener();
        }
    }
}

```

The advantage of using a `WeakMap` here is that, once an object is garbage-collected, its listeners will be garbage-collected, too. In other words: there won't be any memory leaks.

19.3.4.3 Keeping private data via WeakMaps

In the following code, the `WeakMaps` `_counter` and `_action` are used to store the data of virtual properties of instances of `Countdown`:

```

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);
        if (counter < 1) return;
        counter--;
        _counter.set(this, counter);
        if (counter === 0) {
            _action.get(this)();
        }
    }
}

```

More information on this technique is given in [the chapter on classes](#).

19.3.5 WeakMap API

The constructor and the four methods of `WeakMap` work the same as their `Map` equivalents:

```

new WeakMap(entries? : Iterable<[any,any]>)

WeakMap.prototype.get(key) : any
WeakMap.prototype.set(key, value) : this

```

```
WeakMap.prototype.has(key) : boolean  
WeakMap.prototype.delete(key) : boolean
```

19.4 Set

ECMAScript 5 doesn't have a Set data structure, either. There are two possible work-arounds:

- Use the keys of an object to store the elements of a set of strings.
- Store (arbitrary) set elements in an Array: Check whether it contains an element via `indexOf()`, remove elements via `filter()`, etc. This is not a very fast solution, but it's easy to implement. One issue to be aware of is that `indexOf()` can't find the value `NaN`.

ECMAScript 6 has the data structure `Set` which works for arbitrary values, is fast and handles `NaN` correctly.

19.4.1 Basic operations

Managing single elements:

```
> const set = new Set();  
> set.add('red')  
  
> set.has('red')  
true  
> set.delete('red')  
true  
> set.has('red')  
false
```

Determining the size of a Set and clearing it:

```
> const set = new Set();  
> set.add('red')  
> set.add('green')  
  
> set.size  
2  
> set.clear();  
> set.size  
0
```

19.4.2 Setting up a Set

You can set up a Set via an iterable over the elements that make up the Set. For example, via an Array:

```
const set = new Set(['red', 'green', 'blue']);
```

Alternatively, the `add` method is chainable:

```
const set = new Set().add('red').add('green').add('blue');
```

19.4.2.1 Pitfall: `new Set()` has at most one argument

The `Set` constructor has zero or one arguments:

- Zero arguments: an empty Set is created.
- One argument: the argument needs to be iterable; the iterated items define the elements of the Set.

Further arguments are ignored, which may lead to unexpected results:

```
> Array.from(new Set(['foo', 'bar']))  
[ 'foo', 'bar' ]  
> Array.from(new Set('foo', 'bar'))  
[ 'f', 'o' ]
```

For the second Set, only `'foo'` is used (which is iterable) to define the Set.

19.4.3 Comparing Set elements

As with Maps, elements are compared similarly to `===`, with the exception of `NaN` being like any other value.

```
> const set = new Set([NaN]);  
> set.size  
1  
> set.has(NaN)  
true
```

Adding an element a second time has no effect:

```
> const set = new Set();  
  
> set.add('foo');  
> set.size  
1  
  
> set.add('foo');  
> set.size  
1
```

Similarly to `===`, two different objects are never considered equal (which can't currently be customized, [as explained later, in the FAQ](#), later):

```
> const set = new Set();
> set.add({});
> set.size
1
> set.add({});
> set.size
2
```

19.4.4 Iterating

Sets are iterable and the `for-of` loop works as you'd expect:

```
const set = new Set(['red', 'green', 'blue']);
for (const x of set) {
  console.log(x);
}
// Output:
// red
// green
// blue
```

As you can see, Sets preserve iteration order. That is, elements are always iterated over in the order in which they were inserted.

The [previously explained spread operator](#) (`...`) works with iterables and thus lets you convert a Set to an Array:

```
const set = new Set(['red', 'green', 'blue']);
const arr = [...set]; // ['red', 'green', 'blue']
```

We now have a concise way to convert an Array to a Set and back, which has the effect of eliminating duplicates from the Array:

```
const arr = [3, 5, 2, 2, 5, 5];
const unique = [...new Set(arr)]; // [3, 5, 2]
```

19.4.5 Mapping and filtering

In contrast to Arrays, Sets don't have the methods `map()` and `filter()`. A work-around is to convert them to Arrays and back.

Mapping:

```
const set = new Set([1, 2, 3]);
set = new Set([...set].map(x => x * 2));
// Resulting Set: {2, 4, 6}
```

Filtering:

```
const set = new Set([1, 2, 3, 4, 5]);
set = new Set([...set].filter(x => (x % 2) === 0));
// Resulting Set: {2, 4}
```

19.4.6 Union, intersection, difference

ECMAScript 6 Sets have no methods for computing the union (e.g. `addAll`), intersection (e.g. `retainAll`) or difference (e.g. `removeAll`). This section explains how to work around that limitation.

19.4.6.1 Union

Union ($a \cup b$): create a Set that contains the elements of both Set *a* and Set *b*.

```
const a = new Set([1,2,3]);
const b = new Set([4,3,2]);
const union = new Set([...a, ...b]);
// {1,2,3,4}
```

The pattern is always the same:

- Convert one or both Sets to Arrays.
- Perform the operation.
- Convert the result back to a Set.

The [spread operator](#) (`...`) inserts the elements of something iterable (such as a Set) into an Array. Therefore, `[...a, ...b]` means that *a* and *b* are converted to Arrays and concatenated. It is equivalent to `[...a].concat([...b])`.

19.4.6.2 Intersection

Intersection ($a \cap b$): create a Set that contains those elements of Set *a* that are also in Set *b*.

```
const a = new Set([1,2,3]);
const b = new Set([4,3,2]);
const intersection = new Set(
```

```
[...a].filter(x => b.has(x));
// {2,3}
```

Steps: Convert `a` to an Array, filter the elements, convert the result to a Set.

19.4.6.3 Difference

Difference (`a \ b`): create a Set that contains those elements of Set `a` that are not in Set `b`. This operation is also sometimes called *minus* (-).

```
const a = new Set([1,2,3]);
const b = new Set([4,3,2]);
const difference = new Set(
  [...a].filter(x => !b.has(x));
// {1}
```

19.4.7 Set API

Constructor:

- `new Set(elements? : Iterable<any>)`
If you don't provide the parameter `iterable` then an empty Set is created. If you do then the iterated values are added as elements to the Set. For example:

```
const set = new Set(['red', 'green', 'blue']);
```

Single Set elements:

- `Set.prototype.add(value) : this`
Adds `value` to this Set. This method returns `this`, which means that it can be chained.
- `Set.prototype.has(value) : boolean`
Checks whether `value` is in this Set.
- `Set.prototype.delete(value) : boolean`
Removes `value` from this Set.

All Set elements:

- `get Set.prototype.size : number`
Returns how many elements there are in this Set.
- `Set.prototype.clear() : void`
Removes all elements from this Set.

Iterating and looping:

- `Set.prototype.values() : Iterable<any>`
Returns an iterable over all elements of this Set.
- `Set.prototype[Symbol.iterator]() : Iterable<any>`
The default way of iterating over Sets. Points to `Set.prototype.values`.
- `Set.prototype.forEach((value, key, collection) => void, thisArg?)`
Loops over the elements of this Set and invokes the callback (first parameter) for each one. `value` and `key` are both set to the element, so that this method works similarly to `Map.prototype.forEach`. If `thisArg` is provided, this is set to it for each call. Otherwise, this is set to `undefined`.

Symmetry with Map: The following two methods only exist so that the interface of Sets is similar to the interface of Maps. Each Set element is handled as if it were a Map entry whose key and value are the element.

- `Set.prototype.entries() : Iterable<[any,any]>`
- `Set.prototype.keys() : Iterable<any>`

`entries()` allows you to convert a Set to a Map:

```
const set = new Set(['a', 'b', 'c']);
const map = new Map(set.entries());
// Map { 'a' => 'a', 'b' => 'b', 'c' => 'c' }
```

19.5 WeakSet

A `WeakSet` is a Set that doesn't prevent its elements from being garbage-collected. Consult the section on `WeakMap` for an explanation of why `WeakSets` don't allow iteration, looping and clearing.

19.5.1 Use cases for WeakSets

Given that you can't iterate over their elements, there are not that many use cases for `WeakSets`. They do enable you to mark objects.

19.5.1.1 Marking objects created by a factory function

For example, if you have a factory function for proxies, you can use a `WeakSet` to record which objects were created by that factory:

```
const _proxies = new WeakSet();

function createProxy(obj) {
```

```

    const proxy = ...;
    _proxies.add(proxy);
    return proxy;
}

function isProxy(obj) {
    return _proxies.has(obj);
}

```

The complete example is shown in [the chapter on proxies](#).

`_proxies` must be a `WeakSet`, because a normal `Set` would prevent a proxy from being garbage-collected once it isn't referred to, anymore.

19.5.1.2 Marking objects as safe to use with a method

Domenic Denicola shows how a class `Foo` can ensure that its methods are only applied to instances that were created by it:

```

const foos = new WeakSet();

class Foo {
  constructor() {
    foos.add(this);
  }

  method() {
    if (!foos.has(this)) {
      throw new TypeError('Incompatible object!');
    }
  }
}

```

19.5.2 WeakSet API

The constructor and the three methods of `WeakSet` work the same as their `Set` equivalents:

```

new WeakSet(elements? : Iterable<any>)

WeakSet.prototype.add(value)
WeakSet.prototype.has(value)
WeakSet.prototype.delete(value)

```

19.6 FAQ: Maps and Sets

19.6.1 Why do Maps and Sets have the property `size` and not `length`?

Arrays have the property `length` to count the number of entries. Maps and Sets have a different property, `size`.

The reason for this difference is that `length` is for sequences, data structures that are indexable – like Arrays. `size` is for collections that are primarily unordered – like Maps and Sets.

19.6.2 Why can't I configure how Maps and Sets compare keys and values?

It would be nice if there were a way to configure what Map keys and what Set elements are considered equal. But that feature has been postponed, as it is difficult to implement properly and efficiently.

19.6.3 Is there a way to specify a default value when getting something out of a Map?

If you use a key to get something out of a Map, you'd occasionally like to specify a default value that is returned if the key is not in the Map. ES6 Maps don't let you do this directly. But you can use the `or` operator (`||`), as demonstrated in the following code.

`countChars` returns a Map that maps characters to numbers of occurrences.

```

function countChars(chars) {
  const charCounts = new Map();
  for (const ch of chars) {
    ch = ch.toLowerCase();
    const prevCount = charCounts.get(ch) || 0; // (A)
    charCounts.set(ch, prevCount+1);
  }
  return charCounts;
}

```

In line A, the default `0` is used if `ch` is not in the `charCounts` and `get()` returns `undefined`.

19.6.4 When should I use a Map, when an object?

If you map anything other than strings to any kind of data, you have no choice: you must use a Map.

If, however, you are mapping strings to arbitrary data, you must decide whether or not to use an object. A rough general guideline is:

- Is there a fixed set of keys (known at development time)?

- Then use an object and access the values via fixed keys: `obj.key`
- Can the set of keys change at runtime?
Then use a Map and access the values via keys stored in variables:
`map.get(theKey)`

19.6.5 When would I use an object as a key in a Map?

Map keys mainly make sense if they are compared by value (the same “content” means that two values are considered equal, not the same identity). That excludes objects.

There is one use case – externally attaching data to objects, but that use case is better served by WeakMaps where an entry goes away when the key disappears.

20. Typed Arrays

20.1 Overview

Typed Arrays are an ECMAScript 6 API for handling binary data.

Code example:

```
const typedArray = new Uint8Array([0,1,2]);
console.log(typedArray.length); // 3
typedArray[0] = 5;
const normalArray = [...typedArray]; // [5,1,2]

// The elements are stored in typedArray.buffer.
// Get a different view on the same data:
const dataView = new DataView(typedArray.buffer);
console.log(dataView.getUint8(0)); // 5
```

Instances of `ArrayBuffer` store the binary data to be processed. Two kinds of views are used to access the data:

- Typed Arrays (`Uint8Array`, `Int16Array`, `Float32Array`, etc.) interpret the `ArrayBuffer` as an indexed sequence of elements of a single type.
- Instances of `DataView` let you access data as elements of several types (`Uint8`, `Int16`, `Float32`, etc.), at any byte offset inside an `ArrayBuffer`.

The following browser APIs support Typed Arrays ([details are mentioned in a dedicated section](#)):

- File API
- XMLHttpRequest
- Fetch API
- Canvas
- WebSockets
- And more

20.2 Introduction

Much data one encounters on the web is text: JSON files, HTML files, CSS files, JavaScript code, etc. For handling such data, JavaScript’s built-in string data type works well. However, until a few years ago, JavaScript was ill-equipped to handle binary data. On 8 February 2011, [the Typed Array Specification 1.0](#) standardized facilities for handling binary data. By now, Typed Arrays are [well supported](#) by various engines. With ECMAScript 6, they became part of the core language and gained many methods in the process that were previously only available for Arrays (`map()`, `filter()`, etc.).

The main uses cases for Typed Arrays are:

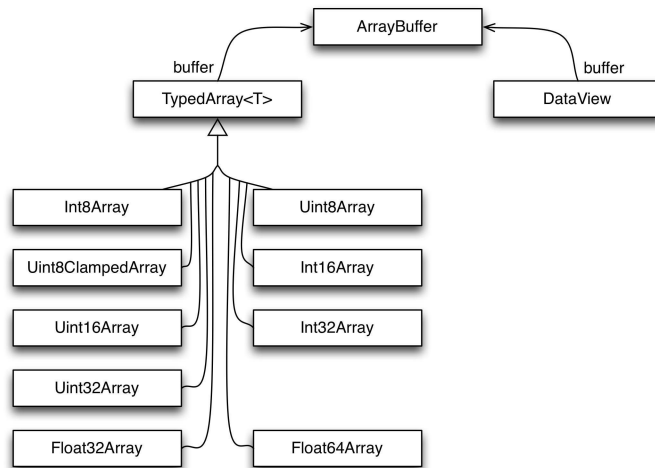
- Processing binary data: manipulating image data in HTML Canvas elements, parsing binary files, handling binary network protocols, etc.
- Interacting with native APIs: Native APIs often receive and return data in a binary format, which you could neither store nor manipulate well in traditional JavaScript. That meant that whenever you were communicating with such an API, data had to be converted from JavaScript to binary and back, for every call. Typed Arrays eliminate this bottleneck. One example of communicating with native APIs is WebGL, for which Typed Arrays were initially created. Section “[History of Typed Arrays](#)” of the article “[Typed Arrays: Binary Data in the Browser](#)” (by Ilmari Heikkinen for HTML5 Rocks) has more information.

Two kinds of objects work together in the Typed Array API:

- Buffers: Instances of `ArrayBuffer` hold the binary data.
- Views: provide the methods for accessing the binary data. There are two kinds of views:
 - An instance of a Typed Array constructor (`Uint8Array`, `Float64Array`, etc.) works much like a normal Array, but only allows a single type for its elements and doesn’t have holes.

- An instance of `DataView` lets you access data at any byte offset in the buffer, and interprets that data as one of several types (`Uint8`, `Float64`, etc.).

This is a diagram of the structure of the Typed Array API (notable: all Typed Arrays have a common superclass):



20.2.1 Element types

The following element types are supported by the API:

| Element type | Bytes | Description | C type |
|--------------|-------|---|----------------|
| Int8 | 1 | 8-bit signed integer | signed char |
| Uint8 | 1 | 8-bit unsigned integer | unsigned char |
| Uint8C | 1 | 8-bit unsigned integer (clamped conversion) | unsigned char |
| Int16 | 2 | 16-bit signed integer | short |
| Uint16 | 2 | 16-bit unsigned integer | unsigned short |
| Int32 | 4 | 32-bit signed integer | int |
| Uint32 | 4 | 32-bit unsigned integer | unsigned int |
| Float32 | 4 | 32-bit floating point | float |
| Float64 | 8 | 64-bit floating point | double |

The element type `Uint8C` is special: it is not supported by `DataView` and only exists to enable `Uint8ClampedArray`. This Typed Array is used by the `canvas` element (where it replaces `CanvasPixelArray`). The only difference between `Uint8C` and `Uint8` is how overflow and underflow are handled (as explained in the next section). It is recommended to avoid the former – quoting [Brendan Eich](#):

Just to be super-clear (and I was around when it was born), `Uint8ClampedArray` is *totally* a historical artifact (of the HTML5 canvas element). Avoid unless you really are doing canvas-y things.

20.2.2 Handling overflow and underflow

Normally, when a value is out of the range of the element type, modulo arithmetic is used to convert it to a value within range. For signed and unsigned integers that means that:

- The highest value plus one is converted to the lowest value (0 for unsigned integers).
- The lowest value minus one is converted to the highest value.

Modulo conversion for unsigned 8-bit integers:

```

> const uint8 = new Uint8Array(1);
> uint8[0] = 255; uint8[0] // highest value within range
255
> uint8[0] = 256; uint8[0] // overflow
0
> uint8[0] = 0; uint8[0] // lowest value within range
0
> uint8[0] = -1; uint8[0] // underflow
255

```

Modulo conversion for signed 8-bit integers:

```

> const int8 = new Int8Array(1);
> int8[0] = 127; int8[0] // highest value within range
127
> int8[0] = 128; int8[0] // overflow
-128
> int8[0] = -128; int8[0] // lowest value within range
-128
> int8[0] = -129; int8[0] // underflow
127

```

Clamped conversion is different:

- All underflowing values are converted to the lowest value.
- All overflowing values are converted to the highest value.

```

> const uint8c = new Uint8ClampedArray(1);
> uint8c[0] = 255; uint8c[0] // highest value within range
255
> uint8c[0] = 256; uint8c[0] // overflow
255
> uint8c[0] = 0; uint8c[0] // lowest value within range
0
> uint8c[0] = -1; uint8c[0] // underflow
0

```

20.2.3 Endianness

Whenever a type (such as `Uint16`) is stored as multiple bytes, *endianness* matters:

- Big endian: the most significant byte comes first. For example, the `Uint16` value 0xABCD is stored as two bytes – first 0xAB, then 0xCD.
- Little endian: the least significant byte comes first. For example, the `Uint16` value 0xABCD is stored as two bytes – first 0xCD, then 0xAB.

Endianness tends to be fixed per CPU architecture and consistent across native APIs. Typed Arrays are used to communicate with those APIs, which is why their endianness follows the endianness of the platform and can't be changed.

On the other hand, the endianness of protocols and binary files varies and is fixed across platforms. Therefore, we must be able to access data with either endianness. DataViews serve this use case and let you specify endianness when you get or set a value.

[Quoting Wikipedia on Endianness:](#)

- Big-endian representation is the most common convention in data networking; fields in the protocols of the Internet protocol suite, such as IPv4, IPv6, TCP, and UDP, are transmitted in big-endian order. For this reason, big-endian byte order is also referred to as network byte order.
- Little-endian storage is popular for microprocessors in part due to significant historical influence on microprocessor designs by Intel Corporation.

You can use the following function to determine the endianness of a platform.

```

const BIG_ENDIAN = Symbol('BIG_ENDIAN');
const LITTLE_ENDIAN = Symbol('LITTLE_ENDIAN');
function getPlatformEndianness() {
  const arr32 = Uint32Array.of(0x12345678);
  const arr8 = new Uint8Array(arr32.buffer);
  switch ((arr8[0]*0x1000000) + (arr8[1]*0x10000) + (arr8[2]*0x100) + (arr8[3])) {
    case 0x12345678:
      return BIG_ENDIAN;
    case 0x78563412:
      return LITTLE_ENDIAN;
    default:
      throw new Error('Unknown endianness');
  }
}

```

There are also platforms that arrange *words* (pairs of bytes) with a different endianness than bytes inside words. That is called mixed endianness. Should you want to support such a platform then it is easy to extend the previous code.

20.2.4 Negative indices

With the bracket operator `[]`, you can only use non-negative indices (starting at 0). The methods of `ArrayBuffers`, `Typed Arrays` and `DataViews` work differently: every index can be negative. If it is, it counts backwards from the length. In other words, it is added to the length to produce a normal index. Therefore `-1` refers to the last element, `-2` to the second-last, etc. Methods of normal Arrays work the same way.

```
> const ui8 = Uint8Array.of(0, 1, 2);
> ui8.slice(-1)
Uint8Array [ 2 ]
```

Offsets, on the other hand, must be non-negative. If, for example, you pass `-1` to:

```
DataView.prototype.getInt8(byteOffset)
```

then you get a `RangeError`.

20.3 ArrayBuffers

`ArrayBuffers` store the data, *views* (`Typed Arrays` and `DataViews`) let you read and change it. In order to create a `DataView`, you need to provide its constructor with an `ArrayBuffer`. `Typed Array` constructors can optionally create an `ArrayBuffer` for you.

20.3.1 ArrayBuffer constructor

The signature of the constructor is:

```
ArrayBuffer(length : number)
```

Invoking this constructor via `new` creates an instance whose capacity is `length` bytes. Each of those bytes is initially 0.

20.3.2 Static ArrayBuffer methods

- `ArrayBuffer.isView(arg)`
Returns `true` if `arg` is an object and a view for an `ArrayBuffer`. Only `Typed Arrays` and `DataViews` have the required internal property `[[ViewedArrayBuffer]]`. That means that this check is roughly equivalent to checking whether `arg` is an instance of a `Typed Array` or of `DataView`.

20.3.3 ArrayBuffer.prototype properties

- `get ArrayBuffer.prototype.byteLength`
Returns the capacity of this `ArrayBuffer` in bytes.
- `ArrayBuffer.prototype.slice(start, end)`
Creates a new `ArrayBuffer` that contains the bytes of this `ArrayBuffer` whose indices are greater than or equal to `start` and less than `end`. `start` and `end` can be negative (see Sect. “[Negative indices](#)”).

20.4 Typed Arrays

The various kinds of `Typed Array` are only different w.r.t. to the type of their elements:

- `Typed Arrays` whose elements are integers: `Int8Array`, `Uint8Array`, `Uint8ClampedArray`, `Int16Array`, `Uint16Array`, `Int32Array`, `Uint32Array`
- `Typed Arrays` whose elements are floats: `Float32Array`, `Float64Array`

20.4.1 Typed Arrays versus normal Arrays

`Typed Arrays` are much like normal `Arrays`: they have a `length`, elements can be accessed via the bracket operator `[]` and they have all of the standard `Array` methods. They differ from `Arrays` in the following ways:

- All of their elements have the same type, setting elements converts values to that type.
- They are contiguous. Normal `Arrays` can have *holes* (indices in the range `[0, arr.length)` that have no associated element), `Typed Arrays` can't.
- **Initialized with zeros.** This is a consequence of the previous item:
 - `new Array(10)` creates a normal `Array` without any elements (it only has holes).
 - `new Uint8Array(10)` creates a `Typed Array` whose 10 elements are all 0.
- An associated buffer. The elements of a `Typed Array` `ta` are not stored in `ta`, they are stored in an associated `ArrayBuffer` that can be accessed via `ta.buffer`.

20.4.2 Typed Arrays are iterable

`Typed Arrays` implement a method whose key is `Symbol.iterator` and are therefore iterable (consult chapter “[Iterables and iterators](#)” for more information). That means that you can use the `for-of` loop and similar mechanisms in ES6:

```
const ui8 = Uint8Array.of(0,1,2);
for (const byte of ui8) {
  console.log(byte);
}
// Output:
// 0
// 1
// 2
```

`ArrayBuffers` and `DataViews` are not iterable.

20.4.3 Converting Typed Arrays to and from normal Arrays

To convert a normal Array to a Typed Array, you make it the parameter of a Typed Array constructor. For example:

```
> const tarr = new Uint8Array([0,1,2]);
```

The classic way to convert a Typed Array to an Array is to invoke `Array.prototype.slice` on it. This trick works for all Array-like objects (such as arguments) and Typed Arrays are Array-like.

```
> Array.prototype.slice.call(tarr)
[ 0, 1, 2 ]
```

In ES6, you can use the spread operator (`...`), because Typed Arrays are iterable:

```
> [...tarr]
[ 0, 1, 2 ]
```

Another ES6 alternative is `Array.from()`, which works with either iterables or Array-like objects:

```
> Array.from(tarr)
[ 0, 1, 2 ]
```

20.4.4 The Species pattern for Typed Arrays

Some methods create new instances that are similar to `this`. The species pattern lets you configure what constructor should be used to do so. For example, if you create a subclass `MyArray` of `Array` then the default is that `map()` creates instances of `MyArray`. If you want it to create instances of `Array`, you can use the species pattern to make that happen. Details are explained in Sect “The species pattern” in the chapter on classes.

`ArrayBuffers` use the species pattern in the following locations:

- `ArrayBuffer.prototype.slice()`
- Whenever an `ArrayBuffer` is cloned inside a Typed Array or `DataView`.

Typed Arrays use the species pattern in the following locations:

- `TypedArray<T>.prototype.filter()`
- `TypedArray<T>.prototype.map()`
- `TypedArray<T>.prototype.slice()`
- `TypedArray<T>.prototype.subarray()`

`DataViews` don't use the species pattern.

20.4.5 The inheritance hierarchy of Typed Arrays

As you could see in the diagram at the beginning of this chapter, all Typed Array classes (`Uint8Array` etc.) have a common superclass. I'm calling that superclass `TypedArray`, but it is not directly accessible from JavaScript (the ES6 specification calls it *the intrinsic object* `%TypedArray%`). `TypedArray.prototype` houses all methods of Typed Arrays.

20.4.6 Static `TypedArray` methods

Both static `TypedArray` methods are inherited by its subclasses (`Uint8Array` etc.).

20.4.6.1 `TypedArray.of()`

This method has the signature:

```
TypedArray.of(...items)
```

It creates a new Typed Array that is an instance of `this` (the class on which `of()` was invoked). The elements of that instance are the parameters of `of()`.

You can think of `of()` as a custom literal for Typed Arrays:

```
> Float32Array.of(0.151, -8, 3.7)
Float32Array [ 0.151, -8, 3.7 ]
```

20.4.6.2 `TypedArray.from()`

This method has the signature:

```
TypedArray<U>.from(source : Iterable<T>, mapfn? : T => U, thisArg?)
```

It converts the iterable `source` into an instance of `this` (a Typed Array).

For example, normal Arrays are iterable and can be converted with this method:

```
> Uint16Array.from([0, 1, 2])
Uint16Array [ 0, 1, 2 ]
```

Typed Arrays are iterable, too:

```
> const u16 = Uint16Array.from(Uint8Array.of(0, 1, 2));
> u16 instanceof Uint16Array
```

true

The optional `mapfn` lets you transform the elements of `source` before they become elements of the result. Why perform the two steps *mapping* and *conversion* in one go? Compared to performing the first step separately, via `source.map()`, there are two advantages:

1. No intermediate Array or Typed Array is needed.
2. When converting a Typed Array to a Typed Array whose elements have a higher precision, the mapping step can make use of that higher precision.

To illustrate the second advantage, let's use `map()` to double the elements of a Typed Array:

```
> Int8Array.of(127, 126, 125).map(x => 2 * x)
Int8Array [ -2, -4, -6 ]
```

As you can see, the values overflow and are coerced into the `Int8` range of values. If `map` via `from()`, you can choose the type of the result so that values don't overflow:

```
> Int16Array.from(Int8Array.of(127, 126, 125), x => 2 * x)
Int16Array [ 254, 252, 250 ]
```

According to Allen Wirfs-Brock, mapping between Typed Arrays was what motivated the `mapfn` parameter of `from()`.

20.4.7 TypedArray.prototype properties

Indices accepted by Typed Array methods can be negative (they work like traditional Array methods that way). Offsets must be non-negative. For details, see Sect. "Negative indices".

20.4.7.1 Methods specific to Typed Arrays

The following properties are specific to Typed Arrays, normal Arrays don't have them:

- `get TypedArray<T>.prototype.buffer : ArrayBuffer`
Returns the buffer backing this Typed Array.
- `get TypedArray<T>.prototype.byteLength : number`
Returns the size in bytes of this Typed Array's buffer.
- `get TypedArray<T>.prototype.byteOffset : number`
Returns the offset where this Typed Array "starts" inside its ArrayBuffer.
- `TypedArray<T>.prototype.set(arrayOrTypedArray, offset=0) : void`
Copies all elements of `arrayOrTypedArray` to this Typed Array. The element at index 0 of `arrayOrTypedArray` is written to index `offset` of this Typed Array (etc.).
 - If `arrayOrTypedArray` is a normal Array, its elements are converted to numbers who are then converted to the element type `T` of this Typed Array.
 - If `arrayOrTypedArray` is a Typed Array then each of its elements is converted directly to the appropriate type for this Typed Array. If both Typed Arrays have the same element type then faster, byte-wise copying is used.
- `TypedArray<T>.prototype.subarray(begin=0, end=this.length) : TypedArray<T>`
Returns a new Typed Array that has the same buffer as this Typed Array, but a (generally) smaller range. If `begin` is non-negative then the first element of the resulting Typed Array is `this[begin]`, the second `this[begin+1]` (etc.). If `begin` is negative, it is converted appropriately.

20.4.7.2 Array methods

The following methods are basically the same as the methods of normal Arrays:

- `TypedArray<T>.prototype.copyWithin(target : number, start : number, end = this.length) : This`
Copies the elements whose indices are between `start` (including) and `end` (excluding) to indices starting at `target`. If the ranges overlap and the former range comes first then elements are copied in reverse order to avoid overwriting source elements before they are copied.
- `TypedArray<T>.prototype.entries() : Iterable<[number,T]>`
Returns an iterable over `[index,element]` pairs for this Typed Array.
- `TypedArray<T>.prototype.every(callbackfn, thisArg?)`
Returns true if `callbackfn` returns true for every element of this Typed Array. Otherwise, it returns false. `every()` stops processing the first time `callbackfn` returns false.
- `TypedArray<T>.prototype.fill(value, start=0, end=this.length) : void`
Set the elements whose indices range from `start` to `end` to `value`.
- `TypedArray<T>.prototype.filter(callbackfn, thisArg?) : TypedArray<T>`
Returns a Typed Array that contains every element of this Typed Array for which `callbackfn` returns true. In general, the result is shorter than this Typed Array.
- `TypedArray<T>.prototype.find(predicate : T => boolean, thisArg?) : T`
Returns the first element for which the function `predicate` returns true.
- `TypedArray<T>.prototype.findIndex(predicate : T => boolean, thisArg?) : number`
Returns the index of the first element for which `predicate` returns true.

- `TypedArray<T>.prototype.forEach(callbackfn, thisArg?) : void`
Iterates over this Typed Array and invokes `callbackfn` for each element.
- `TypedArray<T>.prototype.indexOf(searchElement, fromIndex=0) : number`
Returns the index of the first element that strictly equals `searchElement`. The search starts at `fromIndex`.
- `TypedArray<T>.prototype.join(separator : string = ',') : string`
Converts all elements to strings and concatenates them, separated by `separator`.
- `TypedArray<T>.prototype.keys() : Iterable<number>`
Returns an iterable over the indices of this Typed Array.
- `TypedArray<T>.prototype.lastIndexOf(searchElement, fromIndex?) : number`
Returns the index of the last element that strictly equals `searchElement`. The search starts at `fromIndex`, backwards.
- `get TypedArray<T>.prototype.length : number`
Returns the length of this Typed Array.
- `TypedArray<T>.prototype.map(callbackfn, thisArg?) : TypedArray<T>`
Returns a new Typed Array in which every element is the result of applying `callbackfn` to the corresponding element of this Typed Array.
- `TypedArray<T>.prototype.reduce(callbackfn : (previousValue : any, currentElement : T, currentIndex : number, array : TypedArray<T>) => any, initialValue?) : any`
`callbackfn` is fed one element at a time, together with the result that was computed so far and computes a new result. Elements are visited from left to right.
- `TypedArray<T>.prototype.reduceRight(callbackfn : (previousValue : any, currentElement : T, currentIndex : number, array : TypedArray<T>) => any, initialValue?) : any`
`callbackfn` is fed one element at a time, together with the result that was computed so far and computes a new result. Elements are visited from right to left.
- `TypedArray<T>.prototype.reverse() : This`
Reverses the order of the elements of this Typed Array and returns this.
- `TypedArray<T>.prototype.slice(start=0, end=this.length) : TypedArray<T>`
Create a new Typed Array that only has the elements of this Typed Array whose indices are between `start` (including) and `end` (excluding).
- `TypedArray<T>.prototype.some(callbackfn, thisArg?)`
Returns true if `callbackfn` returns true for at least one element of this Typed Array. Otherwise, it returns false. `some()` stops processing the first time `callbackfn` returns true.
- `TypedArray<T>.prototype.sort(comparefn? : (number, number) => number)`
Sorts this Typed Array, as specified via `comparefn`. If `comparefn` is missing, sorting is done ascendingly, by comparing via the less-than operator (<).
- `TypedArray<T>.prototype.toLocaleString(reserved1?, reserved2?)`
- `TypedArray<T>.prototype.toString()`
- `TypedArray<T>.prototype.values() : Iterable<T>`
Returns an iterable over the values of this Typed Array.

Due to all of these methods being available for Arrays, you can consult the following two sources to find out more about how they work:

- The following methods are new in ES6 and explained in chapter “[New Array features](#)”: `copyWithin`, `entries`, `fill`, `find`, `findIndex`, `keys`, `values`.
- All other methods are explained in chapter “[Arrays](#)” of “[Speaking JavaScript](#)”.

Note that while normal Array methods are generic (any Array-like `this` is OK), the methods listed in this section are not (`this` must be a Typed Array).

20.4.8 «ElementType»Array constructor

Each Typed Array constructor has a name that follows the pattern «ElementType»Array, where «ElementType» is one of the element types in the table at the beginning. That means that there are 9 constructors for Typed Arrays: `Int8Array`, `Uint8Array`, `Uint8ClampedArray` (element type `Uint8C`), `Int16Array`, `Uint16Array`, `Int32Array`, `Uint32Array`, `Float32Array`, `Float64Array`.

Each constructor has five *overloaded* versions – it behaves differently depending on how many arguments it receives and what their types are:

- `«ElementType»Array(buffer, byteOffset=0, length?)`
Creates a new Typed Array whose buffer is `buffer`. It starts accessing the buffer at the given `byteOffset` and will have the given `length`. Note that `length` counts elements of the Typed Array (with 1–4 bytes each), not bytes.
- `«ElementType»Array(length)`
Creates a Typed Array with the given `length` and the appropriate buffer (whose size in bytes is `length * «ElementType»Array.BYTES_PER_ELEMENT`).
- `«ElementType»Array()`
Creates a Typed Array whose `length` is 0. It also creates an associated empty `ArrayBuffer`.
- `«ElementType»Array(typedArray)`
Creates a new Typed Array that has the same length and elements as `typedArray`. Values that are too large or small are converted appropriately.

- `«ElementType»Array(arrayLikeObject)`
Treats `arrayLikeObject` like an `Array` and creates a new `TypedArray` that has the same length and elements. Values that are too large or small are converted appropriately.

The following code shows three different ways of creating the same `Typed Array`:

```
const tarr1 = new Uint8Array([1,2,3]);
const tarr2 = Uint8Array.of(1,2,3);
const tarr3 = new Uint8Array(3);
tarr3[0] = 0;
tarr3[1] = 1;
tarr3[2] = 2;
```

20.4.9 Static `«ElementType»Array` properties

- `«ElementType»Array.BYTES_PER_ELEMENT`
Counts how many bytes are needed to store a single element:

```
> Uint8Array.BYTES_PER_ELEMENT
1
> Int16Array.BYTES_PER_ELEMENT
2
> Float64Array.BYTES_PER_ELEMENT
8
```

20.4.10 `«ElementType»Array.prototype` properties

- `«ElementType»Array.prototype.BYTES_PER_ELEMENT`
The same as `«ElementType»Array.BYTES_PER_ELEMENT`.

20.4.11 Concatenating Typed Arrays

`Typed Arrays` don't have a method `concat()`, like normal `Arrays` do. The work-around is to use the method

```
typedArray.set(arrayOrTypedArray, offset=0)
```

That method copies an existing `Typed Array` (or normal `Array`) into `typedArray` at index `offset`. Then you only have to make sure that `typedArray` is big enough to hold all (Typed) `Arrays` you want to concatenate:

```
function concatenate(resultConstructor, ...arrays) {
  let totalLength = 0;
  for (const arr of arrays) {
    totalLength += arr.length;
  }
  const result = new resultConstructor(totalLength);
  let offset = 0;
  for (const arr of arrays) {
    result.set(arr, offset);
    offset += arr.length;
  }
  return result;
}
console.log(concatenate(Uint8Array,
  Uint8Array.of(1, 2), Uint8Array.of(3, 4)));
// Uint8Array [1, 2, 3, 4]
```

20.5 DataViews

20.5.1 `DataView` constructor

- `DataView(buffer, byteOffset=0, byteLength=buffer.byteLength-byteOffset)`
Creates a new `DataView` whose data is stored in the `ArrayBuffer` `buffer`. By default, the new `DataView` can access all of `buffer`, the last two parameters allow you to change that.

20.5.2 `DataView.prototype` properties

- `get DataView.prototype.buffer`
Returns the `ArrayBuffer` of this `DataView`.
- `get DataView.prototype.byteLength`
Returns how many bytes can be accessed by this `DataView`.
- `get DataView.prototype.byteOffset`
Returns at which offset this `DataView` starts accessing the bytes in its buffer.
- `DataView.prototype.get«ElementType»(byteOffset, littleEndian=false)`
Reads a value from the buffer of this `DataView`.
 - `«ElementType»` can be: `Float32`, `Float64`, `Int8`, `Int16`, `Int32`, `Uint8`, `Uint16`, `Uint32`
- `DataView.prototype.set«ElementType»(byteOffset, value, littleEndian=false)`
Writes `value` to the buffer of this `DataView`.
 - `«ElementType»` can be: `Float32`, `Float64`, `Int8`, `Int16`, `Int32`, `Uint8`, `Uint16`, `Uint32`

20.6 Browser APIs that support Typed Arrays

Typed Arrays have been around for a while, so there are quite a few browser APIs that support them.

20.6.1 File API

The [file API](#) lets you access local files. The following code demonstrates how to get the bytes of a submitted local file in an `ArrayBuffer`.

```
const fileInput = document.getElementById('fileInput');
const file = fileInput.files[0];
const reader = new FileReader();
reader.readAsArrayBuffer(file);
reader.onload = function () {
    const arrayBuffer = reader.result;
    ...
};
```

20.6.2 XMLHttpRequest

In newer versions of the [XMLHttpRequest API](#), you can have the results delivered in an `ArrayBuffer`:

```
const xhr = new XMLHttpRequest();
xhr.open('GET', someUrl);
xhr.responseType = 'arraybuffer';

xhr.onload = function () {
    const arrayBuffer = xhr.response;
    ...
};

xhr.send();
```

20.6.3 Fetch API

Similarly to `XMLHttpRequest`, the [Fetch API](#) lets you request resources. But it is based on Promises, which makes it more convenient to use. The following code demonstrates how to download the content pointed to by `url` as an `ArrayBuffer`:

```
fetch(url)
.then(request => request.arrayBuffer())
.then(arrayBuffer => ...);
```

20.6.4 Canvas

Quoting the [HTML5 specification](#):

The `canvas` element provides scripts with a resolution-dependent bitmap canvas, which can be used for rendering graphs, game graphics, art, or other visual images on the fly.

The [2D Context of canvas](#) lets you retrieve the bitmap data as an instance of `Uint8ClampedArray`:

```
const canvas = document.getElementById('my_canvas');
const context = canvas.getContext('2d');
const imageData = context.getImageData(0, 0, canvas.width, canvas.height);
const uint8ClampedArray = imageData.data;
```

20.6.5 WebSockets

[WebSockets](#) let you send and receive binary data via `ArrayBuffers`:

```
const socket = new WebSocket('ws://127.0.0.1:8081');
socket.binaryType = 'arraybuffer';

// Wait until socket is open
socket.addEventListener('open', function (event) {
    // Send binary data
    const typedArray = new Uint8Array(4);
    socket.send(typedArray.buffer);
});

// Receive binary data
socket.addEventListener('message', function (event) {
    const arrayBuffer = event.data;
    ...
});
```

20.6.6 Other APIs

- [WebGL](#) uses the Typed Array API for: accessing buffer data, specifying pixels for texture mapping, reading pixel data, and more.
- The [Web Audio API](#) lets you [decode audio data](#) submitted via an `ArrayBuffer`.
- [Media Source Extensions](#): The HTML media elements are currently `<audio>` and `<video>`. The Media Source Extensions API enables you to create streams to be played via those elements. You can add binary data to such streams via `ArrayBuffers`, Typed Arrays or `DataViews`.

- Communication with [Web Workers](#): If you send data to a Worker via `postMessage()`, either the message (which will be cloned) or the transferable objects can contain `ArrayBuffers`.
- [Cross-document communication](#): works similarly to communication with Web Workers and also uses the method `postMessage()`.

20.7 Extended example: JPEG SOF0 decoder



The code of the following example is [on GitHub](#). And you can [run it online](#).

The example is a web pages that lets you upload a JPEG file and parses its structure to determine the height and the width of the image and more.

20.7.1 The JPEG file format

A JPEG file is a sequence of *segments* (typed data). Each segment starts with the following four bytes:

- Marker (two bytes): declares what kind of data is stored in the segment. The first of the two bytes is always `0xFF`. Each of the standard markers has a human readable name. For example, the marker `0xFFC0` has the name “Start Of Frame (Baseline DCT)”, short: “SOF0”.
- Length of segment (two bytes): how long is this segment (in bytes, including the length itself)?

JPEG files are big-endian on all platforms. Therefore, this example demonstrates how important it is that we can specify endianness when using `DataViews`.

20.7.2 The JavaScript code

The following function `processArrayBuffer()` is an abridged version of the actual code; I’ve removed a few error checks to reduce clutter. `processArrayBuffer()` receives an `ArrayBuffer` with the contents of the submitted JPEG file and iterates over its segments.

```
// JPEG is big endian
var IS_LITTLE_ENDIAN = false;

function processArrayBuffer(arrayBuffer) {
  try {
    var dv = new DataView(arrayBuffer);
    ...
    var ptr = 2;
    while (true) {
      ...
      var lastPtr = ptr;
      enforceValue(0xFF, dv.getUint8(ptr),
        'Not a marker');
      ptr++;
      var marker = dv.getUint8(ptr);
      ptr++;
      var len = dv.getUint16(ptr, IS_LITTLE_ENDIAN);
      ptr += len;
      logInfo('Marker: ' + hex(marker) + ' (' + len + ' byte(s)');
      ...

      // Did we find what we were looking for?
      if (marker === 0xC0) { // SOF0
        logInfo(decodeSOF0(dv, lastPtr));
        break;
      }
    }
  } catch (e) {
    logError(e.message);
  }
}
```

This code uses the following helper functions (that are not shown here):

- `enforceValue()` throws an error if the expected value (first parameter) doesn’t match the actual value (second parameter).
- `logInfo()` and `logError()` display messages on the page.
- `hex()` turns a number into a string with two hexadecimal digits.

`decodeSOF0()` parses the segment SOF0:

```
function decodeSOF0(dv, start) {
  // Example (16x16):
  // FF C0 00 11 08 00 10 00 10 03 01 22 00 02 11 01 03 11 01
  var data = {};
  start += 4; // skip marker 0xFFC0 and segment length 0x0011
  var data = {
    bitsPerColorComponent: dv.getUint8(start), // usually 0x08
    imageHeight: dv.getUint16(start+1, IS_LITTLE_ENDIAN),
    imageWidth: dv.getUint16(start+3, IS_LITTLE_ENDIAN),
    numberOfColorComponents: dv.getUint8(start+5),
  };
  return JSON.stringify(data, null, 4);
}
```

More information on the structure of JPEG files:

- [“JPEG: Syntax and structure”](#) (on Wikipedia)
- [“JPEG File Interchange Format: File format structure”](#) (on Wikipedia)

20.8 Availability

Much of the Typed Array API is implemented by all modern JavaScript engines, but several features are new to ECMAScript 6:

- Static methods borrowed from Arrays: `TypedArray<T>.from()`, `TypedArray<T>.of()`
- Prototype methods borrowed from Arrays: `TypedArray<T>.prototype.map()` etc.
- Typed Arrays are iterable
- Support for the species pattern
- An inheritance hierarchy where `TypedArray<T>` is the superclass of all Typed Array classes

It may take a while until these are available everywhere. As usual, kangax’ [“ES6 compatibility table”](#) describes the status quo.

21. Iterables and iterators

21.1 Overview

ES6 introduces a new mechanism for traversing data: *iteration*. Two concepts are central to iteration:

- An *iterable* is a data structure that wants to make its elements accessible to the public. It does so by implementing a method whose key is `Symbol.iterator`. That method is a factory for *iterators*.
- An *iterator* is a pointer for traversing the elements of a data structure (think cursors in databases).

Expressed as interfaces in TypeScript notation, these roles look like this:

```
interface Iterable {
  [Symbol.iterator]() : Iterator;
}
interface Iterator {
  next() : IteratorResult;
}
interface IteratorResult {
  value: any;
  done: boolean;
}
```

21.1.1 Iterable values

The following values are iterable:

- Arrays
- Strings
- Maps
- Sets
- DOM data structures (work in progress)

Plain objects are not iterable (why is explained in [a dedicated section](#)).

21.1.2 Constructs supporting iteration

Language constructs that access data via iteration:

- Destructuring via an Array pattern:

```
const [a,b] = new Set(['a', 'b', 'c']);
```

- for-of loop:

```
for (const x of ['a', 'b', 'c']) {
  console.log(x);
}
```

- `Array.from()`:

```
const arr = Array.from(new Set(['a', 'b', 'c']));
```

- Spread operator (`...`):

```
const arr = [...new Set(['a', 'b', 'c'])];
```

- Constructors of Maps and Sets:

```
const map = new Map([[false, 'no'], [true, 'yes']]);
const set = new Set(['a', 'b', 'c']);
```

- `Promise.all()`, `Promise.race()`:


```

Promise.all(iterableOverPromises).then(...);
Promise.race(iterableOverPromises).then(...);

```
- `yield*`:


```

yield* anIterable;

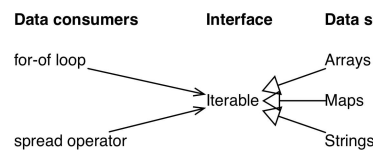
```

21.2 Iterability

The idea of iterability is as follows.

- **Data consumers:** JavaScript has language constructs that consume data. For example, `for-of` loops over values and the spread operator `(...)` inserts values into Arrays or function calls.
- **Data sources:** The data consumers could get their values from a variety of sources. For example, you may want to iterate over the elements of an Array, the key-value entries in a Map or the characters of a string.

It's not practical for every consumer to support all sources, especially because it should be possible to create new sources (e.g. via libraries). Therefore, ES6 introduces the interface `Iterable`. Data consumers use it, data sources implement it:



Given that JavaScript does not have interfaces, `Iterable` is more of a convention:

- **Source:** A value is considered *iterable* if it has a method whose key is the symbol `Symbol.iterator` that returns a so-called *iterator*. The iterator is an object that returns values via its method `next()`. We say: it *iterates over* the *items* (the content) of the iterable, one per method call.
- **Consumption:** Data consumers use the iterator to retrieve the values they are consuming.

Let's see what consumption looks like for an Array `arr`. First, you create an iterator via the method whose key is `Symbol.iterator`:

```

> const arr = ['a', 'b', 'c'];
> const iter = arr[Symbol.iterator]();

```

Then you call the iterator's method `next()` repeatedly to retrieve the items "inside" the Array:

```

> iter.next()
{ value: 'a', done: false }
> iter.next()
{ value: 'b', done: false }
> iter.next()
{ value: 'c', done: false }
> iter.next()
{ value: undefined, done: true }

```

As you can see, `next()` returns each item wrapped in an object, as the value of the property `value`. The boolean property `done` indicates when the end of the sequence of items has been reached.

`Iterable` and iterators are part of a so-called *protocol* (interfaces plus rules for using them) for iteration. A key characteristic of this protocol is that it is sequential: the iterator returns values one at a time. That means that if an iterable data structure is non-linear (such as a tree), iteration will linearize it.

21.3 Iterable data sources

I'll use the `for-of` loop (see Chap. "The `for-of` loop") to iterate over various kinds of iterable data.

21.3.1 Arrays

Arrays (and Typed Arrays) are iterables over their elements:

```

for (const x of ['a', 'b']) {
  console.log(x);
}
// Output:
// 'a'
// 'b'

```

21.3.2 Strings

Strings are iterable, but they iterate over Unicode code points, each of which may comprise one or two JavaScript characters:

```
for (const x of 'a\uD83D\uDC0A') {
  console.log(x);
}
// Output:
// 'a'
// '\uD83D\uDC0A' (crocodile emoji)
```



You have just seen that primitive values can be iterable. A value doesn't have to be an object in order to be iterable. That's because all values are coerced to objects before the iterator method (property key `Symbol.iterator`) is accessed.

21.3.3 Maps

Maps are iterables over their entries. Each entry is encoded as a [key, value] pair, an Array with two elements. The entries are always iterated over deterministically, in the same order in which they were added to the map.

```
const map = new Map().set('a', 1).set('b', 2);
for (const pair of map) {
  console.log(pair);
}
// Output:
// ['a', 1]
// ['b', 2]
```

Note that WeakMaps are not iterable.

21.3.4 Sets

Sets are iterables over their elements (which are iterated over in the same order in which they were added to the Set).

```
const set = new Set().add('a').add('b');
for (const x of set) {
  console.log(x);
}
// Output:
// 'a'
// 'b'
```

Note that WeakSets are not iterable.

21.3.5 arguments

Even though the special variable `arguments` is more or less obsolete in ECMAScript 6 (due to rest parameters), it is iterable:

```
function printArgs() {
  for (const x of arguments) {
    console.log(x);
  }
}
printArgs('a', 'b');

// Output:
// 'a'
// 'b'
```

21.3.6 DOM data structures

Most DOM data structures will eventually be iterable:

```
for (const node of document.querySelectorAll('div')) {
  ...
}
```

Note that implementing this functionality is work in progress. But it is relatively easy to do so, because the symbol `Symbol.iterator` can't clash with existing property keys.

21.3.7 Iterable computed data

Not all iterable content does have to come from data structures, it could also be computed on the fly. For example, all major ES6 data structures (Arrays, Typed Arrays, Maps, Sets) have three methods that return iterable objects:

- `entries()` returns an iterable over entries encoded as [key, value] Arrays. For Arrays, the values are the Array elements and the keys are their indices. For Sets, each key and value are the same – the Set element.
- `keys()` returns an iterable over the keys of the entries.
- `values()` returns an iterable over the values of the entries.

Let's see what that looks like. `entries()` gives you a nice way to get both Array elements and their indices:

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

```

}
// Output:
// [0, 'a']
// [1, 'b']
// [2, 'c']

```

21.3.8 Plain objects are not iterable

Plain objects (as created by object literals) are not iterable:

```

for (const x of {}) { // TypeError
  console.log(x);
}

```

Why aren't objects iterable over properties, by default? The reasoning is as follows. There are two levels at which you can iterate in JavaScript:

1. The program level: iterating over properties means examining the structure of the program.
2. The data level: iterating over a data structure means examining the data managed by the program.

Making iteration over properties the default would mean mixing those levels, which would have two disadvantages:

- You can't iterate over the properties of data structures.
- Once you iterate over the properties of an object, turning that object into a data structure would break your code.

If engines were to implement iterability via a method

`Object.prototype[Symbol.iterator]()` then there would be an additional caveat: Objects created via `Object.create(null)` wouldn't be iterable, because `Object.prototype` is not in their prototype chain.

It is important to remember that iterating over the properties of an object is mainly interesting if you use objects as Maps¹. But we only do that in ES5 because we have no better alternative. In ECMAScript 6, we have the built-in data structure `Map`.

21.3.8.1 How to iterate over properties

The proper (and safe) way to iterate over properties is via a tool function. For example, via `objectEntries()`, whose implementation is shown later (future ECMAScript versions may have something similar built in):

```

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

for (const [key,value] of objectEntries(obj)) {
  console.log(` ${key}: ${value}`);
}

// Output:
// first: Jane
// last: Doe

```

21.4 Iterating language constructs

The following ES6 language constructs make use of the iteration protocol:

- Destructuring via an Array pattern
- `for-of` loop
- `Array.from()`
- Spread operator (`...`)
- Constructors of Maps and Sets
- `Promise.all()`, `Promise.race()`
- `yield*`

The next sections describe each one of them in detail.

21.4.1 Destructuring via an Array pattern

Destructuring via Array patterns works for any iterable:

```

const set = new Set().add('a').add('b').add('c');

const [x,y] = set;
// x='a'; y='b'

const [first, ...rest] = set;
// first='a'; rest=['b','c'];

```

21.4.2 The `for-of` loop

`for-of` is a new loop in ECMAScript 6. It's basic form looks like this:

```

for (const x of iterable) {
  ...
}

```

For more information, consult Chap. “The `for-of` loop”.

Note that the iterability of `iterable` is required, otherwise `for-of` can't loop over a value. That means that non-iterable values must be converted to something iterable. For example, via `Array.from()`.

21.4.3 `Array.from()`

`Array.from()` converts iterable and Array-like values to Arrays. It is also available for typed Arrays.

```
> Array.from(new Map().set(false, 'no').set(true, 'yes'))
[[false,'no'], [true,'yes']]
> Array.from({ length: 2, 0: 'hello', 1: 'world' })
['hello', 'world']
```

For more information on `Array.from()`, consult [the chapter on Arrays](#).

21.4.4 The spread operator (`...`)

The spread operator inserts the values of an iterable into an Array:

```
> const arr = ['b', 'c'];
> ['a', ...arr, 'd']
['a', 'b', 'c', 'd']
```

That means that it provides you with a compact way to convert any iterable to an Array:

```
const arr = [...iterable];
```

The spread operator also turns an iterable into the arguments of a function, method or constructor call:

```
> Math.max(...[-1, 8, 3])
8
```

21.4.5 Maps and Sets

The constructor of a Map turns an iterable over `[key, value]` pairs into a Map:

```
> const map = new Map([[ 'uno', 'one'], [ 'dos', 'two']]);
> map.get('uno')
'one'
> map.get('dos')
'two'
```

The constructor of a Set turns an iterable over elements into a Set:

```
> const set = new Set(['red', 'green', 'blue']);
> set.has('red')
true
> set.has('yellow')
false
```

The constructors of `WeakMap` and `WeakSet` work similarly. Furthermore, Maps and Sets are iterable themselves (`WeakMaps` and `WeakSets` aren't), which means that you can use their constructors to clone them.

21.4.6 Promises

`Promise.all()` and `Promise.race()` accept iterables over Promises:

```
Promise.all(iterableOverPromises).then(...);
Promise.race(iterableOverPromises).then(...);
```

21.4.7 `yield*`

`yield*` is an operator that is only available inside generators. It yields all items iterated over by an iterable.

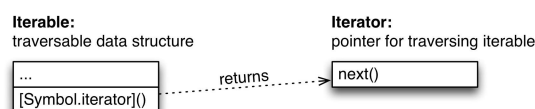
```
function* yieldAllValuesOf(iterable) {
  yield* iterable;
}
```

The most important use case for `yield*` is to recursively call a generator (which produces something iterable).

21.5 Implementing iterables

In this section, I explain in detail how to implement iterables. Note that [ES6 generators](#) are usually much more convenient for this task than doing so “manually”.

The iteration protocol looks as follows.



An object becomes *iterable* (“implements” the interface `Iterable`) if it has a method (own or inherited) whose key is `Symbol.iterator`. That method must return an *iterator*, an object that *iterates over the items* “inside” the iterable via its method `next()`.

In TypeScript notation, the interfaces for iterables and iterators look as follows².

```
interface Iterable {
  [Symbol.iterator]() : Iterator;
}
interface Iterator {
  next() : IteratorResult;
  return?(value? : any) : IteratorResult;
}
interface IteratorResult {
  value: any;
  done: boolean;
}
```

`return()` is an optional method that we’ll get to later³. Let’s first implement a dummy iterable to get a feeling for how iteration works.

```
const iterable = {
  [Symbol.iterator]() {
    let step = 0;
    const iterator = {
      next() {
        if (step <= 2) {
          step++;
        }
        switch (step) {
          case 1:
            return { value: 'hello', done: false };
          case 2:
            return { value: 'world', done: false };
          default:
            return { value: undefined, done: true };
        }
      }
    };
    return iterator;
  }
};
```

Let’s check that `iterable` is, in fact, iterable:

```
for (const x of iterable) {
  console.log(x);
}
// Output:
// hello
// world
```

The code executes three steps, with the counter `step` ensuring that everything happens in the right order. First, we return the value `'hello'`, then the value `'world'` and then we indicate that the end of the iteration has been reached. Each item is wrapped in an object with the properties:

- `value` which holds the actual item and
- `done` which is a boolean flag that indicates whether the end has been reached, yet.

You can omit `done` if it is `false` and `value` if it is `undefined`. That is, the `switch` statement could be written as follows.

```
switch (step) {
  case 1:
    return { value: 'hello' };
  case 2:
    return { value: 'world' };
  default:
    return { done: true };
}
```

As is explained in the [blue chapter on generators](#), there are cases where you want even the last item with `done: true` to have a value. Otherwise, `next()` could be simpler and return items directly (without wrapping them in objects). The end of iteration would then be indicated via a special value (e.g., a symbol).

Let’s look at one more implementation of an iterable. The function `iterateOver()` returns an iterable over the arguments that are passed to it:

```
function iterateOver(...args) {
  let index = 0;
  const iterable = {
    [Symbol.iterator]() {
      const iterator = {
        next() {
          if (index < args.length) {
            return { value: args[index++] };
          } else {
            return { done: true };
          }
        }
      };
      return iterator;
    }
  };
  return iterable;
}
```

```
// Using `iterateOver()`:
for (const x of iterateOver('fee', 'fi', 'fo', 'fum')) {
  console.log(x);
}

// Output:
// fee
// fi
// fo
// fum
```

21.5.1 Iterators that are iterable

The previous function can be simplified if the iterable and the iterator are the same object:

```
function iterateOver(...args) {
  let index = 0;
  const iterable = {
    [Symbol.iterator]() {
      return this;
    },
    next() {
      if (index < args.length) {
        return { value: args[index++] };
      } else {
        return { done: true };
      }
    },
  };
  return iterable;
}
```

Even if the original iterable and the iterator are not the same object, it is still occasionally useful if an iterator has the following method (which also makes it an iterable):

```
[Symbol.iterator]() {
  return this;
}
```

All built-in ES6 iterators follow this pattern (via a common prototype, see [the chapter on generators](#)). For example, the default iterator for Arrays:

```
> const arr = [];
> const iterator = arr[Symbol.iterator]();
> iterator[Symbol.iterator]() === iterator
true
```

Why is it useful if an iterator is also an iterable? `for-of` only works for iterables, not for iterators. Because Array iterators are iterable, you can continue an iteration in another loop:

```
const arr = ['a', 'b'];
const iterator = arr[Symbol.iterator]();

for (const x of iterator) {
  console.log(x); // a
  break;
}

// Continue with same iterator:
for (const x of iterator) {
  console.log(x); // b
}
```

One use case for continuing an iteration is that you can remove initial items (e.g. a header) before processing the actual content via `for-of`.

21.5.2 Optional iterator methods: `return()` and `throw()`

Two iterator methods are optional:

- `return()` gives an iterator the opportunity to clean up if an iteration ends prematurely.
- `throw()` is about forwarding a method call to a generator that is iterated over via `yield*`. It is explained in [the chapter on generators](#).

21.5.2.1 Closing iterators via `return()`

As mentioned before, the optional iterator method `return()` is about letting an iterator clean up if it wasn't iterated over until the end. It *closes* an iterator. In `for-of` loops, premature (or *abrupt*, in spec language) termination can be caused by:

- `break`
- `continue` (if you continue an outer loop, `continue` acts like a `break`)
- `throw`
- `return`

In each of these cases, `for-of` lets the iterator know that the loop won't finish. Let's look at an example, a function `readLinesSync` that returns an iterable of text lines in a file and would like to close that file no matter what happens:


```
function readLinesSync(fileName) {
  const file = ...;
  return {
    ...
    next() {
      if (file.isAtEndOfFile()) {
        file.close();
        return { done: true };
      }
      ...
    },
    return() {
      file.close();
      return { done: true };
    },
  };
}
```

Due to `return()`, the file will be properly closed in the following loop:

```
// Only print first line
for (const line of readLinesSync(fileName)) {
  console.log(x);
  break;
}
```

The `return()` method must return an object. That is due to how generators handle the `return` statement and will be explained in [the chapter on generators](#).

The following constructs close iterators that aren't completely "drained":

- `for-of`
- `yield*`
- **Destructuring**
- `Array.from()`
- `Map()`, `Set()`, `WeakMap()`, `WeakSet()`
- `Promise.all()`, `Promise.race()`

A [later section](#) has more information on closing iterators.

21.6 More examples of iterables

In this section, we look at a few more examples of iterables. Most of these iterables are easier to implement via generators. [The chapter on generators](#) shows how.

21.6.1 Tool functions that return iterables

Tool functions and methods that return iterables are just as important as iterable data structures. The following is a tool function for iterating over the own properties of an object.

```
function objectEntries(obj) {
  let index = 0;

  // In ES6, you can use strings or symbols as property keys,
  // Reflect.ownKeys() retrieves both
  const propKeys = Reflect.ownKeys(obj);

  return {
    [Symbol.iterator]() {
      return this;
    },
    next() {
      if (index < propKeys.length) {
        const key = propKeys[index];
        index++;
        return { value: [key, obj[key]] };
      } else {
        return { done: true };
      }
    }
  };
}

const obj = { first: 'Jane', last: 'Doe' };
for (const [key,value] of objectEntries(obj)) {
  console.log(`${key}: ${value}`);
}

// Output:
// first: Jane
// last: Doe
```

Another option is to use an iterator instead of an index to traverse the Array with the property keys:

```
function objectEntries(obj) {
  let iter = Reflect.ownKeys(obj)[Symbol.iterator]();

  return {
    [Symbol.iterator]() {
      return this;
    },
    next() {
      let { done, value: key } = iter.next();
      if (done) {
        return { done: true };
      }
      return { value: [key, obj[key]] };
    }
  };
}
```

```

    }
  };
}

```

21.6.2 Combinators for iterables

*Combinators*⁴ are functions that combine existing iterables to create new ones.

21.6.2.1 take(n, iterable)

Let's start with the combinator function `take(n, iterable)`, which returns an iterable over the first `n` items of `iterable`.

```

function take(n, iterable) {
  const iter = iterable[Symbol.iterator]();
  return {
    [Symbol.iterator]() {
      return this;
    },
    next() {
      if (n > 0) {
        n--;
        return iter.next();
      } else {
        return { done: true };
      }
    }
  };
}

const arr = ['a', 'b', 'c', 'd'];
for (const x of take(2, arr)) {
  console.log(x);
}

// Output:
// a
// b

```



This version of `take()` doesn't close the iterator `iter`. How to do that is shown later, after I explain what closing an iterator actually means.

21.6.2.2 zip(...iterables)

`zip` turns `n` iterables into an iterable of `n`-tuples (encoded as Arrays of length `n`).

```

function zip(...iterables) {
  const iterators = iterables.map(i => i[Symbol.iterator]());
  let done = false;
  return {
    [Symbol.iterator]() {
      return this;
    },
    next() {
      if (!done) {
        const items = iterators.map(i => i.next());
        done = items.some(item => item.done);
        if (!done) {
          return { value: items.map(i => i.value) };
        }
        // Done for the first time: close all iterators
        for (const iterator of iterators) {
          if (typeof iterator.return === 'function') {
            iterator.return();
          }
        }
      }
      // We are done
      return { done: true };
    }
  };
}

```

As you can see, the shortest iterable determines the length of the result:

```

const zipped = zip(['a', 'b', 'c'], ['d', 'e', 'f', 'g']);
for (const x of zipped) {
  console.log(x);
}

// Output:
// ['a', 'd']
// ['b', 'e']
// ['c', 'f']

```

21.6.3 Infinite iterables

Some iterable may never be done.

```

function naturalNumbers() {
  let n = 0;
  return {
    [Symbol.iterator]() {
      return this;
    },
    next() {
      return { value: n++ };
    }
  };
}

```

```
}
}
```

With an infinite iterable, you must not iterate over “all” of it. For example, by breaking from a `for-of` loop:

```
for (const x of naturalNumbers()) {
  if (x > 2) break;
  console.log(x);
}
```

Or by only accessing the beginning of an infinite iterable:

```
const [a, b, c] = naturalNumbers();
// a=0; b=1; c=2;
```

Or by using a combinator. `take()` is one possibility:

```
for (const x of take(3, naturalNumbers())) {
  console.log(x);
}
// Output:
// 0
// 1
// 2
```

The “length” of the iterable returned by `zip()` is determined by its shortest input iterable. That means that `zip()` and `naturalNumbers()` provide you with the means to number iterables of arbitrary (finite) length:

```
const zipped = zip(['a', 'b', 'c'], naturalNumbers());
for (const x of zipped) {
  console.log(x);
}
// Output:
// ['a', 0]
// ['b', 1]
// ['c', 2]
```

21.7 FAQ: iterables and iterators

21.7.1 Isn’t the iteration protocol slow?

You may be worried about the iteration protocol being slow, because a new object is created for each invocation of `next()`. However, memory management for small objects is fast in modern engines and in the long run, engines can optimize iteration so that no intermediate objects need to be allocated. A [thread on es-discuss](#) has more information.

21.7.2 Can I reuse the same object several times?

In principle, nothing prevents an iterator from reusing the same iteration result object several times – I’d expect most things to work well. However, there will be problems if a client caches iteration results:

```
const iterationResults = [];
const iterator = iterable[Symbol.iterator]();
let iterationResult;
while (!(iterationResult = iterator.next()).done) {
  iterationResults.push(iterationResult);
}
```

If an iterator reuses its iteration result object, `iterationResults` will, in general, contain the same object multiple times.

21.7.3 Why doesn’t ECMAScript 6 have iterable combinators?

You may be wondering why ECMAScript 6 does not have *iterable combinators*, tools for working with iterables or for creating iterables. That is because the plans are to proceed in two steps:

- Step 1: standardize an iteration protocol.
- Step 2: wait for libraries based on that protocol.

Eventually, one such library or pieces from several libraries will be added to the JavaScript standard library.

If you want to get an impression of what such a library could look like, take a look at the standard Python module [itertools](#).

21.7.4 Aren’t iterables difficult to implement?

Yes, iterables are difficult to implement – if you implement them manually. [The next chapter](#) will introduce *generators* that help with this task (among other things).

21.8 The ECMAScript 6 iteration protocol in depth

The iteration protocol comprises the following interfaces (I have omitted `throw()` from `Iterator`, which is only supported by `yield*` and optional there):

```

interface Iterable {
  [Symbol.iterator]() : Iterator;
}
interface Iterator {
  next() : IteratorResult;
  return?(value? : any) : IteratorResult;
}
interface IteratorResult {
  value : any;
  done : boolean;
}

```



The spec has [a section on the iteration protocol](#).

21.8.1 Iteration

Rules for `next()`:

- As long as the iterator still has values `x` to produce, `next()` returns objects {
value: `x`, done: `false` }.
- After the last value was iterated over, `next()` should always return an object whose property `done` is `true`.

21.8.1.1 The `IteratorResult`

The property `done` of an iterator result doesn't have to be `true` or `false`, `truthy` or `falsy` is enough. All built-in language mechanisms let you omit `done: false`.

21.8.1.2 Iterables that return fresh iterators versus those that always return the same iterator

Some iterables produce a new iterator each time they are asked for one. For example, Arrays:

```

function getIterator(iterable) {
  return iterable[Symbol.iterator]();
}

const iterable = ['a', 'b'];
console.log(getIterator(iterable) === getIterator(iterable)); // false

```

Other iterables return the same iterator each time. For example, generator objects:

```

function* elements() {
  yield 'a';
  yield 'b';
}

const iterable = elements();
console.log(getIterator(iterable) === getIterator(iterable)); // true

```

Whether an iterable produces a fresh iterators or not matter when you iterate over the same iterable multiple times. For example, via the following function:

```

function iterateTwice(iterable) {
  for (const x of iterable) {
    console.log(x);
  }
  for (const x of iterable) {
    console.log(x);
  }
}

```

With fresh iterators, you can iterate over the same iterable multiple times:

```

iterateTwice(['a', 'b']);
// Output:
// a
// b
// a
// b

```

If the same iterator is returned each time, you can't:

```

iterateTwice(elements());
// Output:
// a
// b

```

Note that each iterator in the standard library is also an iterable. Its method `[Symbol.iterator]()` return `this`, meaning that it always returns the same iterator (itself).

21.8.2 Closing iterators

The iteration protocol distinguishes two ways of finishing an iterator:

- **Exhaustion:** the regular way of finishing an iterator is by retrieving all of its values. That is, one calls `next()` until it returns an object whose property `done` is `true`.
- **Closing:** by calling `return()`, you tell the iterator that you don't intend to call `next()`, anymore.

Rules for calling `return()`:

- `return()` is an optional method, not all iterators have it. Iterators that do have it are called *closable*.
- `return()` should only be called if an iterator hasn't be exhausted. For example, `for-of` calls `return()` whenever it is left "abruptly" (before it is finished). The following operations cause abrupt exits: `break`, `continue` (with a label of an outer block), `return`, `throw`.

Rules for implementing `return()`:

- The method call `return(x)` should normally produce the object `{ done: true, value: x }`, but language mechanisms only throw an error ([source in spec](#)) if the result isn't an object.
- After `return()` was called, the objects returned by `next()` should be `done`, too.

The following code illustrates that the `for-of` loop calls `return()` if it is aborted before it receives a `done` iterator result. That is, `return()` is even called if you abort after receiving the last value. This is subtle and you have to be careful to get it right when you iterate manually or implement iterators.

```
function createIterable() {
  let done = false;
  const iterable = {
    [Symbol.iterator]() {
      return this;
    },
    next() {
      if (!done) {
        done = true;
        return { done: false, value: 'a' };
      } else {
        return { done: true, value: undefined };
      }
    },
    return() {
      console.log('return() was called!');
    },
  };
  return iterable;
}
for (const x of createIterable()) {
  console.log(x);
  // There is only one value in the iterable and
  // we abort the loop after receiving it
  break;
}
// Output:
// a
// return() was called!
```

21.8.2.1 Closable iterators

An iterator is *closable* if it has a method `return()`. Not all iterators are closable. For example, Array iterators are not:

```
> let iterable = ['a', 'b', 'c'];
> const iterator = iterable[Symbol.iterator]();
> 'return' in iterator
false
```

Generator objects are closable by default. For example, the ones returned by the following generator function:

```
function* elements() {
  yield 'a';
  yield 'b';
  yield 'c';
}
```

If you invoke `return()` on the result of `elements()`, iteration is finished:

```
> const iterator = elements();
> iterator.next()
{ value: 'a', done: false }
> iterator.return()
{ value: undefined, done: true }
> iterator.next()
{ value: undefined, done: true }
```

If an iterator is not closable, you can continue iterating over it after an abrupt exit (such as the one in line A) from a `for-of` loop:

```
function twoLoops(iterator) {
  for (const x of iterator) {
    console.log(x);
    break; // (A)
  }
  for (const x of iterator) {
    console.log(x);
  }
}
function getIterator(iterable) {
  return iterable[Symbol.iterator]();
}
twoLoops(getIterator(['a', 'b', 'c']));
// Output:
```

```
// a
// b
// c
```

Conversely, `elements()` returns a closable iterator and the second loop inside `twoLoops()` doesn't have anything to iterate over:

```
twoLoops(elements());
// Output:
// a
```

21.8.2.2 Preventing iterators from being closed

The following class is a generic solution for preventing iterators from being closed. It does so by wrapping the iterator and forwarding all method calls except `return()`.

```
class PreventReturn {
  constructor(iterator) {
    this.iterator = iterator;
  }
  /** Must also be iterable, so that for-of works */
  [Symbol.iterator]() {
    return this;
  }
  next() {
    return this.iterator.next();
  }
  return(value = undefined) {
    return { done: false, value };
  }
  // Not relevant for iterators: `throw()`
}
```

If we use `PreventReturn`, the result of the generator `elements()` won't be closed after the abrupt exit in the first loop of `twoLoops()`.

```
function* elements() {
  yield 'a';
  yield 'b';
  yield 'c';
}

function twoLoops(iterator) {
  for (const x of iterator) {
    console.log(x);
    break; // abrupt exit
  }
  for (const x of iterator) {
    console.log(x);
  }
}

twoLoops(elements());
// Output:
// a

twoLoops(new PreventReturn(elements()));
// Output:
// a
// b
// c
```

There is another way of making generators unclosable: All generator objects produced by the generator function `elements()` have the prototype object `elements.prototype`. Via `elements.prototype`, you can hide the default implementation of `return()` (which resides in a prototype of `elements.prototype`) as follows:

```
// Make generator object unclosable
// Warning: may not work in transpilers
elements.prototype.return = undefined;

twoLoops(elements());
// Output:
// a
// b
// c
```

21.8.2.3 Handling clean-up in generators via try-finally

Some generators need to clean up (release allocated resources, close open files, etc.) after iteration over them is finished. Naively, this is how we'd implement it:

```
function* genFunc() {
  yield 'a';
  yield 'b';

  console.log('Performing cleanup');
}
```

In a normal `for-of` loop, everything is fine:

```
for (const x of genFunc()) {
  console.log(x);
}
// Output:
// a
// b
// Performing cleanup
```

However, if you exit the loop after the first `yield`, execution seemingly pauses there forever and never reaches the cleanup step:

```

for (const x of genFunc()) {
  console.log(x);
  break;
}
// Output:
// a

```

What actually happens is that, whenever one leaves a `for-of` loop early, `for-of` sends a `return()` to the current iterator. That means that the cleanup step isn't reached because the generator function returns beforehand.

Thankfully, this is easily fixed, by performing the cleanup in a `finally` clause:

```

function* genFunc() {
  try {
    yield 'a';
    yield 'b';
  } finally {
    console.log('Performing cleanup');
  }
}

```

Now everything works as desired:

```

for (const x of genFunc()) {
  console.log(x);
  break;
}
// Output:
// a
// Performing cleanup

```

The general pattern for using resources that need to be closed or cleaned up in some manner is therefore:

```

function* funcThatUsesResource() {
  const resource = allocateResource();
  try {
    ...
  } finally {
    resource.deallocate();
  }
}

```

21.8.2.4 Handling clean-up in manually implemented iterators

```

const iterable = {
  [Symbol.iterator]() {
    function hasNextValue() { ... }
    function getNextValue() { ... }
    function cleanUp() { ... }
    let returnedDoneResult = false;
    return {
      next() {
        if (hasNextValue()) {
          const value = getNextValue();
          return { done: false, value: value };
        } else {
          if (!returnedDoneResult) {
            // Client receives first `done` iterator result
            // => won't call `return()`
            cleanUp();
            returnedDoneResult = true;
          }
          return { done: true, value: undefined };
        }
      },
      return() {
        cleanUp();
      }
    };
  }
};

```

Note that you must call `cleanUp()` when you are going to return a `done` iterator result for the first time. You must not do it earlier, because then `return()` may still be called. This can be tricky to get right.

21.8.2.5 Closing iterators you use

If you use iterators, you should close them properly. In generators, you can let `for-of` do all the work for you:

```

/**
 * Converts a (potentially infinite) sequence of
 * iterated values into a sequence of length `n`
 */
function* take(n, iterable) {
  for (const x of iterable) {
    if (n <= 0) {
      break; // closes iterable
    }
    n--;
    yield x;
  }
}

```

If you manage things manually, more work is required:

```
function* take(n, iterable) {
  const iterator = iterable[Symbol.iterator]();
  while (true) {
    const {value, done} = iterator.next();
    if (done) break; // exhausted
    if (n <= 0) {
      // Abrupt exit
      maybeCloseIterator(iterator);
      break;
    }
    yield value;
    n--;
  }
}

function maybeCloseIterator(iterator) {
  if (typeof iterator.return === 'function') {
    iterator.return();
  }
}
```

Even more work is necessary if you don't use generators:

```
function take(n, iterable) {
  const iter = iterable[Symbol.iterator]();
  return {
    [Symbol.iterator]() {
      return this;
    },
    next() {
      if (n > 0) {
        n--;
        return iter.next();
      } else {
        maybeCloseIterator(iter);
        return { done: true };
      }
    },
    return() {
      n = 0;
      maybeCloseIterator(iter);
    }
  };
}
```

21.8.3 Checklist

- Documenting an iterable: provide the following information.
 - Does it return fresh iterators or the same iterator each time?
 - Are its iterators closable?
- Implementing an iterator:
 - Clean-up activity must happen if either an iterator is exhausted or if `return()` is called.
 - In generators, `try-finally` lets you handle both in a single location.
 - After an iterator was closed via `return()`, it should not produce any more iterator results via `next()`.
- Using an iterator manually (versus via `for-of` etc.):
 - Don't forget to close the iterator via `return`, if – and only if – you don't exhaust it. Getting this right can be tricky.
- Continuing to iterate over an iterator after an abrupt exit: The iterator must either be unclosable or made unclosable (e.g. via a tool class).

22. Generators



The following GitHub repository contains the example code: [generator-examples](#)

22.1 Overview

Generators, a new feature of ES6, are functions that can be paused and resumed (think cooperative multitasking or coroutines). That helps with many applications. Two important ones are:

1. Implementing iterables
2. Blocking on asynchronous function calls

The following subsections give brief overviews of these applications.

22.1.1 Implementing iterables via generators

The following function returns an iterable over the properties of an object, one [key, value] pair per property:

```
// The asterisk after `function` means that
// `objectEntries` is a generator
function* objectEntries(obj) {
```



```

const propKeys = Reflect.ownKeys(obj);

for (const propKey of propKeys) {
  // `yield` returns a value and then pauses
  // the generator. Later, execution continues
  // where it was previously paused.
  yield [propKey, obj[propKey]];
}

```

`objectEntries()` is used like this:

```

const jane = { first: 'Jane', last: 'Doe' };
for (const [key, value] of objectEntries(jane)) {
  console.log(`${key}: ${value}`);
}
// Output:
// first: Jane
// last: Doe

```

How exactly `objectEntries()` works is explained in [a dedicated section](#). Implementing the same functionality without generators is much more work.

22.1.2 Blocking on asynchronous function calls

In the following code, I use [the control flow library co](#) to asynchronously retrieve two JSON files. Note how, in line A, execution blocks (waits) via `yield` until the result of `Promise.all()` is ready. That means that the code looks synchronous while performing asynchronous operations.

```

co(function* () {
  try {
    const [croftStr, bondStr] = yield Promise.all([ // (A)
      getFile('http://localhost:8000/croft.json'),
      getFile('http://localhost:8000/bond.json'),
    ]);
    const croftJson = JSON.parse(croftStr);
    const bondJson = JSON.parse(bondStr);

    console.log(croftJson);
    console.log(bondJson);
  } catch (e) {
    console.log('Failure to read: ' + e);
  }
});

```

The Promise-based function `getFile(url)` retrieves the file pointed to by `url`. Its implementation is shown later. I'll also explain how `co` works. For more info on Promises, consult Chap. “[Promises for asynchronous programming](#)”.

22.2 What are generators?

Generators are functions that can be paused and resumed (think cooperative multitasking or coroutines), which enables a variety of applications.

As a first example, consider the following generator function whose name is `genFunc`:

```

function* genFunc() {
  // (A)
  console.log('First');
  yield; // (B)
  console.log('Second'); // (C)
}

```

Two things distinguish `genFunc` from a normal function declaration:

- It starts with the “keyword” `function*`.
- It can pause itself, via `yield` (line B).

Calling `genFunc` does not execute its body. Instead, you get a so-called *generator object*, with which you can control the execution of the body:

```
> const genObj = genFunc();
```

`genFunc()` is initially suspended before the body (line A). The method call `genObj.next()` continues execution until the next `yield`:

```
> genObj.next()
First
{ value: undefined, done: false }
```

As you can see in the last line, `genObj.next()` also returns an object. Let's ignore that for now. It will matter later.

`genFunc` is now paused in line B. If we call `next()` again, execution resumes and line C is executed:

```
> genObj.next()
Second
{ value: undefined, done: true }
```

Afterwards, the function is finished, execution has left the body and further calls of `genObj.next()` have no effect.

22.2.1 Kinds of generators

There are four kinds of generators:

1. Generator function declarations:

```
function* genFunc() { ... }  
const genObj = genFunc();
```

2. Generator function expressions:

```
const genFunc = function* () { ... };  
const genObj = genFunc();
```

3. Generator method definitions in object literals:

```
const obj = {  
  * generatorMethod() {  
    ...  
  }  
};  
const genObj = obj.generatorMethod();
```

4. Generator method definitions in class definitions (class declarations or class expressions):

```
class MyClass {  
  * generatorMethod() {  
    ...  
  }  
}  
const myInst = new MyClass();  
const genObj = myInst.generatorMethod();
```

22.2.2 Roles played by generators

Generators can play three roles:

1. Iterators (data producers): Each `yield` can return a value via `next()`, which means that generators can produce sequences of values via loops and recursion. Due to generator objects implementing the interface `Iterable` (which is explained in [the chapter on iteration](#)), these sequences can be processed by any ECMAScript 6 construct that supports iterables. Two examples are: `for-of` loops and the spread operator `...`.
2. Observers (data consumers): `yield` can also receive a value from `next()` (via a parameter). That means that generators become data consumers that pause until a new value is pushed into them via `next()`.
3. Coroutines (data producers and consumers): Given that generators are pausable and can be both data producers and data consumers, not much work is needed to turn them into coroutines (cooperatively multitasked tasks).

The next sections provide deeper explanations of these roles.

22.3 Generators as iterators (data production)



For this section, you should be familiar with ES6 iteration. [The previous chapter](#) has more information.

As explained before, generator objects can be data producers, data consumers or both. This section looks at them as data producers, where they implement both the interfaces `Iterable` and `Iterator` (shown below). That means that the result of a generator function is both an iterable and an iterator. The full interface of generator objects will be shown later.

```
interface Iterable {  
  [Symbol.iterator]() : Iterator;  
}  
interface Iterator {  
  next() : IteratorResult;  
}  
interface IteratorResult {  
  value : any;  
  done : boolean;  
}
```

I have omitted method `return()` of interface `Iterable`, because it is not relevant in this section.

A generator function produces a sequence of values via `yield`, a data consumer consumes those values via the iterator method `next()`. For example, the following generator function produces the values `'a'` and `'b'`:

```
function* genFunc() {  
  yield 'a';  
  yield 'b';  
}
```

This interaction shows how to retrieve the yielded values via the generator object

genObj:

```
> const genObj = genFunc();
> genObj.next()
{ value: 'a', done: false }
> genObj.next()
{ value: 'b', done: false }
> genObj.next() // done: true => end of sequence
{ value: undefined, done: true }
```

22.3.1 Ways of iterating over a generator

As generator objects are iterable, ES6 language constructs that support iterables can be applied to them. The following three ones are especially important.

First, the `for-of` loop:

```
for (const x of genFunc()) {
  console.log(x);
}
// Output:
// a
// b
```

Second, the spread operator (`...`), which turns iterated sequences into elements of an array (consult [the chapter on parameter handling](#) for more information on this operator):

```
const arr = [...genFunc()]; // ['a', 'b']
```

Third, destructuring:

```
> const [x, y] = genFunc();
> x
'a'
> y
'b'
```

22.3.2 Returning from a generator

The previous generator function did not contain an explicit `return`. An implicit `return` is equivalent to returning `undefined`. Let's examine a generator with an explicit `return`:

```
function* genFuncWithReturn() {
  yield 'a';
  yield 'b';
  return 'result';
}
```

The returned value shows up in the last object returned by `next()`, whose property `done` is `true`:

```
> const genObjWithReturn = genFuncWithReturn();
> genObjWithReturn.next()
{ value: 'a', done: false }
> genObjWithReturn.next()
{ value: 'b', done: false }
> genObjWithReturn.next()
{ value: 'result', done: true }
```

However, most constructs that work with iterables ignore the value inside the `done` object:

```
for (const x of genFuncWithReturn()) {
  console.log(x);
}
// Output:
// a
// b

const arr = [...genFuncWithReturn()]; // ['a', 'b']
```

`yield*`, an operator for making recursive generator calls, does consider values inside `done` objects. It is explained later.

22.3.3 Throwing an exception from a generator

If an exception leaves the body of a generator then `next()` throws it:

```
function* genFunc() {
  throw new Error('Problem!');
}
const genObj = genFunc();
genObj.next(); // Error: Problem!
```

That means that `next()` can produce three different “results”:

- For an item `x` in an iteration sequence, it returns `{ value: x, done: false }`
- For the end of an iteration sequence with a return value `z`, it returns `{ value: z, done: true }`
- For an exception that leaves the generator body, it throws that exception.

22.3.4 Example: iterating over properties

Let's look at an example that demonstrates how convenient generators are for implementing iterables. The following function, `objectEntries()`, returns an iterable over the properties of an object:

```
function* objectEntries(obj) {
  // In ES6, you can use strings or symbols as property keys,
  // Reflect.ownKeys() retrieves both
  const propKeys = Reflect.ownKeys(obj);

  for (const propKey of propKeys) {
    yield [propKey, obj[propKey]];
  }
}
```

This function enables you to iterate over the properties of an object `jane` via the `for-of` loop:

```
const jane = { first: 'Jane', last: 'Doe' };
for (const [key,value] of objectEntries(jane)) {
  console.log(`${key}: ${value}`);
}
// Output:
// first: Jane
// last: Doe
```

For comparison – an implementation of `objectEntries()` that doesn't use generators is much more complicated:

```
function objectEntries(obj) {
  let index = 0;
  let propKeys = Reflect.ownKeys(obj);

  return {
    [Symbol.iterator]() {
      return this;
    },
    next() {
      if (index < propKeys.length) {
        let key = propKeys[index];
        index++;
        return { value: [key, obj[key]] };
      } else {
        return { done: true };
      }
    }
  };
}
```

22.3.5 You can only `yield` in generators

A significant limitation of generators is that you can only `yield` while you are (statically) inside a generator function. That is, yielding in callbacks doesn't work:

```
function* genFunc() {
  ['a', 'b'].forEach(x => yield x); // SyntaxError
}
```

`yield` is not allowed inside non-generator functions, which is why the previous code causes a syntax error. In this case, it is easy to rewrite the code so that it doesn't use callbacks (as shown below). But unfortunately that isn't always possible.

```
function* genFunc() {
  for (const x of ['a', 'b']) {
    yield x; // OK
  }
}
```

The upside of this limitation is [explained later](#): it makes generators easier to implement and compatible with event loops.

22.3.6 Recursion via `yield*`

You can only use `yield` within a generator function. Therefore, if you want to implement a recursive algorithm with generator, you need a way to call one generator from another one. This section shows that that is more complicated than it sounds, which is why ES6 has a special operator, `yield*`, for this. For now, I only explain how `yield*` works if both generators produce output, I'll later explain how things work if input is involved.

How can one generator recursively call another generator? Let's assume you have written a generator function `foo`:

```
function* foo() {
  yield 'a';
  yield 'b';
}
```

How would you call `foo` from another generator function `bar`? The following approach does not work!

```
function* bar() {
  yield 'x';
  foo(); // does nothing!
  yield 'y';
}
```

Calling `foo()` returns an object, but does not actually execute `foo()`. That's why ECMAScript 6 has the operator `yield*` for making recursive generator calls:

```
function* bar() {
  yield 'x';
  yield* foo();
  yield 'y';
}

// Collect all values yielded by bar() in an array
const arr = [...bar()];
// ['x', 'a', 'b', 'y']
```

Internally, `yield*` works roughly as follows:

```
function* bar() {
  yield 'x';
  for (const value of foo()) {
    yield value;
  }
  yield 'y';
}
```

The operand of `yield*` does not have to be a generator object, it can be any iterable:

```
function* bla() {
  yield 'sequence';
  yield* ['of', 'yielded'];
  yield 'values';
}

const arr = [...bla()];
// ['sequence', 'of', 'yielded', 'values']
```

22.3.6.1 `yield*` considers end-of-iteration values

Most constructs that support iterables ignore the value included in the end-of-iteration object (whose property `done` is `true`). Generators provide that value via `return`. The result of `yield*` is the end-of-iteration value:

```
function* genFuncWithReturn() {
  yield 'a';
  yield 'b';
  return 'The result';
}

function* logReturned(genObj) {
  const result = yield* genObj;
  console.log(result); // (A)
}
```

If we want to get to line A, we first must iterate over all values yielded by `logReturned()`:

```
> [...logReturned(genFuncWithReturn())]
The result
[ 'a', 'b' ]
```

22.3.6.2 Iterating over trees

Iterating over a tree with recursion is simple, writing an iterator for a tree with traditional means is complicated. That's why generators shine here: they let you implement an iterator via recursion. As an example, consider the following data structure for binary trees. It is iterable, because it has a method whose key is `Symbol.iterator`. That method is a generator method and returns an iterator when called.

```
class BinaryTree {
  constructor(value, left=null, right=null) {
    this.value = value;
    this.left = left;
    this.right = right;
  }

  /** Prefix iteration */
  * [Symbol.iterator]() {
    yield this.value;
    if (this.left) {
      yield* this.left;
      // Short for: yield* this.left[Symbol.iterator]()
    }
    if (this.right) {
      yield* this.right;
    }
  }
}
```

The following code creates a binary tree and iterates over it via `for-of`:

```
const tree = new BinaryTree('a',
  new BinaryTree('b',
    new BinaryTree('c'),
    new BinaryTree('d')),
  new BinaryTree('e'));

for (const x of tree) {
  console.log(x);
}

// Output:
// a
// b
// c
```

```
// d
// e
```

22.4 Generators as observers (data consumption)

As consumers of data, generator objects conform to the second half of the generator interface, `Observer`:

```
interface Observer {
  next(value? : any) : void;
  return(value? : any) : void;
  throw(error) : void;
}
```

As an observer, a generator pauses until it receives input. There are three kinds of input, transmitted via the methods specified by the interface:

- `next()` sends normal input.
- `return()` terminates the generator.
- `throw()` signals an error.

22.4.1 Sending values via `next()`

If you use a generator as an observer, you send values to it via `next()` and it receives those values via `yield`:

```
function* dataConsumer() {
  console.log('Started');
  console.log(`1. ${yield}`); // (A)
  console.log(`2. ${yield}`);
  return 'result';
}
```

Let's use this generator interactively. First, we create a generator object:

```
> const genObj = dataConsumer();
```

We now call `genObj.next()`, which starts the generator. Execution continues until the first `yield`, which is where the generator pauses. The result of `next()` is the value yielded in line A (`undefined`, because `yield` doesn't have an operand). In this section, we are not interested in what `next()` returns, because we only use it to send values, not to retrieve values.

```
> genObj.next()
Started
{ value: undefined, done: false }
```

We call `next()` two more times, in order to send the value `'a'` to the first `yield` and the value `'b'` to the second `yield`:

```
> genObj.next('a')
1. a
{ value: undefined, done: false }

> genObj.next('b')
2. b
{ value: 'result', done: true }
```

The result of the last `next()` is the value returned from `dataConsumer()`. `done` being `true` indicates that the generator is finished.

Unfortunately, `next()` is asymmetric, but that can't be helped: It always sends a value to the currently suspended `yield`, but returns the operand of the following `yield`.

22.4.1.1 The first `next()`

When using a generator as an observer, it is important to note that the only purpose of the first invocation of `next()` is to start the observer. It is only ready for input afterwards, because this first invocation advances execution to the first `yield`. Therefore, any input you send via the first `next()` is ignored:

```
function* gen() {
  // (A)
  while (true) {
    const input = yield; // (B)
    console.log(input);
  }
}
const obj = gen();
obj.next('a');
obj.next('b');

// Output:
// b
```

Initially, execution is paused in line A. The first invocation of `next()`:

- Feeds the argument `'a'` of `next()` to the generator, which has no way to receive it (as there is no `yield`). That's why it is ignored.
- Advances to the `yield` in line B and pauses execution.
- Returns `yield`'s operand (`undefined`, because it doesn't have an operand).

The second invocation of `next()`:

- Feeds the argument `'b'` of `next()` to the generator, which receives it via the `yield` in line B and assigns it to the variable `input`.
- Then execution continues until the next loop iteration, where it is paused again, in line B.
- Then `next()` returns with that `yield`'s operand (undefined).

The following utility function fixes this issue:

```
/**
 * Returns a function that, when called,
 * returns a generator object that is immediately
 * ready for input via `next()`
 */
function coroutine(generatorFunction) {
  return function (...args) {
    const generatorObject = generatorFunction(...args);
    generatorObject.next();
    return generatorObject;
  };
}
```

To see how `coroutine()` works, let's compare a wrapped generator with a normal one:

```
const wrapped = coroutine(function* () {
  console.log(`First input: ${yield}`);
  return 'DONE';
});
const normal = function* () {
  console.log(`First input: ${yield}`);
  return 'DONE';
};
```

The wrapped generator is immediately ready for input:

```
> wrapped().next('hello!')
First input: hello!
```

The normal generator needs an extra `next()` until it is ready for input:

```
> const genObj = normal();
> genObj.next()
{ value: undefined, done: false }
> genObj.next('hello!')
First input: hello!
{ value: 'DONE', done: true }
```

22.4.2 `yield` binds loosely

`yield` binds very loosely, so that we don't have to put its operand in parentheses:

```
yield a + b + c;
```

This is treated as:

```
yield (a + b + c);
```

Not as:

```
(yield a) + b + c;
```

As a consequence, many operators bind more tightly than `yield` and you have to put `yield` in parentheses if you want to use it as an operand. For example, you get a `SyntaxError` if you make an unparenthesized `yield` an operand of plus:

```
console.log('Hello' + yield); // SyntaxError
console.log('Hello' + yield 123); // SyntaxError

console.log('Hello' + (yield)); // OK
console.log('Hello' + (yield 123)); // OK
```

You do not need parens if `yield` is a direct argument in a function or method call:

```
foo(yield 'a', yield 'b');
```

You also don't need parens if you use `yield` on the right-hand side of an assignment:

```
const input = yield;
```

22.4.2.1 `yield` in the ES6 grammar

The need for parens around `yield` can be seen in the following grammar rules in the [ECMAScript 6 specification](#). These rules describe how expressions are parsed. I list them here from general (loose binding, lower precedence) to specific (tight binding, higher precedence). Wherever a certain kind of expression is demanded, you can also use more specific ones. The opposite is not true. The hierarchy ends with `ParenthesizedExpression`, which means that you can mention any expression anywhere, if you put it in parentheses.

```
Expression :
  AssignmentExpression
  Expression , AssignmentExpression
AssignmentExpression :
```

```

ConditionalExpression
YieldExpression
ArrowFunction
LeftHandSideExpression = AssignmentExpression
LeftHandSideExpression AssignmentOperator AssignmentExpression
...

AdditiveExpression :
    MultiplicativeExpression
    AdditiveExpression + MultiplicativeExpression
    AdditiveExpression - MultiplicativeExpression
MultiplicativeExpression :
    UnaryExpression
    MultiplicativeExpression MultiplicativeOperator UnaryExpression
...

PrimaryExpression :
    this
    IdentifierReference
    Literal
    ArrayLiteral
    ObjectLiteral
    FunctionExpression
    ClassExpression
    GeneratorExpression
    RegularExpressionLiteral
    TemplateLiteral
    ParenthesizedExpression
ParenthesizedExpression :
    ( Expression )

```

The operands of an `AdditiveExpression` are an `AdditiveExpression` and a `MultiplicativeExpression`. Therefore, using a (more specific) `ParenthesizedExpression` as an operand is OK, but using a (more general) `YieldExpression` isn't.

22.4.3 `return()` and `throw()`

Generator objects have two additional methods, `return()` and `throw()`, that are similar to `next()`.

Let's recap how `next(x)` works (after the first invocation):

1. The generator is currently suspended at a `yield` operator.
2. Send the value `x` to that `yield`, which means that it evaluates to `x`.
3. Proceed to the next `yield`, `return` or `throw`:
 - `yield x` leads to `next()` returning with `{ value: x, done: false }`
 - `return x` leads to `next()` returning with `{ value: x, done: true }`
 - `throw err` (not caught inside the generator) leads to `next()` throwing `err`.

`return()` and `throw()` work similarly to `next()`, but they do something different in step 2:

- `return(x)` executes `return x` at the location of `yield`.
- `throw(x)` executes `throw x` at the location of `yield`.

22.4.4 `return()` terminates the generator

`return()` performs a `return` at the location of the `yield` that led to the last suspension of the generator. Let's use the following generator function to see how that works.

```

function* genFunc1() {
  try {
    console.log('Started');
    yield; // (A)
  } finally {
    console.log('Exiting');
  }
}

```

In the following interaction, we first use `next()` to start the generator and to proceed until the `yield` in line A. Then we return from that location via `return()`.

```

> const genObj1 = genFunc1();
> genObj1.next()
Started
{ value: undefined, done: false }
> genObj1.return('Result')
Exiting
{ value: 'Result', done: true }

```

22.4.4.1 Preventing termination

You can prevent `return()` from terminating the generator if you `yield` inside the `finally` clause (using a `return` statement in that clause is also possible):

```

function* genFunc2() {
  try {
    console.log('Started');
    yield;
  } finally {
    yield 'Not done, yet!';
  }
}

```


This time, `return()` does not exit the generator function. Accordingly, the property `done` of the object it returns is `false`.

```
> const genObj2 = genFunc2();
> genObj2.next()
Started
{ value: undefined, done: false }
> genObj2.return('Result')
{ value: 'Not done, yet!', done: false }
```

You can invoke `next()` one more time. Similarly to non-generator functions, the return value of the generator function is the value that was queued prior to entering the `finally` clause.

```
> genObj2.next()
{ value: 'Result', done: true }
```

22.4.4.2 Returning from a newborn generator

Returning a value from a *newborn* generator (that hasn't started yet) is allowed:

```
> function* genFunc() {}
> genFunc().return('yes')
{ value: 'yes', done: true }
```



Further reading

`return()` is also used to close iterators. The chapter on iteration has [a detailed section on that](#).

22.4.5 `throw()` signals an error

`throw()` throws an exception at the location of the `yield` that led to the last suspension of the generator. Let's examine how that works via the following generator function.

```
function* genFunc1() {
  try {
    console.log('Started');
    yield; // (A)
  } catch (error) {
    console.log('Caught: ' + error);
  }
}
```

In the following interaction, we first use `next()` to start the generator and proceed until the `yield` in line A. Then we throw an exception from that location.

```
> const genObj1 = genFunc1();
> genObj1.next()
Started
{ value: undefined, done: false }
> genObj1.throw(new Error('Problem!'))
Caught: Error: Problem!
{ value: undefined, done: true }
```

The result of `throw()` (shown in the last line) stems from us leaving the function with an implicit `return`.

22.4.5.1 Throwing from a newborn generator

Throwing an exception in a *newborn* generator (that hasn't started yet) is allowed:

```
> function* genFunc() {}
> genFunc().throw(new Error('Problem!'))
Error: Problem!
```

22.4.6 Example: processing asynchronously pushed data

The fact that generators-as-observers pause while they wait for input makes them perfect for on-demand processing of data that is received asynchronously. The pattern for setting up a chain of generators for processing is as follows:

- Each member of the chain of generators (except the last one) has a parameter `target`. It receives data via `yield` and sends data via `target.next()`.
- The last member of the chain of generators has no parameter `target` and only receives data.

The whole chain is prefixed by a non-generator function that makes an asynchronous request and pushes the results into the chain of generators via `next()`.

As an example, let's chain generators to process a file that is read asynchronously.



The code of this example is in the file [generator-examples/node/readlines.js](#). It must be executed via `babel-node`.

The following code sets up the chain: it contains the generators `splitLines`, `numberLines` and `printLines`. Data is pushed into the chain via the non-generator function `readFile`.

```
readFile(fileName, splitLines(numberLines(printLines())));
```

I'll explain what these functions do when I show their code.

As previously explained, if generators receive input via `yield`, the first invocation of `next()` on the generator object doesn't do anything. That's why I use [the previously shown helper function `coroutine\(\)`](#) to create coroutines here. It executes the first `next()` for us.

`readFile()` is the non-generator function that starts everything:

```
import {createReadStream} from 'fs';

/**
 * Creates an asynchronous ReadStream for the file whose name
 * is 'fileName' and feed it to the generator object 'target'.
 *
 * @see ReadStream https://nodejs.org/api/fs.html#fs_class_fs_readstream
 */
function readFile(fileName, target) {
  const readStream = createReadStream(fileName,
    { encoding: 'utf8', bufferSize: 1024 });
  readStream.on('data', buffer => {
    const str = buffer.toString('utf8');
    target.next(str);
  });
  readStream.on('end', () => {
    // Signal end of output sequence
    target.return();
  });
}
```

The chain of generators starts with `splitLines`:

```
/**
 * Turns a sequence of text chunks into a sequence of lines
 * (where lines are separated by newlines)
 */
const splitLines = coroutine(function* (target) {
  let previous = '';
  try {
    while (true) {
      previous += yield;
      let eolIndex;
      while ((eolIndex = previous.indexOf('\n')) >= 0) {
        const line = previous.slice(0, eolIndex);
        target.next(line);
        previous = previous.slice(eolIndex+1);
      }
    }
  } finally {
    // Handle the end of the input sequence
    // (signaled via `return()`)
    if (previous.length > 0) {
      target.next(previous);
    }
    // Signal end of output sequence
    target.return();
  }
});
```

Note an important pattern:

- `readFile` uses the generator object method `return()` to signal the end of the sequence of chunks that it sends.
- `readFile` sends that signal while `splitLines` is waiting for input via `yield`, inside an infinite loop. `return()` breaks from that loop.
- `splitLines` uses a `finally` clause to handle the end-of-sequence.

The next generator is `numberLines`:

```
/**
 * Prefixes numbers to a sequence of lines
 */
const numberLines = coroutine(function* (target) {
  try {
    for (const lineNo = 0; ; lineNo++) {
      const line = yield;
      target.next(`${lineNo}: ${line}`);
    }
  } finally {
    // Signal end of output sequence
    target.return();
  }
});
```

The last generator is `printLines`:

```

/**
 * Receives a sequence of lines (without newlines)
 * and logs them (adding newlines).
 */
const printLines = coroutine(function* () {
  while (true) {
    const line = yield;
    console.log(line);
  }
});

```

The neat thing about this code is that everything happens lazily (on demand): lines are split, numbered and printed as they arrive; we don't have to wait for all of the text before we can start printing.

22.4.7 `yield*`: the full story

As a rough rule of thumb, `yield*` performs (the equivalent of) a function call from one generator (the *caller*) to another generator (the *callee*).

So far, we have only seen one aspect of `yield`: it propagates yielded values from the callee to the caller. Now that we are interested in generators receiving input, another aspect becomes relevant: `yield*` also forwards input received by the caller to the callee. In a way, the callee becomes the active generator and can be controlled via the caller's generator object.

22.4.7.1 Example: `yield*` forwards `next()`

The following generator function `caller()` invokes the generator function `callee()` via `yield*`.

```

function* callee() {
  console.log('callee: ' + (yield));
}
function* caller() {
  while (true) {
    yield* callee();
  }
}

```

`callee` logs values received via `next()`, which allows us to check whether it receives the value 'a' and 'b' that we send to `caller`.

```

> const callerObj = caller();

> callerObj.next() // start
{ value: undefined, done: false }

> callerObj.next('a')
callee: a
{ value: undefined, done: false }

> callerObj.next('b')
callee: b
{ value: undefined, done: false }

```

`throw()` and `return()` are forwarded in a similar manner.

22.4.7.2 The semantics of `yield*` expressed in JavaScript

I'll explain the complete semantics of `yield*` by showing how you'd implemented it in JavaScript.

The following statement:

```
let yieldStarResult = yield* calleeFunc();
```

is roughly equivalent to:

```

let yieldStarResult;

const calleeObj = calleeFunc();
let prevReceived = undefined;
while (true) {
  try {
    // Forward input previously received
    const {value,done} = calleeObj.next(prevReceived);
    if (done) {
      yieldStarResult = value;
      break;
    }
    prevReceived = yield value;
  } catch (e) {
    // Pretend `return` can be caught like an exception
    if (e instanceof Return) {
      // Forward input received via return()
      calleeObj.return(e.returnedValue);
      return e.returnedValue; // "re-throw"
    } else {
      // Forward input received via throw()
      calleeObj.throw(e); // may throw
    }
  }
}

```

To keep things simple, several things are missing in this code:

- The operand of `yield*` can be any iterable value.
- `return()` and `throw()` are optional iterator methods. We should only call them if they exist.
- If an exception is received and `throw()` does not exist, but `return()` does then `return()` is called (before throwing an exception) to give `calleeObject` the opportunity to clean up.
- `calleeObj` can refuse to close, by returning an object whose property `done` is `false`. Then the caller also has to refuse to close and `yield*` must continue to iterate.

22.5 Generators as coroutines (cooperative multitasking)

We have seen generators being used as either sources or sinks of data. For many applications, it's good practice to strictly separate these two roles, because it keeps things simpler. This section describes the full generator interface (which combines both roles) and one use case where both roles are needed: cooperative multitasking, where tasks must be able to both send and receive information.

22.5.1 The full generator interface

The full interface of generator objects, `Generator`, handles both output and input:

```
interface Generator {
  next(value? : any) : IteratorResult;
  throw(value? : any) : IteratorResult;
  return(value? : any) : IteratorResult;
}
interface IteratorResult {
  value : any;
  done : boolean;
}
```



This interface is described in the spec in the section "[Properties of Generator Prototype](#)".

The interface `Generator` combines two interfaces that we have seen previously: `Iterator` for output and `Observer` for input.

```
interface Iterator { // data producer
  next() : IteratorResult;
  return?(value? : any) : IteratorResult;
}
interface Observer { // data consumer
  next(value? : any) : void;
  return(value? : any) : void;
  throw(error) : void;
}
```

22.5.2 Cooperative multitasking

Cooperative multitasking is an application of generators where we need them to handle both output and input. Before we get into how that works, let's first review the current state of parallelism in JavaScript.

JavaScript runs in a single process. There are two ways in which this limitation is being abolished:

- **Multiprocessing:** *Web Workers* let you run JavaScript in multiple processes. Shared access to data is one of the biggest pitfalls of multiprocessing. Web Workers avoid it by not sharing any data. That is, if you want a Web Worker to have a piece of data, you must send it a copy or transfer your data to it (after which you can't access it anymore).
- **Cooperative multitasking:** There are various patterns and libraries that experiment with cooperative multitasking. Multiple tasks are run, but only one at a time. Each task must explicitly suspend itself, giving it full control over when a task switch happens. In these experiments, data is often shared between tasks. But due to explicit suspension, there are few risks.

Two use cases benefit from cooperative multitasking, because they involve control flows that are mostly sequential, anyway, with occasional pauses:

- **Streams:** A task sequentially processes a stream of data and pauses if there is no data available.
 - For binary streams, WHATWG is currently working on a [standard proposal](#) that is based on callbacks and Promises.
 - For streams of data, Communicating Sequential Processes (CSP) are an interesting solution. A generator-based CSP library is covered [later in this chapter](#).
- **Asynchronous computations:** A task blocks (pauses) until it receives the result of a long- running computation.
 - In JavaScript, [Promises](#) have become a popular way of handling asynchronous computations. Support for them is included in ES6. The next section explains how generators can make using Promises simpler.

22.5.2.1 Simplifying asynchronous computations via generators

Several Promise-based libraries simplify asynchronous code via generators. Generators are ideal as clients of Promises, because they can be suspended until a result arrives.

The following example demonstrates what that looks like if one uses [the library co](#) by T.J. Holowaychuk. We need two libraries (if we run Node.js code via `babel-node`):

```
import fetch from 'isomorphic-fetch';
const co = require('co');
```

`co` is the actual library for cooperative multitasking, `isomorphic-fetch` is a polyfill for the new Promise-based `fetch` API (a replacement of `XMLHttpRequest`; read [“That’s so fetch!”](#) by Jake Archibald for more information). `fetch` makes it easy to write a function `getFile` that returns the text of a file at a `url` via a Promise:

```
function getFile(url) {
  return fetch(url)
    .then(request => request.text());
}
```

We now have all the ingredients to use `co`. The following task reads the texts of two files, parses the JSON inside them and logs the result.

```
co(function* () {
  try {
    const [croftStr, bondStr] = yield Promise.all([ // (A)
      getFile('http://localhost:8000/croft.json'),
      getFile('http://localhost:8000/bond.json'),
    ]);
    const croftJson = JSON.parse(croftStr);
    const bondJson = JSON.parse(bondStr);

    console.log(croftJson);
    console.log(bondJson);
  } catch (e) {
    console.log('Failure to read: ' + e);
  }
});
```

Note how nicely synchronous this code looks, even though it makes an asynchronous call in line A. A generator-as-task makes an async call by yielding a Promise to the scheduler function `co`. The yielding pauses the generator. Once the Promise returns a result, the scheduler resumes the generator by passing it the result via `next()`. A simple version of `co` looks as follows.

```
function co(genFunc) {
  const genObj = genFunc();
  step(genObj.next());

  function step({value, done}) {
    if (!done) {
      // A Promise was yielded
      value
        .then(result => {
          step(genObj.next(result)); // (A)
        })
        .catch(error => {
          step(genObj.throw(error)); // (B)
        });
    }
  }
}
```

I have ignored that `next()` (line A) and `throw()` (line B) may throw exceptions (whenever an exception escapes the body of the generator function).

22.5.3 The limitations of cooperative multitasking via generators

Coroutines are cooperatively multitasked tasks that have no limitations: Inside a coroutine, any function can suspend the whole coroutine (the function activation itself, the activation of the function’s caller, the caller’s caller, etc.).

In contrast, you can only suspend a generator from directly within a generator and only the current function activation is suspended. Due to these limitations, generators are occasionally called [shallowcoroutines](#) [3].

22.5.3.1 The benefits of the limitations of generators

The limitations of generators have two main benefits:

- Generators are compatible with *event loops*, which provide simple cooperative multitasking in browsers. I’ll explain the details momentarily.
- Generators are relatively easy to implement, because only a single function activation needs to be suspended and because browsers can continue to use event loops.

JavaScript already has a very simple style of cooperative multitasking: the event loop, which schedules the execution of tasks in a queue. Each task is started by calling a function and finished once that function is finished. Events, `setTimeout()` and other mechanisms add tasks to the queue.



This explanation of the event loop is a simplification that is good enough for now. If you are interested in details, consult [the chapter on asynchronous programming](#).

This style of multitasking makes one important guarantee: *run to completion*; every function can rely on not being interrupted by another task until it is finished. Functions become transactions and can perform complete algorithms without anyone seeing the data they operate on in an intermediate state. Concurrent access to shared data makes multitasking complicated and is not allowed by JavaScript's concurrency model. That's why run to completion is a good thing.

Alas, coroutines prevent run to completion, because any function could suspend its caller. For example, the following algorithm consists of multiple steps:

```
step1(sharedData);
step2(sharedData);
lastStep(sharedData);
```

If `step2` was to suspend the algorithm, other tasks could run before the last step of the algorithm is performed. Those tasks could contain other parts of the application which would see `sharedData` in an unfinished state. Generators preserve run to completion, they only suspend themselves and return to their caller.

`co` and similar libraries give you most of the power of coroutines, without their disadvantages:

- They provide schedulers for tasks defined via generators.
- Tasks “are” generators and can thus be fully suspended.
- A recursive (generator) function call is only suspendable if it is done via `yield*`. That gives callers control over suspension.

22.6 Examples of generators

This section gives several examples of what generators can be used for.



The following GitHub repository contains the example code: [generator-examples](#)

22.6.1 Implementing iterables via generators

In [the chapter on iteration](#), I implemented several iterables “by hand”. In this section, I use generators, instead.

22.6.1.1 The iterable combinator `take()`

`take()` converts a (potentially infinite) sequence of iterated values into a sequence of length `n`:

```
function* take(n, iterable) {
  for (const x of iterable) {
    if (n <= 0) return;
    n--;
    yield x;
  }
}
```

The following is an example of using it:

```
const arr = ['a', 'b', 'c', 'd'];
for (const x of take(2, arr)) {
  console.log(x);
}
// Output:
// a
// b
```

An implementation of `take()` without generators is more complicated:

```
function take(n, iterable) {
  const iter = iterable[Symbol.iterator]();
  return {
    [Symbol.iterator]() {
      return this;
    },
    next() {
      if (n > 0) {
        n--;
        return iter.next();
      } else {
        maybeCloseIterator(iter);
        return { done: true };
      }
    },
  };
}
return {
  n: 0;
}
```

```

        maybeCloseIterator(iter);
    }
};
}
function maybeCloseIterator(iterator) {
    if (typeof iterator.return === 'function') {
        iterator.return();
    }
}
}

```

Note that the iterable combinator `zip()` does not profit much from being implemented via a generator, because multiple iterables are involved and `for-of` can't be used.

22.6.1.2 Infinite iterables

`naturalNumbers()` returns an iterable over all natural numbers:

```

function* naturalNumbers() {
    for (const n=0;; n++) {
        yield n;
    }
}

```

This function is often used in conjunction with a combinator:

```

for (const x of take(3, naturalNumbers())) {
    console.log(x);
}
// Output
// 0
// 1
// 2

```

Here is the non-generator implementation, so you can compare:

```

function naturalNumbers() {
    let n = 0;
    return {
        [Symbol.iterator]() {
            return this;
        },
        next() {
            return { value: n++ };
        }
    }
}

```

22.6.1.3 Array-inspired iterable combinators: `map`, `filter`

Arrays can be transformed via the methods `map` and `filter`. Those methods can be generalized to have iterables as input and iterables as output.

22.6.1.3.1 A generalized `map()`

This is the generalized version of `map`:

```

function* map(iterable, mapFunc) {
    for (const x of iterable) {
        yield mapFunc(x);
    }
}

```

`map()` works with infinite iterables:

```

> [...take(4, map(naturalNumbers(), x => x * x))]
[ 0, 1, 4, 9 ]

```

22.6.1.3.2 A generalized `filter()`

This is the generalized version of `filter`:

```

function* filter(iterable, filterFunc) {
    for (const x of iterable) {
        if (filterFunc(x)) {
            yield x;
        }
    }
}

```

`filter()` works with infinite iterables:

```

> [...take(4, filter(naturalNumbers(), x => (x % 2) === 0))]
[ 0, 2, 4, 6 ]

```

22.6.2 Generators for lazy evaluation

The next two examples show how generators can be used to process a stream of characters.

- The input is a stream of characters.
- Step 1 – tokenizing (characters → words): The characters are grouped into *words*, strings that match the regular expression `/^[A-Za-z0-9]+$/` . Non-word characters are ignored, but they separate words. The input of this step is a stream of characters, the output a stream of words.

- Step 2 – extracting numbers (words → numbers): only keep words that match the regular expression `/^[0-9]+$` and convert them to numbers.
- Step 3 – adding numbers (numbers → numbers): for every number received, return the total received so far.

The neat thing is that everything is computed *lazily* (incrementally and on demand): computation starts as soon as the first character arrives. For example, we don't have to wait until we have all characters to get the first word.

22.6.2.1 Lazy pull (generators as iterators)

Lazy pull with generators works as follows. The three generators implementing steps 1–3 are chained as follows:

```
addNumbers(extractNumbers(tokenize(CHARS)))
```

Each of the chain members pulls data from a source and yields a sequence of items. Processing starts with `tokenize` whose source is the string `CHARS`.

22.6.2.1.1 Step 1 – tokenizing

The following trick makes the code a bit simpler: the end-of-sequence iterator result (whose property `done` is `false`) is converted into the sentinel value `END_OF_SEQUENCE`.

```
/**
 * Returns an iterable that transforms the input sequence
 * of characters into an output sequence of words.
 */
function* tokenize(chars) {
  const iterator = chars[Symbol.iterator]();
  let ch;
  do {
    ch = getNextItem(iterator); // (A)
    if (isWordChar(ch)) {
      let word = '';
      do {
        word += ch;
        ch = getNextItem(iterator); // (B)
      } while (isWordChar(ch));
      yield word; // (C)
    }
    // Ignore all other characters
  } while (ch !== END_OF_SEQUENCE);
}
const END_OF_SEQUENCE = Symbol();
function getNextItem(iterator) {
  const {value, done} = iterator.next();
  return done ? END_OF_SEQUENCE : value;
}
function isWordChar(ch) {
  return typeof ch === 'string' && /^[A-Za-z0-9]+$/.test(ch);
}
```

How is this generator lazy? When you ask it for a token via `next()`, it pulls its `iterator` (lines A and B) as often as needed to produce a token and then yields that token (line C). Then it pauses until it is again asked for a token. That means that tokenization starts as soon as the first characters are available, which is convenient for streams.

Let's try out tokenization. Note that the spaces and the dot are non-words. They are ignored, but they separate words. We use the fact that strings are iterables over characters (Unicode code points). The result of `tokenize()` is an iterable over words, which we turn into an array via the spread operator (`...`).

```
> [...tokenize('2 apples and 5 oranges.')]
[ '2', 'apples', 'and', '5', 'oranges' ]
```

22.6.2.1.2 Step 2 – extracting numbers

This step is relatively simple, we only `yield` words that contain nothing but digits, after converting them to numbers via `Number()`.

```
/**
 * Returns an iterable that filters the input sequence
 * of words and only yields those that are numbers.
 */
function* extractNumbers(words) {
  for (const word of words) {
    if (/^[0-9]+$/.test(word)) {
      yield Number(word);
    }
  }
}
```

You can again see the laziness: If you ask for a number via `next()`, you get one (via `yield`) as soon as one is encountered in `words`.

Let's extract the numbers from an Array of words:

```
> [...extractNumbers(['hello', '123', 'world', '45'])]
[ 123, 45 ]
```

Note that strings are converted to numbers.

22.6.2.1.3 Step 3 – adding numbers


```

/**
 * Returns an iterable that contains, for each number in
 * `numbers`, the total sum of numbers encountered so far.
 * For example: 7, 4, -1 --> 7, 11, 10
 */
function* addNumbers(numbers) {
  let result = 0;
  for (const n of numbers) {
    result += n;
    yield result;
  }
}

```

Let's try a simple example:

```

> [...addNumbers([5, -2, 12])]
[ 5, 3, 15 ]

```

22.6.2.1.4 Pulling the output

On its own, the chain of generator doesn't produce output. We need to actively pull the output via the spread operator:

```

const CHARS = '2 apples and 5 oranges.';
const CHAIN = addNumbers(extractNumbers(tokenize(CHARS)));
console.log(...CHAIN);
// [ 2, 7 ]

```

The helper function `logAndYield` allows us to examine whether things are indeed computed lazily:

```

function* logAndYield(iterable, prefix='') {
  for (const item of iterable) {
    console.log(prefix + item);
    yield item;
  }
}

const CHAIN2 = logAndYield(addNumbers(extractNumbers(tokenize(logAndYield(CHARS))))), '-> ');
[...CHAIN2];

// Output:
// 2
//
// -> 2
// a
// p
// p
// l
// e
// s
//
// a
// n
// d
//
// 5
//
// -> 7
// o
// r
// a
// n
// g
// e
// s
// .

```

The output shows that `addNumbers` produces a result as soon as the characters '2' and ' ' are received.

22.6.2.2 Lazy push (generators as observables)

Not much work is needed to convert the previous pull-based algorithm into a push-based one. The steps are the same. But instead of finishing via pulling, we start via pushing.

As previously explained, if generators receive input via `yield`, the first invocation of `next()` on the generator object doesn't do anything. That's why I use [the previously shown helper function `coroutine\(\)`](#) to create coroutines here. It executes the first `next()` for us.

The following function `send()` does the pushing.

```

/**
 * Pushes the items of `iterable` into `sink`, a generator.
 * It uses the generator method `next()` to do so.
 */
function send(iterable, sink) {
  for (const x of iterable) {
    sink.next(x);
  }
  sink.return(); // signal end of stream
}

```

When a generator processes a stream, it needs to be aware of the end of the stream, so that it can clean up properly. For pull, we did this via a special end-of-stream

sentinel. For push, the end-of-stream is signaled via `return()`.

Let's test `send()` via a generator that simply outputs everything it receives:

```
/**
 * This generator logs everything that it receives via `next()`.
 */
const logItems = coroutine(function* () {
  try {
    while (true) {
      const item = yield; // receive item via `next()`
      console.log(item);
    }
  } finally {
    console.log('DONE');
  }
});
```

Let's send `logItems()` three characters via a string (which is an iterable over Unicode code points).

```
> send('abc', logItems());
a
b
c
DONE
```

22.6.2.2.1 Step 1 – tokenize

Note how this generator reacts to the end of the stream (as signaled via `return()`) in two `finally` clauses. We depend on `return()` being sent to either one of the two `yields`. Otherwise, the generator would never terminate, because the infinite loop starting in line A would never terminate.

```
/**
 * Receives a sequence of characters (via the generator object
 * method `next()`), groups them into words and pushes them
 * into the generator `sink`.
 */
const tokenize = coroutine(function* (sink) {
  try {
    while (true) { // (A)
      let ch = yield; // (B)
      if (isWordChar(ch)) {
        // A word has started
        let word = '';
        try {
          do {
            word += ch;
            ch = yield; // (C)
          } while (isWordChar(ch));
        } finally {
          // The word is finished.
          // We get here if
          // - the loop terminates normally
          // - the loop is terminated via `return()` in line C
          sink.next(word); // (D)
        }
      }
      // Ignore all other characters
    }
  } finally {
    // We only get here if the infinite loop is terminated
    // via `return()` (in line B or C).
    // Forward `return()` to `sink` so that it is also
    // aware of the end of stream.
    sink.return();
  }
});

function isWordChar(ch) {
  return /^[A-Za-z0-9]$/.test(ch);
}
```

This time, the laziness is driven by push: as soon as the generator has received enough characters for a word (in line C), it pushes the word into `sink` (line D). That is, the generator does not wait until it has received all characters.

`tokenize()` demonstrates that generators work well as implementations of linear state machines. In this case, the machine has two states: “inside a word” and “not inside a word”.

Let's tokenize a string:

```
> send('2 apples and 5 oranges.', tokenize(logItems()));
2
apples
and
5
oranges
```

22.6.2.2.2 Step 2 – extract numbers

This step is straightforward.

```
/**
 * Receives a sequence of strings (via the generator object
 * method `next()`) and pushes only those strings to the generator
 * `sink` that are “numbers” (consist only of decimal digits).
 */
```

```

const extractNumbers = coroutine(function* (sink) {
  try {
    while (true) {
      const word = yield;
      if (/^[0-9]+$/.test(word)) {
        sink.next(Number(word));
      }
    }
  } finally {
    // Only reached via `return()`, forward.
    sink.return();
  }
});

```

Things are again lazy: as soon as a number is encountered, it is pushed to `sink`.

Let's extract the numbers from an Array of words:

```

> send(['hello', '123', 'world', '45'], extractNumbers(logItems()));
123
45
DONE

```

Note that the input is a sequence of strings, while the output is a sequence of numbers.

22.6.2.2.3 Step 3 – add numbers

This time, we react to the end of the stream by pushing a single value and then closing the sink.

```

/**
 * Receives a sequence of numbers (via the generator object
 * method `next()`). For each number, it pushes the total sum
 * so far to the generator `sink`.
 */
const addNumbers = coroutine(function* (sink) {
  let sum = 0;
  try {
    while (true) {
      sum += yield;
      sink.next(sum);
    }
  } finally {
    // We received an end-of-stream
    sink.return(); // signal end of stream
  }
});

```

Let's try out this generator:

```

> send([5, -2, 12], addNumbers(logItems()));
5
3
15
DONE

```

22.6.2.2.4 Pushing the input

The chain of generators starts with `tokenize` and ends with `logItems`, which logs everything it receives. We push a sequence of characters into the chain via `send`:

```

const INPUT = '2 apples and 5 oranges.';
const CHAIN = tokenize(extractNumbers(addNumbers(logItems())));
send(INPUT, CHAIN);

// Output
// 2
// 7
// DONE

```

The following code proves that processing really happens lazily:

```

const CHAIN2 = tokenize(extractNumbers(addNumbers(logItems({ prefix: '-> ' })))\
));
send(INPUT, CHAIN2, { log: true });

// Output
// 2
//
// -> 2
// a
// p
// p
// l
// e
// s
//
// a
// n
// d
//
// 5
//
// -> 7
// o
// r
// a
// n
// g
// e
// s

```

```
// .  
// DONE
```

The output shows that `addNumbers` produces a result as soon as the characters `'2'` and `' '` are pushed.

22.6.3 Cooperative multi-tasking via generators

22.6.3.1 Pausing long-running tasks

In this example, we create a counter that is displayed on a web page. We improve an initial version until we have a cooperatively multitasked version that doesn't block the main thread and the user interface.

This is the part of the web page in which the counter should be displayed:

```
<body>  
  Counter: <span id="counter"></span>  
</body>
```

This function displays a counter that counts up forever⁵:

```
function countUp(start = 0) {  
  const counterSpan = document.querySelector('#counter');  
  while (true) {  
    counterSpan.textContent = String(start);  
    start++;  
  }  
}
```

If you ran this function, it would completely block the user interface thread in which it runs and its tab would become unresponsive.

Let's implement the same functionality via a generator that periodically pauses via `yield` (a scheduling function for running this generator is shown later):

```
function* countUp(start = 0) {  
  const counterSpan = document.querySelector('#counter');  
  while (true) {  
    counterSpan.textContent = String(start);  
    start++;  
    yield; // pause  
  }  
}
```

Let's add one small improvement. We move the update of the user interface to another generator, `displayCounter`, which we call via `yield*`. As it is a generator, it can also take care of pausing.

```
function* countUp(start = 0) {  
  while (true) {  
    start++;  
    yield* displayCounter(start);  
  }  
}  
  
function* displayCounter(counter) {  
  const counterSpan = document.querySelector('#counter');  
  counterSpan.textContent = String(counter);  
  yield; // pause  
}
```

Lastly, this is a scheduling function that we can use to run `countUp()`. Each execution step of the generator is handled by a separate task, which is created via `setTimeout()`. That means that the user interface can schedule other tasks in between and will remain responsive.

```
function run(generatorObject) {  
  if (!generatorObject.next().done) {  
    // Add a new task to the event queue  
    setTimeout(function () {  
      run(generatorObject);  
    }, 1000);  
  }  
}
```

With the help of `run`, we get a (nearly) infinite count-up that doesn't block the user interface:

```
run(countUp());
```



You can [run this example online](#).

22.6.3.2 Cooperative multitasking with generators and Node.js-style callbacks

If you call a generator function (or method), it does not have access to its generator object; its `this` is the `this` it would have if it were a non-generator function. A workaround is to pass the generator object into the generator function via `yield`.

The following Node.js script uses this technique, but wraps the generator object in a callback (next, line A). It must be run via `babel-node`.

```
import {readFile} from 'fs';

const fileNames = process.argv.slice(2);

run(function* () {
  const next = yield;
  for (const f of fileNames) {
    const contents = yield readFile(f, { encoding: 'utf8' }, next);
    console.log('#### ' + f);
    console.log(contents);
  }
});
```

In line A, we get a callback that we can use with functions that follow Node.js callback conventions. The callback uses the generator object to wake up the generator, as you can see in the implementation of `run()`:

```
function run(generatorFunction) {
  const generatorObject = generatorFunction();

  // Step 1: Proceed to first `yield`
  generatorObject.next();

  // Step 2: Pass in a function that the generator can use as a callback
  function nextFunction(error, result) {
    if (error) {
      generatorObject.throw(error);
    } else {
      generatorObject.next(result);
    }
  }
  generatorObject.next(nextFunction);

  // Subsequent invocations of `next()` are triggered by `nextFunction`
}
```

22.6.3.3 Communicating Sequential Processes (CSP)

The library `js-csp` brings Communicating Sequential Processes (CSP) to JavaScript, a style of cooperative multitasking that is similar to ClojureScript's `core.async` and Go's `goroutines`. `js-csp` has two abstractions:

- Processes: are cooperatively multitasked tasks and implemented by handing a generator function to the scheduling function `go()`.
- Channels: are queues for communication between processes. Channels are created by calling `chan()`.

As an example, let's use CSP to handle DOM events, in a manner reminiscent of Functional Reactive Programming. The following code uses the function `listen()` (which is shown later) to create a channel that outputs `mousemove` events. It then continuously retrieves the output via `take`, inside an infinite loop. Thanks to `yield`, the process blocks until the channel has output.

```
import csp from 'js-csp';

csp.go(function* () {
  const element = document.querySelector('#uiElement1');
  const channel = listen(element, 'mousemove');
  while (true) {
    const event = yield csp.take(channel);
    const x = event.layerX || event.clientX;
    const y = event.layerY || event.clientY;
    element.textContent = `${x}, ${y}`;
  }
});
```

`listen()` is implemented as follows.

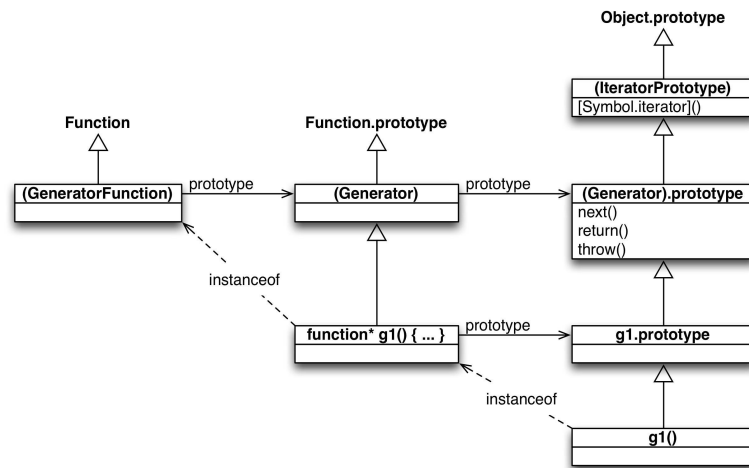
```
function listen(element, type) {
  const channel = csp.chan();
  element.addEventListener(type,
    event => {
      csp.putAsync(channel, event);
    });
  return channel;
}
```



This example is taken from the blog post “[Taming the Asynchronous Beast with CSP Channels in JavaScript](#)” by James Long. Consult this blog post for more information on CSP.

22.7 Inheritance within the iteration API (including generators)

This is a diagram of how various objects are connected in ECMAScript 6 (it is based on [Allen Wirf-Brock's diagram](#) in the ECMAScript specification):



Legend:

- The white (hollow) arrows express the has-prototype relationship (inheritance) between objects. In other words: a white arrow from x to y means that `Object.getPrototypeOf(x) === y`.
- Parentheses indicate that an object exists, but is not accessible via a global variable.
- An instanceof arrow from x to y means that `x instanceof y`.
 - Remember that `x instanceof C` is equivalent to `C.prototype.isPrototypeOf(x)`.
- A prototype arrow from x to y means that `x.prototype === y`.
- The right column shows an instance with its prototypes, the middle column shows a function and its prototypes, the left column shows classes for functions (metafunctions, if you will), connected via a subclass-of relationship.

The diagram reveals two interesting facts:

First, a generator function `g` works very much like a constructor (however, you can't invoke it via `new`; that causes a `TypeError`): The generator objects it creates are instances of it, methods added to `g.prototype` become prototype methods, etc.:

```
> function* g() {}
> g.prototype.hello = function () { return 'hi!'; };
> const obj = g();
> obj instanceof g
true
> obj.hello()
'hi!'
```

Second, if you want to make methods available for all generator objects, it's best to add them to `(Generator).prototype`. One way of accessing that object is as follows:

```
const Generator = Object.getPrototypeOf(function* () {});
Generator.prototype.hello = function () { return 'hi!'; };
const generatorObject = (function* () {}());
generatorObject.hello(); // 'hi!'
```

22.7.1 IteratorPrototype

There is no `(Iterator)` in the diagram, because no such object exists. But, given how `instanceof` works and because `(IteratorPrototype)` is a prototype of `g1()`, you could still say that `g1()` is an instance of `Iterator`.

All iterators in ES6 have `(IteratorPrototype)` in their prototype chain. That object is iterable, because it has the following method. Therefore, all ES6 iterators are iterable (as a consequence, you can apply `for-of` etc. to them).

```
[Symbol.iterator]() {
  return this;
}
```

The specification recommends to use the following code to access `(IteratorPrototype)`:

```
const proto = Object.getPrototypeOf.bind(Object);
const IteratorPrototype = proto(proto([][Symbol.iterator]()));
```

You could also use:

```
const IteratorPrototype = proto(proto(function* () {}().prototype));
```

Quoting the ECMAScript 6 specification:

ECMAScript code may also define objects that inherit from `IteratorPrototype`. The `IteratorPrototype` object provides a place where additional methods that are applicable to all iterator objects may be added.

IteratorPrototype will probably become directly accessible in an upcoming version of ECMAScript and contain tool methods such as `map()` and `filter()` ([source](#)).

22.7.2 The value of `this` in generators

A generator function combines two concerns:

1. It is a function that sets up and returns a generator object.
2. It contains the code that the generator object steps through.

That's why it's not immediately obvious what the value of `this` should be inside a generator.

In function calls and method calls, `this` is what it would be if `gen()` wasn't a generator function, but a normal function:

```
function* gen() {
  'use strict'; // just in case
  yield this;
}

// Retrieve the yielded value via destructuring
const [functionThis] = gen();
console.log(functionThis); // undefined

const obj = { method: gen };
const [methodThis] = obj.method();
console.log(methodThis === obj); // true
```

If you access `this` in a generator that was invoked via `new`, you get a `ReferenceError` ([source: ES6 spec](#)):

```
function* gen() {
  console.log(this); // ReferenceError
}
new gen();
```

A work-around is to wrap the generator in a normal function that hands the generator its generator object via `next()`. That means that the generator must use its first `yield` to retrieve its generator object:

```
const generatorObject = yield;
```

22.8 Style consideration: whitespace before and after the asterisk

Reasonable – and legal – variations of formatting the asterisk are:

- A space before and after it:
`function * foo(x, y) { ... }`
- A space before it:
`function *foo(x, y) { ... }`
- A space after it:
`function* foo(x, y) { ... }`
- No whitespace before and after it:
`function*foo(x, y) { ... }`

Let's figure out which of these variations make sense for which constructs and why.

22.8.1 Generator function declarations and expressions

Here, the star is only used because `generator` (or something similar) isn't available as a keyword. If it were, then a generator function declaration would look like this:

```
generator foo(x, y) {
  ...
}
```

Instead of `generator`, ECMAScript 6 marks the `function` keyword with an asterisk. Thus, `function*` can be seen as a synonym for `generator`, which suggests writing generator function declarations as follows.

```
function* foo(x, y) {
  ...
}
```

Anonymous generator function expressions would be formatted like this:

```
const foo = function* (x, y) {
  ...
}
```

22.8.2 Generator method definitions

When writing generator method definitions, I recommend to format the asterisk as follows.

```
const obj = {
  * generatorMethod(x, y) {
    ...
  }
}
```

```
    }
};
```

There are three arguments in favor of writing a space after the asterisk.

First, the asterisk shouldn't be part of the method name. On one hand, it isn't part of the name of a generator function. On the other hand, the asterisk is only mentioned when defining a generator, not when using it.

Second, a generator method definition is an abbreviation for the following syntax. (To make my point, I'm redundantly giving the function expression a name, too.)

```
const obj = {
  generatorMethod: function* generatorMethod(x, y) {
    ...
  }
};
```

If method definitions are about omitting the `function` keyword then the asterisk should be followed by a space.

Third, generator method definitions are syntactically similar to getters and setters (which are already available in ECMAScript 5):

```
const obj = {
  get foo() {
    ...
  }
  set foo(value) {
    ...
  }
};
```

The keywords `get` and `set` can be seen as modifiers of a normal method definition. Arguably, an asterisk is also such a modifier.

22.8.3 Formatting recursive `yield`

The following is an example of a generator function yielding its own yielded values recursively:

```
function* foo(x) {
  ...
  yield* foo(x - 1);
  ...
}
```

The asterisk marks a different kind of `yield` operator, which is why the above way of writing it makes sense.

22.8.4 Documenting generator functions and methods

Kyle Simpson (@getify) proposed something interesting: Given that we often append parentheses when we write about functions and methods such as `Math.max()`, wouldn't it make sense to prepend an asterisk when writing about generator functions and methods? For example: should we write `*foo()` to refer to the generator function in the previous subsection? Let me argue against that.

When it comes to writing a function that returns an iterable, a generator is only one of the several options. I think it is better to not give away this implementation detail via marked function names.

Furthermore, you don't use the asterisk when calling a generator function, but you do use parentheses.

Lastly, the asterisk doesn't provide useful information – `yield*` can also be used with functions that return an iterable. But it may make sense to mark the names of functions and methods (including generators) that return iterables. For example, via the suffix `Iter`.

22.9 FAQ: generators

22.9.1 Why use the keyword `function*` for generators and not `generator`?

Due to backward compatibility, using the keyword `generator` wasn't an option. For example, the following code (a hypothetical ES6 anonymous generator expression) could be an ES5 function call followed by a code block.

```
generator (a, b, c) {
  ...
}
```

I find that the asterisk naming scheme extends nicely to `yield*`.

22.9.2 Is `yield` a keyword?

`yield` is only a reserved word in strict mode. A trick is used to bring it to ES6 sloppy mode: it becomes a *contextual keyword*, one that is only available inside generators.

22.10 Conclusion

I hope that this chapter convinced you that generators are a useful and versatile tool.

I like that generators let you implement cooperatively multitasked tasks that block while making asynchronous function calls. In my opinion that's the right mental model for async calls. Hopefully, JavaScript goes further in this direction in the future.

22.11 Further reading

Sources of this chapter:

[1] [“Async Generator Proposal”](#) by Jafar Husain

[2] [“A Curious Course on Coroutines and Concurrency”](#) by David Beazley

[3] [“Why coroutines won't work on the web”](#) by David Herman

V Standard library

23. New regular expression features

This chapter explains new regular expression features in ECMAScript 6. It helps if you are familiar with ES5 regular expression features and Unicode. Consult the following two chapters of “Speaking JavaScript” if necessary:

- [“Regular Expressions”](#)
- [“Unicode and JavaScript”](#)

23.1 Overview

The following regular expression features are new in ECMAScript 6:

- The new flag `/y` (sticky) anchors each match of a regular expression to the end of the previous match.
- The new flag `/u` (unicode) handles surrogate pairs (such as `\uD83D\uDE80`) as code points and lets you use Unicode code point escapes (such as `\u{1F680}`) in regular expressions.
- The new data property `flags` gives you access to the flags of a regular expression, just like `source` already gives you access to the pattern in ES5:

```
> /abc/ig.source // ES5
'abc'
> /abc/ig.flags // ES6
'gi'
```

- You can use the constructor `RegExp()` to make a copy of a regular expression:

```
> new RegExp(/abc/ig).flags
'gi'
> new RegExp(/abc/ig, 'i').flags // change flags
'i'
```

23.2 New flag `/y` (sticky)

The new flag `/y` changes two things while matching a regular expression `re` against a string:

- Anchored to `re.lastIndex`: The match must start at `re.lastIndex` (the index after the previous match). This behavior is similar to the `^` anchor, but with that anchor, matches must always start at index 0.
- Match repeatedly: If a match was found, `re.lastIndex` is set to the index after the match. This behavior is similar to the `/g` flag. Like `/g`, `/y` is normally used to match multiple times.

The main use case for this matching behavior is tokenizing, where you want each match to immediately follow its predecessor. An example of tokenizing via a sticky regular expression and `exec()` is given later.

Let's look at how various regular expression operations react to the `/y` flag. The following tables give an overview. I'll provide more details afterwards.

Methods of regular expressions (`re` is the regular expression that a method is invoked on):

| | Flags | Start matching | Anchored to | Result if match | No match | re.lastIndex |
|--------|-------|----------------|---------------|-----------------|----------|-------------------|
| exec() | – | 0 | – | Match object | null | unchanged |
| | /g | re.lastIndex | – | Match object | null | index after match |
| | /y | re.lastIndex | re.lastIndex | Match object | null | index after match |
| | /gy | re.lastIndex | re.lastIndex | Match object | null | index after match |
| test() | (Any) | (like exec()) | (like exec()) | true | false | (like exec()) |

Methods of strings (*str* is the string that a method is invoked on, *r* is the regular expression parameter):

| | Flags | Start matching | Anchored to | Result if match | No match | r.lastIndex |
|-----------|---------|----------------|-------------|---------------------------------------|----------|-------------------|
| search() | –, /g | 0 | – | Index of match | -1 | unchanged |
| | /y, /gy | 0 | 0 | Index of match | -1 | unchanged |
| match() | – | 0 | – | Match object | null | unchanged |
| | /y | r.lastIndex | r.lastIndex | Match object | null | index after match |
| | /g | After prev. | – | Array with matches | null | 0 |
| | | match (loop) | | | | |
| | /gy | After prev. | After prev. | Array with matches | null | 0 |
| | | match (loop) | match | | | |
| split() | –, /g | After prev. | – | Array with strings between matches | [str] | unchanged |
| | | match (loop) | | | | |
| | /y, /gy | After prev. | After prev. | Arr. w/ empty strings between matches | [str] | unchanged |
| | | match (loop) | match | | | |
| replace() | – | 0 | – | First match replaced | No repl. | unchanged |
| | /y | 0 | 0 | First match replaced | No repl. | unchanged |
| | /g | After prev. | – | All matches replaced | No repl. | unchanged |
| | | match (loop) | | | | |
| | /gy | After prev. | After prev. | All matches replaced | No repl. | unchanged |
| | | match (loop) | match | | | |

23.2.1 RegExp.prototype.exec(str)

If */g* is not set, matching always starts at the beginning, but skips ahead until a match is found. `REGEX.lastIndex` is not changed.

```
const REGEX = /a/;

REGEX.lastIndex = 7; // ignored
const match = REGEX.exec('xaxa');
console.log(match.index); // 1
console.log(REGEX.lastIndex); // 7 (unchanged)
```

If */g* is set, matching starts at `REGEX.lastIndex` and skips ahead until a match is found. `REGEX.lastIndex` is set to the position after the match. That means that you receive all matches if you loop until `exec()` returns `null`.

```
const REGEX = /a/g;

REGEX.lastIndex = 2;
const match = REGEX.exec('xaxa');
console.log(match.index); // 3
console.log(REGEX.lastIndex); // 4 (updated)
```

```
// No match at index 4 or later
console.log(REGEX.exec('xaxa')); // null
```

If only `/y` is set, matching starts at `REGEX.lastIndex` and is anchored to that position (no skipping ahead until a match is found). `REGEX.lastIndex` is updated similarly to when `/g` is set.

```
const REGEX = /a/y;

// No match at index 2
REGEX.lastIndex = 2;
console.log(REGEX.exec('xaxa')); // null

// Match at index 3
REGEX.lastIndex = 3;
const match = REGEX.exec('xaxa');
console.log(match.index); // 3
console.log(REGEX.lastIndex); // 4
```

Setting both `/y` and `/g` is the same as only setting `/y`.

23.2.2 `RegExp.prototype.test(str)`

`test()` works the same as `exec()`, but it returns `true` or `false` (instead of a match object or `null`) when matching succeeds or fails:

```
const REGEX = /a/y;

REGEX.lastIndex = 2;
console.log(REGEX.test('xaxa')); // false

REGEX.lastIndex = 3;
console.log(REGEX.test('xaxa')); // true
console.log(REGEX.lastIndex); // 4
```

23.2.3 `String.prototype.search(regex)`

`search()` ignores the flag `/g` and `lastIndex` (which is not changed, either). Starting at the beginning of the string, it looks for the first match and returns its index (or `-1` if there was no match):

```
const REGEX = /a/;

REGEX.lastIndex = 2; // ignored
console.log('xaxa'.search(REGEX)); // 1
```

If you set the flag `/y`, `lastIndex` is still ignored, but the regular expression is now anchored to index 0.

```
const REGEX = /a/y;

REGEX.lastIndex = 1; // ignored
console.log('xaxa'.search(REGEX)); // -1 (no match)
```

23.2.4 `String.prototype.match(regex)`

`match()` has two modes:

- If `/g` is not set, it works like `exec()`.
- If `/g` is set, it returns an Array with the string parts that matched, or `null`.

If the flag `/g` is not set, `match()` captures groups like `exec()`:

```
{
  const REGEX = /a/;

  REGEX.lastIndex = 7; // ignored
  console.log('xaxa'.match(REGEX).index); // 1
  console.log(REGEX.lastIndex); // 7 (unchanged)
}
{
  const REGEX = /a/y;

  REGEX.lastIndex = 2;
  console.log('xaxa'.match(REGEX)); // null

  REGEX.lastIndex = 3;
  console.log('xaxa'.match(REGEX).index); // 3
  console.log(REGEX.lastIndex); // 4
}
```

If only the flag `/g` is set then `match()` returns all matching substrings in an Array (or `null`). Matching always starts at position 0.

```
const REGEX = /a|b/g;
REGEX.lastIndex = 7;
console.log('xaxb'.match(REGEX)); // ['a', 'b']
console.log(REGEX.lastIndex); // 0
```

If you additionally set the flag `/y`, then matching is still performed repeatedly, while anchoring the regular expression to the index after the previous match (or 0).

```
const REGEX = /a|b/gy;

REGEX.lastIndex = 0; // ignored
```

```

console.log('xab'.match(REGEX)); // null
REGEX.lastIndex = 1; // ignored
console.log('xab'.match(REGEX)); // null

console.log('ab'.match(REGEX)); // ['a', 'b']
console.log('axb'.match(REGEX)); // ['a']

```

23.2.5 String.prototype.split(separator, limit)

The complete details of `split()` are explained in [Speaking JavaScript](#).

For ES6, it is interesting to see how things change if you use the flag `/y`.

With `/y`, the string must start with a separator:

```

> 'x##'.split(/#/y) // no match
[ 'x##' ]
> '##x'.split(/#/y) // 2 matches
[ '', '', 'x' ]

```

Subsequent separators are only recognized if they immediately follow the first separator:

```

> '##x'.split(/#/y) // 1 match
[ '', 'x#' ]
> '##'.split(/#/y) // 2 matches
[ '', '', '' ]

```

That means that the string before the first separator and the strings between separators are always empty.

As usual, you can use groups to put parts of the separators into the result array:

```

> '##'.split(/(##)/y)
[ '', '#', '', '#', '' ]

```

23.2.6 String.prototype.replace(search, replacement)

Without the flag `/g`, `replace()` only replaces the first match:

```

const REGEX = /a/;

// One match
console.log('xaxa'.replace(REGEX, '-')); // 'x-xa'

```

If only `/y` is set, you also get at most one match, but that match is always anchored to the beginning of the string. `lastIndex` is ignored and unchanged.

```

const REGEX = /a/y;

// Anchored to beginning of string, no match
REGEX.lastIndex = 1; // ignored
console.log('xaxa'.replace(REGEX, '-')); // 'xaxa'
console.log(REGEX.lastIndex); // 1 (unchanged)

// One match
console.log('axa'.replace(REGEX, '-')); // '-xa'

```

With `/g` set, `replace()` replaces all matches:

```

const REGEX = /a/g;

// Multiple matches
console.log('xaxa'.replace(REGEX, '-')); // 'x-x-'

```

With `/gy` set, `replace()` replaces all matches, but each match is anchored to the end of the previous match:

```

const REGEX = /a/gy;

// Multiple matches
console.log('aaxa'.replace(REGEX, '-')); // '--xa'

```

The parameter `replacement` can also be a function, consult [“Speaking JavaScript” for details](#).

23.2.7 Example: using sticky matching for tokenizing

The main use case for sticky matching is *tokenizing*, turning a text into a sequence of tokens. One important trait about tokenizing is that tokens are fragments of the text and that there must be no gaps between them. Therefore, sticky matching is perfect here.

```

function tokenize(TOKEN_REGEX, str) {
  const result = [];
  let match;
  while (match = TOKEN_REGEX.exec(str)) {
    result.push(match[1]);
  }
  return result;
}

const TOKEN_GY = /\s*(\+[0-9]+\s*)/gy;
const TOKEN_G = /\s*(\+[0-9]+\s*)/g;

```

In a legal sequence of tokens, sticky matching and non-sticky matching produce the same output:

```
> tokenize(TOKEN_GY, '3 + 4')
[ '3', '+', '4' ]
> tokenize(TOKEN_G, '3 + 4')
[ '3', '+', '4' ]
```

If, however, there is non-token text in the string then sticky matching stops tokenizing, while non-sticky matching skips the non-token text:

```
> tokenize(TOKEN_GY, '3x + 4')
[ '3' ]
> tokenize(TOKEN_G, '3x + 4')
[ '3', '+', '4' ]
```

The behavior of sticky matching during tokenizing helps with error handling.

23.2.8 Example: manually implementing sticky matching

If you wanted to manually implement sticky matching, you'd do it as follows: The function `execSticky()` works like `RegExp.prototype.exec()` in sticky mode.

```
function execSticky(regex, str) {
  // Anchor the regex to the beginning of the string
  let matchSource = regex.source;
  if (!matchSource.startsWith('^')) {
    matchSource = '^' + matchSource;
  }
  // Ensure that instance property `lastIndex` is updated
  let matchFlags = regex.flags; // ES6 feature!
  if (!regex.global) {
    matchFlags = matchFlags + 'g';
  }
  const matchRegex = new RegExp(matchSource, matchFlags);

  // Ensure we start matching `str` at `regex.lastIndex`
  const matchOffset = regex.lastIndex;
  const matchStr = str.slice(matchOffset);
  let match = matchRegex.exec(matchStr);

  // Translate indices from `matchStr` to `str`
  regex.lastIndex = matchRegex.lastIndex + matchOffset;
  match.index = match.index + matchOffset;
  return match;
}
```

23.3 New flag `/u` (unicode)

The flag `/u` switches on a special Unicode mode for a regular expression. That mode has two features:

1. You can use Unicode code point escape sequences such as `\u{1F42A}` for specifying characters via code points. Normal Unicode escapes such as `\u03B1` only have a range of four hexadecimal digits (which equals the basic multilingual plane).
2. “characters” in the regular expression pattern and the string are code points (not UTF-16 code units). Code units are converted into code points.

A section in the [chapter on Unicode](#) has more information on escape sequences. I'll explain the consequences of feature 2 next. Instead of Unicode code point escapes (e.g., `\u{1F680}`), I'm using two UTF-16 code units (e.g., `\uD83D\uDE80`). That makes it clear that surrogate pairs are grouped in Unicode mode and works in both Unicode mode and non-Unicode mode.

```
> '\u{1F680}' === '\uD83D\uDE80' // code point vs. surrogate pairs
true
```

23.3.1 Consequence: lone surrogates in the regular expression only match lone surrogates

In non-Unicode mode, a lone surrogate in a regular expression is even found inside (surrogate pairs encoding) code points:

```
> /\uD83D/.test('\uD83D\uD83D')
true
```

In Unicode mode, surrogate pairs become atomic units and lone surrogates are not found “inside” them:

```
> /\uD83D/u.test('\uD83D\uD83D')
false
```

Actual lone surrogate are still found:

```
> /\uD83D/u.test('\uD83D \uD83D\uD83D')
true
> /\uD83D/u.test('\uD83D\uD83D \uD83D')
true
```

23.3.2 Consequence: you can put code points in character classes

In Unicode mode, you can put code points into character classes and they won't be interpreted as two characters, anymore.

```
> /^[uD83D\uDC2A]$/u.test('\uD83D\uDC2A')
true
> /^[uD83D\uDC2A]$/u.test('\uD83D\uDC2A')
false

> /^[uD83D\uDC2A]$/u.test('\uD83D')
false
> /^[uD83D\uDC2A]$/u.test('\uD83D')
true
```

23.3.3 Consequence: the dot operator (.) matches code points, not code units

In Unicode mode, the dot operator matches code points (one or two code units). In non-Unicode mode, it matches single code units. For example:

```
> '\uD83D\uDE80'.match(/./gu).length
1
> '\uD83D\uDE80'.match(/./g).length
2
```

23.3.4 Consequence: quantifiers apply to code points, not code units

In Unicode mode, quantifiers apply to code points (one or two code units). In non-Unicode mode, they apply to single code units. For example:

```
> /\uD83D\uDE80{2}/u.test('\uD83D\uDE80\uD83D\uDE80')
true

> /\uD83D\uDE80{2}/.test('\uD83D\uDE80\uD83D\uDE80')
false
> /\uD83D\uDE80{2}/.test('\uD83D\uDE80\uDE80')
true
```

23.4 New data property `flags`

In ECMAScript 6, regular expressions have the following data properties:

- The pattern: `source`
- The flags: `flags`
- Individual flags: `global`, `ignoreCase`, `multiline`, `sticky`, `unicode`
- Other: `lastIndex`

As an aside, `lastIndex` is the only instance property now, all other data properties are implemented via internal instance properties and getters such as `get RegExp.prototype.global`.

The property `source` (which already existed in ES5) contains the regular expression pattern as a string:

```
> /abc/ig.source
'abc'
```

The property `flags` is new, it contains the flags as a string, with one character per flag:

```
> /abc/ig.flags
'gi'
```

You can't change the flags of an existing regular expression (`ignoreCase` etc. have always been immutable), but `flags` allows you to make a copy where the flags are changed:

```
function copyWithIgnoreCase(regex) {
  return new RegExp(regex.source, regex.flags+'i');
}
```

The next section explains another way to make modified copies of regular expressions.

23.5 `RegExp()` can be used as a copy constructor

In ES6 there are two variants of the constructor `RegExp()` (the second one is new):

- `new RegExp(pattern : string, flags = '')`
A new regular expression is created as specified via `pattern`. If `flags` is missing, the empty string '' is used.
- `new RegExp(regex : RegExp, flags = regex.flags)`
`regex` is cloned. If `flags` is provided then it determines the flags of the copy.

The following interaction demonstrates the latter variant:

```
> new RegExp(/abc/ig).flags
'gi'
> new RegExp(/abc/ig, 'i').flags // change flags
'i'
```

Therefore, the `RegExp` constructor gives us another way to change flags:

```
function copyWithIgnoreCase(regex) {
  return new RegExp(regex, regex.flags+'i');
}
```

23.5.1 Example: an iterable version of `exec()`

The following function `execAll()` is an iterable version of `exec()` that fixes several issues with using `exec()` to retrieve all matches of a regular expression:

- Looping over the matches is unnecessarily complicated (you call `exec()` until it returns `null`).
- `exec()` mutates the regular expression, which means that side effects can become a problem.
- The flag `/g` must be set. Otherwise, only the first match is returned.

```
function* execAll(regex, str) {
  // Make sure flag /g is set and regex.index isn't changed
  const localCopy = new RegExp(regex, regex.flags+'g');
  let match;
  while (match = localCopy.exec(str)) {
    yield match;
  }
}
```

Using `execAll()`:

```
const str = "fee" "fi" "fo" "fum";
const regex = /^([a-z]*)/;

// Access capture of group #1 via destructuring
for (const [, group1] of execAll(regex, str)) {
  console.log(group1);
}

// Output:
// fee
// fi
// fo
// fum
```

23.6 String methods that delegate to regular expression methods

The following string methods now delegate some of their work to regular expression methods:

- `String.prototype.match` calls `RegExp.prototype[Symbol.match]`.
- `String.prototype.replace` calls `RegExp.prototype[Symbol.replace]`.
- `String.prototype.search` calls `RegExp.prototype[Symbol.search]`.
- `String.prototype.split` calls `RegExp.prototype[Symbol.split]`.

For more information, consult Sect. “[String methods that delegate regular expression work to their parameters](#)” in the chapter on strings.



Further reading

If you want to know in more detail how the regular expression flag `/u` works, I recommend the article “[Unicode-aware regular expressions in ECMAScript 6](#)” by Mathias Bynens.

24. Asynchronous programming (background)

This chapter explains foundations of asynchronous programming in JavaScript. It provides background knowledge for [the next chapter on ES6 Promises](#).

24.1 The JavaScript call stack

When a function `f` calls a function `g`, `g` needs to know where to return to (inside `f`) after it is done. This information is usually managed with a stack, the *call stack*. Let’s look at an example.

```
function h(z) {
  // Print stack trace
  console.log(new Error().stack); // (A)
}
function g(y) {
  h(y + 1); // (B)
}
function f(x) {
  g(x + 1); // (C)
}
f(3); // (D)
return; // (E)
```

Initially, when the program above is started, the call stack is empty. After the function call `f(3)` in line D, the stack has one entry:

- Location in global scope

After the function call `g(x + 1)` in line C, the stack has two entries:

- Location in `f`
- Location in global scope

After the function call `h(y + 1)` in line B, the stack has three entries:

- Location in `g`
- Location in `f`
- Location in global scope

The stack trace printed in line A shows you what the call stack looks like:

```
Error
  at h (stack_trace.js:2:17)
  at g (stack_trace.js:6:5)
  at f (stack_trace.js:9:5)
  at <global> (stack_trace.js:11:1)
```

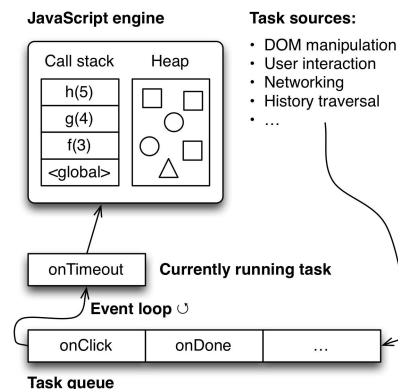
Next, each of the functions terminates and each time the top entry is removed from the stack. After function `f` is done, we are back in global scope and the call stack is empty. In line E we return and the stack is empty, which means that the program terminates.

24.2 The browser event loop

Simplifyingly, each browser tab runs (in) a single process: the *event loop*. This loop executes browser-related things (so-called *tasks*) that it is fed via a *task queue*. Examples of tasks are:

1. Parsing HTML
2. Executing JavaScript code in script elements
3. Reacting to user input (mouse clicks, key presses, etc.)
4. Processing the result of an asynchronous network request

Items 2–4 are tasks that run JavaScript code, via the engine built into the browser. They terminate when the code terminates. Then the next task from the queue can be executed. The following diagram (inspired by [a slide by Philip Roberts \[1\]](#)) gives an overview of how all these mechanisms are connected.



The event loop is surrounded by other processes running in parallel to it (timers, input handling, etc.). These processes communicate with it by adding tasks to its queue.

24.2.1 Timers

Browsers have *timers*. `setTimeout()` creates a timer, waits until it fires and then adds a task to the queue. It has the signature:

```
setTimeout(callback, ms)
```

After `ms` milliseconds, `callback` is added to the task queue. It is important to note that `ms` only specifies when the callback is *added*, not when it actually executed. That may happen much later, especially if the event loop is blocked (as demonstrated later in this chapter).

`setTimeout()` with `ms` set to zero is a commonly used work-around to add something to the task queue right away. However, some browsers do not allow `ms` to be below a minimum (4 ms in Firefox); they set it to that minimum if it is.

24.2.2 Displaying DOM changes

For most DOM changes (especially those involving a re-layout), the display isn't updated right away. "Layout happens off a refresh tick every 16ms" ([@bz_moz](#)) and must be given a chance to run via the event loop.

There are ways to coordinate frequent DOM updates with the browser, to avoid clashing with its layout rhythm. Consult the [documentation](#) on `requestAnimationFrame()` for details.

24.2.3 Run-to-completion semantics

JavaScript has so-called run-to-completion semantics: The current task is always

finished before the next task is executed. That means that each task has complete control over all current state and doesn't have to worry about concurrent modification.

Let's look at an example:

```
setTimeout(function () { // (A)
  console.log('Second');
}, 0);
console.log('First'); // (B)
```

The function starting in line A is added to the task queue immediately, but only executed after the current piece of code is done (in particular line B!). That means that this code's output will always be:

```
First
Second
```

24.2.4 Blocking the event loop

As we have seen, each tab (in some browsers, the complete browser) is managed by a single process – both the user interface and all other computations. That means that you can freeze the user interface by performing a long-running computation in that process. The following code demonstrates that.

```
<a id="block" href="">Block for 5 seconds</a>
<p>
  <button>This is a button</button>
  <div id="statusMessage"></div>
</p>
<script>
  document.getElementById('block')
    .addEventListener('click', onClick);

  function onClick(event) {
    event.preventDefault();

    setStatusMessage('Blocking...');

    // Call setTimeout(), so that browser has time to display
    // status message
    setTimeout(function () {
      sleep(5000);
      setStatusMessage('Done');
    }, 0);
  }

  function setStatusMessage(msg) {
    document.getElementById('statusMessage').textContent = msg;
  }

  function sleep(milliseconds) {
    var start = Date.now();
    while ((Date.now() - start) < milliseconds);
  }
</script>
```



You can try out the code [online](#).

Whenever the link at the beginning is clicked, the function `onClick()` is triggered. It uses the – synchronous – `sleep()` function to block the event loop for five seconds. During those seconds, the user interface doesn't work. For example, you can't click the "Simple button".

24.2.5 Avoiding blocking

You avoid blocking the event loop in two ways:

First, you don't perform long-running computations in the main process, you move them to a different process. This can be achieved via the [Worker API](#).

Second, you don't (synchronously) wait for the results of a long-running computation (your own algorithm in a Worker process, a network request, etc.), you carry on with the event loop and let the computation notify you when it is finished. In fact, you usually don't even have a choice in browsers and have to do things this way. For example, there is no built-in way to sleep synchronously (like the previously implemented `sleep()`). Instead, `setTimeout()` lets you sleep asynchronously.

The next section explains techniques for waiting asynchronously for results.

24.3 Receiving results asynchronously

Two common patterns for receiving results asynchronously are: events and callbacks.

24.3.1 Asynchronous results via events

In this pattern for asynchronously receiving results, you create an object for each request and register event handlers with it: one for a successful computation, another one for handling errors. The following code shows how that works with the XMLHttpRequest API:

```

var req = new XMLHttpRequest();
req.open('GET', url);

req.onload = function () {
  if (req.status == 200) {
    processData(req.response);
  } else {
    console.log('ERROR', req.statusText);
  }
};

req.onerror = function () {
  console.log('Network Error');
};

req.send(); // Add request to task queue

```

Note that the last line doesn't actually perform the request, it adds it to the task queue. Therefore, you could also call that method right after `open()`, before setting up `onload` and `onerror`. Things would work the same, due to JavaScript's run-to-completion semantics.

24.3.1.1 Implicit requests

The browser API IndexedDB has a slightly peculiar style of event handling:

```

var openRequest = indexedDB.open('test', 1);

openRequest.onsuccess = function (event) {
  console.log('Success!');
  var db = event.target.result;
};

openRequest.onerror = function (error) {
  console.log(error);
};

```

You first create a request object, to which you add event listeners that are notified of results. However, you don't need to explicitly queue the request, that is done by `open()`. It is executed after the current task is finished. That is why you can (and in fact must) register event handlers *after* calling `open()`.

If you are used to multi-threaded programming languages, this style of handling requests probably looks strange, as if it may be prone to race conditions. But, due to run to completion, things are always safe.

24.3.1.2 Events don't work well for single results

This style of handling asynchronously computed results is OK if you receive results multiple times. If, however, there is only a single result then the verbosity becomes a problem. For that use case, callbacks have become popular.

24.3.2 Asynchronous results via callbacks

If you handle asynchronous results via callbacks, you pass callback functions as trailing parameters to asynchronous function or method calls.

The following is an example in Node.js. We read the contents of a text file via an asynchronous call to `fs.readFile()`:

```

// Node.js
fs.readFile('myfile.txt', { encoding: 'utf8' },
  function (error, text) { // (A)
    if (error) {
      // ...
    }
    console.log(text);
  });

```

If `readFile()` is successful, the callback in line A receives a result via the parameter `text`. If it isn't, the callback gets an error (often an instance of `Error` or a sub-constructor) via its first parameter.

The same code in classic functional programming style would look like this:

```

// Functional
readFileFunctional('myfile.txt', { encoding: 'utf8' },
  function (text) { // success
    console.log(text);
  },
  function (error) { // failure
    // ...
  });

```

24.3.3 Continuation-passing style

The programming style of using callbacks (especially in the functional manner shown previously) is also called *continuation-passing style* (CPS), because the next step (the *continuation*) is explicitly passed as a parameter. This gives an invoked function more control over what happens next and when.

The following code illustrates CPS:

```

console.log('A');
identity('B', function step2(result2) {
  console.log(result2);
  identity('C', function step3(result3) {
    console.log(result3);
  });
  console.log('D');
});
console.log('E');

// Output: A E B D C

function identity(input, callback) {
  setTimeout(function () {
    callback(input);
  }, 0);
}

```

For each step, the control flow of the program continues inside the callback. This leads to nested functions, which are sometimes referred to as *callback hell*. However, you can often avoid nesting, because JavaScript's function declarations are *hoisted* (their definitions are evaluated at the beginning of their scope). That means that you can call ahead and invoke functions defined later in the program. The following code uses hoisting to flatten the previous example.

```

console.log('A');
identity('B', step2);
function step2(result2) {
  // The program continues here
  console.log(result2);
  identity('C', step3);
  console.log('D');
}
function step3(result3) {
  console.log(result3);
}
console.log('E');

```

[More information on CPS is given in \[3\].](#)

24.3.4 Composing code in CPS

In normal JavaScript style, you compose pieces of code via:

1. Putting them one after another. This is blindingly obvious, but it's good to remind ourselves that concatenating code in normal style is sequential composition.
2. Array methods such as `map()`, `filter()` and `forEach()`
3. Loops such as `for` and `while`

The library [Async.js](#) provides combinators to let you do similar things in CPS, with Node.js-style callbacks. It is used in the following example to load the contents of three files, whose names are stored in an Array.

```

var async = require('async');

var fileNames = [ 'foo.txt', 'bar.txt', 'baz.txt' ];
async.map(fileNames,
  function (fileName, callback) {
    fs.readFile(fileName, { encoding: 'utf8' }, callback);
  },
  // Process the result
  function (error, textArray) {
    if (error) {
      console.log(error);
      return;
    }
    console.log('TEXTS:\n' + textArray.join('\n----\n'));
  });

```

24.3.5 Pros and cons of callbacks

Using callbacks results in a radically different programming style, CPS. The main advantage of CPS is that its basic mechanisms are easy to understand. But there are also disadvantages:

- Error handling becomes more complicated: There are now two ways in which errors are reported – via callbacks and via exceptions. You have to be careful to combine both properly.
- Less elegant signatures: In synchronous functions, there is a clear separation of concerns between input (parameters) and output (function result). In asynchronous functions that use callbacks, these concerns are mixed: the function result doesn't matter and some parameters are used for input, others for output.
- Composition is more complicated: Because the concern “output” shows up in the parameters, it is more complicated to compose code via combinators.

Callbacks in Node.js style have three disadvantages (compared to those in a functional style):

- The `if` statement for error handling adds verbosity.
- Reusing error handlers is harder.
- Providing a default error handler is also harder. A default error handler is useful if you make a function call and don't want to write your own handler. It could also be used by a function if a caller doesn't specify a handler.

24.4 Looking ahead

The next chapter covers Promises and the ES6 Promise API. Promises are more complicated under the hood than callbacks. In exchange, they bring several significant advantages and eliminate most of the aforementioned cons of callbacks.

24.5 Further reading

[1] [“Help, I’m stuck in an event-loop”](#) by Philip Roberts (video).

[2] [“Event loops”](#) in the HTML Specification.

[3] [“Asynchronous programming and continuation-passing style in JavaScript”](#) by Axel Rauschmayer.

25. Promises for asynchronous programming

This chapter is an introduction to asynchronous programming via Promises in general and the ECMAScript 6 Promise API in particular. [The previous chapter](#) explains the foundations of asynchronous programming in JavaScript. You can consult it whenever there is something that you don’t understand in this chapter.

25.1 Overview

Promises are an alternative to callbacks for delivering the results of an asynchronous computation. They require more effort from implementors of asynchronous functions, but provide several benefits for users of those functions.

The following function returns a result asynchronously, via a Promise:

```
function asyncFunc() {
  return new Promise(
    function (resolve, reject) {
      ...
      resolve(result);
      ...
      reject(error);
    });
}
```

You call `asyncFunc()` as follows:

```
asyncFunc()
  .then(result => { ... })
  .catch(error => { ... });
```

25.1.1 Chaining `then()` calls

`then()` always returns a Promise, which enables you to chain method calls:

```
asyncFunc1()
  .then(result1 => {
    // Use result1
    return asyncFunction2(); // (A)
  })
  .then(result2 => { // (B)
    // Use result2
  })
  .catch(error => {
    // Handle errors of asyncFunc1() and asyncFunc2()
  });
```

How the Promise `P` returned by `then()` is settled depends on what its callback does:

- If it returns a Promise (as in line A), the settlement of that Promise is forwarded to `P`. That’s why the callback from line B can pick up the settlement of `asyncFunction2`’s Promise.
- If it returns a different value, that value is used to settle `P`.
- If it throws an exception then `P` is rejected with that exception.

Furthermore, note how `catch()` handles the errors of two asynchronous function calls (`asyncFunction1()` and `asyncFunction2()`). That is, uncaught errors are passed on until there is an error handler.

25.1.2 Executing asynchronous functions in parallel

If you chain asynchronous function calls via `then()`, they are executed sequentially, one at a time:

```
asyncFunc1()
  .then(() => asyncFunc2());
```

If you don’t do that and call all of them immediately, they are basically executed in parallel (a *fork* in Unix process terminology):

```
asyncFunc1();
asyncFunc2();
```

`Promise.all()` enables you to be notified once all results are in (a *join* in Unix process terminology). Its input is an Array of Promises, its output a single Promise that is fulfilled with an Array of the results.

```
Promise.all([
  asyncFunc1(),
  asyncFunc2(),
])
.then(([result1, result2]) => {
  ...
})
.catch(err => {
  // Receives first rejection among the Promises
  ...
});
```

25.1.3 Glossary: Promises

The Promise API is about delivering results asynchronously. A *Promise object* (short: Promise) is a stand-in for the result, which is delivered via that object.

States:

- A Promise is always in one of three mutually exclusive states:
 - Before the result is ready, the Promise is *pending*.
 - If a result is available, the Promise is *fulfilled*.
 - If an error happened, the Promise is *rejected*.
- A Promise is *settled* if “things are done” (if it is either fulfilled or rejected).
- A Promise is settled exactly once and then remains unchanged.

Reacting to state changes:

- *Promise reactions* are callbacks that you register with the Promise method `then()`, to be notified of a fulfillment or a rejection.
- A *thenable* is an object that has a Promise-style `then()` method. Whenever the API is only interested in being notified of settlements, it only demands thenables (e.g. the values returned from `then()` and `catch()`; or the values handed to `Promise.all()` and `Promise.race()`).

Changing states: There are two operations for changing the state of a Promise. After you have invoked either one of them once, further invocations have no effect.

- *Rejecting* a Promise means that the Promise becomes rejected.
- *Resolving* a Promise has different effects, depending on what value you are resolving with:
 - Resolving with a normal (non-thenable) value fulfills the Promise.
 - Resolving a Promise P with a thenable T means that P can't be resolved anymore and will now follow T's state, including its fulfillment or rejection value. The appropriate P reactions will get called once T settles (or are called immediately if T is already settled).

25.2 Introduction: Promises

Promises are a pattern that helps with one particular kind of asynchronous programming: a function (or method) that returns its result asynchronously. One popular technique for implementing such a function in JavaScript is to have it pass the result to a callback (“callbacks as continuations”):

```
asyncFunction(arg1, arg2,
  result => {
    console.log(result);
  });
```

Promises provide a better way of working with callbacks: Now an asynchronous function returns a *Promise*, an object that serves as a placeholder for the final result. Callbacks registered via the Promise method `then()` are notified of the result:

```
asyncFunction(arg1, arg2)
.then(result => {
  console.log(result);
});
```

Compared to callbacks as continuations, Promises have the following advantages:

- No inversion of control: similarly to synchronous code, Promise-based functions return results, they don't (directly) continue – and control – execution via callbacks.
- Chaining is simpler: If `then()` detects that its callback triggered another asynchronous operation and returned a Promise, it returns that Promise. That means that you can immediately follow it with another `then()` method call:

```
asyncFunction1(a, b)
.then(result1 => {
  console.log(result1);
  return asyncFunction2(x, y);
})
.then(result2 => {
```

```

        console.log(result2);
    });

```

- Composing asynchronous calls (loops, mapping, etc.): is a little easier, because you have data (Promise objects) you can work with.
- Error handling: As we shall see later, error handling is simpler with Promises, because both exceptions and asynchronous errors are managed the same way.
- Cleaner signature: With callbacks, the parameters of a function are mixed; some are input for the function, others are responsible for delivering its output. With Promises, function signatures become cleaner; all parameters are input.

The de-facto standard for JavaScript Promises is called [Promises/A+ \[1\]](#). The ECMAScript 6 Promise API follows that standard.

25.3 A first example

Let's look at a first example, to give you a taste of what working with Promises is like.

With Node.js-style callbacks, reading a file asynchronously looks like this:

```

fs.readFile('config.json',
  function (error, text) {
    if (error) {
      console.error('Error while reading config file');
    } else {
      try {
        const obj = JSON.parse(text);
        console.log(JSON.stringify(obj, null, 4));
      } catch (e) {
        console.error('Invalid JSON in file');
      }
    }
  });

```

With Promises, the same functionality is used like this:

```

readFilePromisified('config.json')
  .then(function (text) { // (A)
    const obj = JSON.parse(text);
    console.log(JSON.stringify(obj, null, 4));
  })
  .catch(function (error) { // (B)
    // File read error or JSON SyntaxError
    console.error('An error occurred', error);
  });

```

There are still callbacks, but they are provided via methods that are invoked on the result (`then()` and `catch()`). The error callback in line B is convenient in two ways: First, it's a single style of handling errors (versus `if (error)` and `try-catch` in the previous example). Second, you can handle the errors of both `readFilePromisified()` and the callback in line A from a single location.

The code of `readFilePromisified()` is [shown later](#).

25.4 Creating and using Promises

Let's look at how Promises are operated from the producer and the consumer side.

25.4.1 Producing a Promise

As a producer, you create a Promise and send a result via it:

```

const p = new Promise(
  function (resolve, reject) { // (A)
    ...
    if (...) {
      resolve(value); // success
    } else {
      reject(reason); // failure
    }
  });

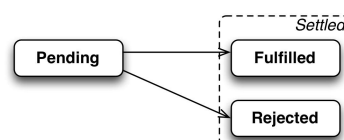
```

25.4.2 The states of Promises

Once a result was delivered via a Promise, the Promise stays locked in to that result. That means each Promise is always in either one of three (mutually exclusive) states:

- Pending: the result hasn't been computed, yet (the initial state of each Promise)
- Fulfilled: the result was computed successfully
- Rejected: a failure occurred during computation

A Promise is *settled* (the computation it represents has finished) if it is either fulfilled or rejected. A Promise can only be settled once and then stays settled. Subsequent attempts to settle have no effect.



The parameter of `new Promise()` (starting in line A) is called an *executor*:

- **Resolving:** If the computation went well, the executor sends the result via `resolve()`. That usually fulfills the Promise `p`. But it may not – resolving with a Promise `q` leads to `p` tracking `q`: If `q` is still pending then so is `p`. However `q` is settled, `p` will be settled the same way.
- **Rejecting:** If an error happened, the executor notifies the Promise consumer via `reject()`. That always rejects the Promise.

If an exception is thrown inside the executor, `p` is rejected with that exception.

25.4.3 Consuming a Promise

As a consumer of `promise`, you are notified of a fulfillment or a rejection via *reactions* – callbacks that you register with the methods `then()` and `catch()`:

```
promise
  .then(value => { /* fulfillment */ })
  .catch(error => { /* rejection */ });
```

What makes Promises so useful for asynchronous functions (with one-off results) is that once a Promise is settled, it doesn't change anymore. Furthermore, there are never any race conditions, because it doesn't matter whether you invoke `then()` or `catch()` before or after a Promise is settled:

- Reactions that are registered with a Promise before it is settled, are notified of the settlement once it happens.
- Reactions that are registered with a Promise after it is settled, receive the cached settled value “immediately” (their invocations are queued as tasks).

Note that `catch()` is simply a more convenient (and recommended) alternative to calling `then()`. That is, the following two invocations are equivalent:

```
promise.then(
  null,
  error => { /* rejection */ });

promise.catch(
  error => { /* rejection */ });
```

25.4.4 Promises are always asynchronous

A Promise library has complete control over whether results are delivered to Promise reactions synchronously (right away) or asynchronously (after the current continuation, the current piece of code, is finished). However, the Promises/A+ specification demands that the latter mode of execution be always used. It states so via the following [requirement](#) (2.2.4) for the `then()` method:

`onFulfilled` or `onRejected` must not be called until the execution context stack contains only platform code.

That means that your code can rely on run-to-completion semantics (as explained in [the previous chapter](#)) and that chaining Promises won't starve other tasks of processing time.

Additionally, this constraint prevents you from writing functions that sometimes return results immediately, sometimes asynchronously. This is an anti-pattern, because it makes code unpredictable. For more information, consult “[Designing APIs for Asynchrony](#)” by Isaac Z. Schlueter.

25.5 Examples

Before we dig deeper into Promises, let's use what we have learned so far in a few examples.



Some of the examples in this section are available in the GitHub repository [promise-examples](#).

25.5.1 Example: promisifying `fs.readFile()`

The following code is a Promise-based version of the built-in Node.js function `fs.readFile()`.

```
import {readFile} from 'fs';

function readFilePromisified(filename) {
  return new Promise(
    function(resolve, reject) {
      readFile(filename, { encoding: 'utf8' },
        (error, data) => {
          if (error) {
            reject(error);
          } else {
            resolve(data);
          }
        }
      );
    }
  );
}
```

```

    });
  });
}

```

`readFilePromisified()` is used like this:

```

readFilePromisified(process.argv[2])
  .then(text => {
    console.log(text);
  })
  .catch(error => {
    console.log(error);
  });

```

25.5.2 Example: promisifying XMLHttpRequest

The following is a Promise-based function that performs an HTTP GET via the event-based [XMLHttpRequest](#) API:

```

function httpGet(url) {
  return new Promise(
    function (resolve, reject) {
      const request = new XMLHttpRequest();
      request.onload = function () {
        if (this.status === 200) {
          // Success
          resolve(this.response);
        } else {
          // Something went wrong (404 etc.)
          reject(new Error(this.statusText));
        }
      };
      request.onerror = function () {
        reject(new Error(
          'XMLHttpRequest Error: ' + this.statusText));
      };
      request.open('GET', url);
      request.send();
    });
}

```

This is how you use `httpGet()`:

```

httpGet('http://example.com/file.txt')
  .then(
    function (value) {
      console.log('Contents: ' + value);
    },
    function (reason) {
      console.error('Something went wrong', reason);
    });

```

25.5.3 Example: delaying an activity

Let's implement `setTimeout()` as the Promise-based function `delay()` (similar to [Q.delay\(\)](#)).

```

function delay(ms) {
  return new Promise(function (resolve, reject) {
    setTimeout(resolve, ms); // (A)
  });
}

// Using delay():
delay(5000).then(function () { // (B)
  console.log('5 seconds have passed!');
});

```

Note that in line A, we are calling `resolve` with zero parameters, which is the same as calling `resolve(undefined)`. We don't need the fulfillment value in line B, either and simply ignore it. Just being notified is enough here.

25.5.4 Example: timing out a Promise

```

function timeout(ms, promise) {
  return new Promise(function (resolve, reject) {
    promise.then(resolve);
    setTimeout(function () {
      reject(new Error('Timeout after '+ms+' ms')); // (A)
    }, ms);
  });
}

```

Note that the rejection after the timeout (in line A) does not cancel the request, but it does prevent the Promise being fulfilled with its result.

Using `timeout()` looks like this:

```

timeout(5000, httpGet('http://example.com/file.txt'))
  .then(function (value) {
    console.log('Contents: ' + value);
  })
  .catch(function (reason) {
    console.error('Error or timeout', reason);
  });

```

25.6 Other ways of creating Promises

Now we are ready to dig deeper into the features of Promises. Let's first explore two more ways of creating Promises.

25.6.1 `Promise.resolve()`

`Promise.resolve(x)` works as follows:

- For most values `x`, it returns a Promise that is fulfilled with `x`:

```
Promise.resolve('abc')
  .then(x => console.log(x)); // abc
```

- If `x` is a Promise whose constructor is the receiver (`Promise` if you call `Promise.resolve()`) then `x` is returned unchanged:

```
const p = new Promise(() => null);
console.log(Promise.resolve(p) === p); // true
```

- If `x` is a thenable, it is converted to a Promise: the settlement of the thenable will also become the settlement of the Promise. The following code demonstrates that `fulfilledThenable` behaves roughly like a Promise that was fulfilled with the string `'hello'`. After converting it to the Promise `promise`, method `then()` works as expected (last line).

```
const fulfilledThenable = {
  then(reaction) {
    reaction('hello');
  }
};
const promise = Promise.resolve(fulfilledThenable);
console.log(promise instanceof Promise); // true
promise.then(x => console.log(x)); // hello
```

That means that you can use `Promise.resolve()` to convert any value (Promise, thenable or other) to a Promise. In fact, it is used by `Promise.all()` and `Promise.race()` to convert Arrays of arbitrary values to Arrays of Promises.

25.6.2 `Promise.reject()`

`Promise.reject(err)` returns a Promise that is rejected with `err`:

```
const myError = new Error('Problem!');
Promise.reject(myError)
  .catch(err => console.log(err === myError)); // true
```

25.7 Chaining Promises

In this section, we take a closer look at how Promises can be chained. The result of the method call:

```
P.then(onFulfilled, onRejected)
```

is a new Promise `Q`. That means that you can keep the Promise-based control flow going by invoking `then()` on `Q`:

- `Q` is resolved with what is returned by either `onFulfilled` or `onRejected`.
- `Q` is rejected if either `onFulfilled` or `onRejected` throw an exception.

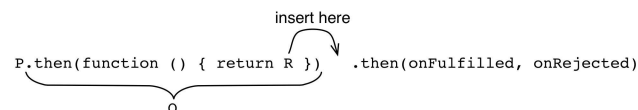
25.7.1 Resolving `Q` with a normal value

If you resolve the Promise `Q` returned by `then()` with a normal value, you can pick up that value via a subsequent `then()`:

```
asyncFunc()
  .then(function (value1) {
    return 123;
  })
  .then(function (value2) {
    console.log(value2); // 123
  });
```

25.7.2 Resolving `Q` with a thenable

You can also resolve the Promise `Q` returned by `then()` with a *thenable* `R`. A thenable is any object that has a method `then()` that works like `Promise.prototype.then()`. Thus, Promises are thenables. Resolving with `R` (e.g. by returning it from `onFulfilled`) means that it is inserted “after” `Q`: `R`'s settlement is forwarded to `Q`'s `onFulfilled` and `onRejected` callbacks. In a way, `Q` becomes `R`.



The main use for this mechanism is to flatten nested `then()` calls, like in the following example:

```
asyncFunc1()
  .then(function (value1) {
```

```

    asyncFunc2 ()
    .then(function (value2) {
        ...
    });
})

```

The flat version looks like this:

```

asyncFunc1 ()
    .then(function (value1) {
        return asyncFunc2 ();
    })
    .then(function (value2) {
        ...
    })

```

25.7.3 Resolving Q from onRejected

Whatever you return in an error handler becomes a fulfillment value (not rejection value!). That allows you to specify default values that are used in case of failure:

```

retrieveFileName ()
    .catch(function () {
        // Something went wrong, use a default value
        return 'Untitled.txt';
    })
    .then(function (fileName) {
        ...
    });

```

25.7.4 Rejecting Q by throwing an exception

Exceptions that are thrown in the callbacks of `then()` and `catch()` are passed on to the next error handler, as rejections:

```

asyncFunc ()
    .then(function (value) {
        throw new Error();
    })
    .catch(function (reason) {
        // Handle error here
    });

```

25.7.5 Chaining and errors

There can be one or more `then()` method calls that don't have error handlers. Then the error is passed on until there is an error handler.

```

asyncFunc1 ()
    .then(asyncFunc2)
    .then(asyncFunc3)
    .catch(function (reason) {
        // Something went wrong above
    });

```

25.8 Common Promise chaining mistakes

25.8.1 Mistake: losing the tail of a Promise chain

In the following code, a chain of two Promises is built, but only the first part of it is returned. As a consequence, the tail of the chain is lost.

```

// Don't do this
function foo () {
    const promise = asyncFunc();
    promise.then(result => {
        ...
    });

    return promise;
}

```

This can be fixed by returning the tail of the chain:

```

function foo () {
    const promise = asyncFunc();
    return promise.then(result => {
        ...
    });
}

```

If you don't need the variable `promise`, you can simplify this code further:

```

function foo () {
    return asyncFunc()
        .then(result => {
            ...
        });
}

```

25.8.2 Mistake: nesting Promises

In the following code, the invocation of `asyncFunc2()` is nested:

```

// Don't do this
asyncFunc1 ()

```

```

.then(result1 => {
  asyncFunc2()
  .then(result2 => {
    ...
  });
});

```

The fix is to un-nest this code by returning the second Promise from the first `then()` and handling it via a second, chained, `then()`:

```

asyncFunc1()
.then(result1 => {
  return asyncFunc2();
})
.then(result2 => {
  ...
});

```

25.8.3 Mistake: creating Promises instead of chaining

In the following code, method `insertInto()` creates a new Promise for its result (line A):

```

// Don't do this
class Model {
  insertInto(db) {
    return new Promise((resolve, reject) => { // (A)
      db.insert(this.fields) // (B)
      .then(resultCode => {
        this.notifyObservers({event: 'created', model: this});
        resolve(resultCode); // (C)
      }).catch(err => {
        reject(err); // (D)
      })
    });
  }
  ...
}

```

If you look closely, you can see that the result Promise is mainly used to forward the fulfillment (line C) and the rejection (line D) of the asynchronous method call `db.insert()` (line B).

The fix is to not create a Promise, by relying on `then()` and chaining:

```

class Model {
  insertInto(db) {
    return db.insert(this.fields) // (A)
    .then(resultCode => {
      this.notifyObservers({event: 'created', model: this});
      return resultCode; // (B)
    });
  }
  ...
}

```

Explanations:

- We return `resultCode` (line B) and let `then()` create the Promise for us.
- We return the Promise chain (line A) and `then()` will pass on any rejection produced by `db.insert()`.

25.8.4 Mistake: using `then()` for error handling

In principle, `catch(cb)` is an abbreviation for `then(null, cb)`. But using both parameters of `then()` at the same time can cause problems:

```

// Don't do this
asyncFunc1()
.then(
  value => { // (A)
    doSomething(); // (B)
    return asyncFunc2(); // (C)
  },
  error => { // (D)
    ...
  });

```

The rejection callback (line D) receives all rejections of `asyncFunc1()`, but it does not receive rejections created by the fulfillment callback (line A). For example, the synchronous function call in line B may throw an exception or the asynchronous function call in line C may produce a rejection.

Therefore, it is better to move the rejection callback to a chained `catch()`:

```

asyncFunc1()
.then(value => {
  doSomething();
  return asyncFunc2();
})
.catch(error => {
  ...
});

```

25.9 Tips for error handling

25.9.1 Operational errors vs. programmer errors

In programs, there are two kinds of errors:

- *Operational errors* happen when a correct program encounters an exceptional situation that requires deviating from the “normal” algorithm. For example, a storage device may run out of memory while the program is writing data to it. This kind of error is expected.
- *Programmer errors* happen when code does something wrong. For example, a function may require a parameter to be a string, but receives a number. This kind of error is unexpected.

25.9.1.1 Operational errors: don't mix rejections and exceptions

For operational errors, each function should support exactly one way of signaling errors. For Promise-based functions that means not mixing rejections and exceptions, which is the same as saying that they shouldn't throw exceptions.

25.9.1.2 Programmer errors: fail quickly

For programmer errors, it can make sense to fail as quickly as possible, by throwing an exception:

```
function downloadFile(url) {
  if (typeof url !== 'string') {
    throw new Error('Illegal argument: ' + url);
  }
  return new Promise(...).
}
```

If you do this, you must make sure that your asynchronous code can handle exceptions. I find throwing exceptions acceptable for assertions and similar things that could, in theory, be checked statically (e.g. via a linter that analyzes the source code).

25.9.2 Handling exceptions in Promise-based functions

If exceptions are thrown inside the callbacks of `then()` and `catch()` then that's not a problem, because these two methods convert them to rejections.

However, things are different if you start your async function by doing something synchronous:

```
function asyncFunc() {
  doSomethingSync(); // (A)
  return doSomethingAsync()
    .then(result => {
      ...
    });
}
```

If an exception is thrown in line A then the whole function throws an exception. There are two solutions to this problem.

25.9.2.1 Solution 1: returning a rejected Promise

You can catch exceptions and return them as rejected Promises:

```
function asyncFunc() {
  try {
    doSomethingSync();
    return doSomethingAsync()
      .then(result => {
        ...
      });
  } catch (err) {
    return Promise.reject(err);
  }
}
```

25.9.2.2 Solution 2: executing the sync code inside a callback

You can also start a chain of `then()` method calls via `Promise.resolve()` and execute the synchronous code inside a callback:

```
function asyncFunc() {
  return Promise.resolve()
    .then(() => {
      doSomethingSync();
      return doSomethingAsync();
    })
    .then(result => {
      ...
    });
}
```

An alternative is to start the Promise chain via the Promise constructor:

```
function asyncFunc() {
  return new Promise((resolve, reject) => {
    doSomethingSync();
    resolve(doSomethingAsync());
  })
}
```

```

        .then(result => {
            ...
        });
    }
}

```

This approach saves you a tick (the synchronous code is executed right away), but it makes your code less regular.

25.9.3 Further reading

Sources of this section:

- Chaining:
 - [“Promise Anti-patterns”](#) on Tao of Code.
- Error handling:
 - [“Error Handling in Node.js”](#) by Joyent
 - [A post by user Mörré Noseshine](#) in the “Exploring ES6” Google Group
 - Feedback to [a tweet](#) asking whether it is OK to throw exceptions from Promise-based functions.

25.10 Composing Promises

Composing means creating new things out of existing pieces. We have already encountered sequential composition of Promises: Given two Promises P and Q, the following code produces a new Promise that executes Q after P is fulfilled.

```
P.then(() => Q)
```

Note that this is similar to the semicolon for synchronous code: Sequential composition of the synchronous operations `f()` and `g()` looks as follows.

```
f(); g()
```

This section describes additional ways of composing Promises.

25.10.1 Manually forking and joining computations

Let’s assume you want to perform two asynchronous computations, `asyncFunc1()` and `asyncFunc2()` in parallel:

```

// Don't do this
asyncFunc1()
  .then(result1 => {
    handleSuccess({result1});
  });
.catch(handleError);

asyncFunc2()
  .then(result2 => {
    handleSuccess({result2});
  })
  .catch(handleError);

const results = {};
function handleSuccess(props) {
  Object.assign(results, props);
  if (Object.keys(results).length === 2) {
    const {result1, result2} = results;
    ...
  }
}

let errorCounter = 0;
function handleError(err) {
  errorCounter++;
  if (errorCounter === 1) {
    // One error means that everything failed,
    // only react to first error
    ...
  }
}

```

The two function calls `asyncFunc1()` and `asyncFunc2()` are made without `then()` chaining. As a consequence, they are both executed immediately and more or less in parallel. Execution is now forked; each function call spawned a separate “thread”. Once both threads are finished (with a result or an error), execution is joined into a single thread in either `handleSuccess()` or `handleError()`.

The problem with this approach is that it involves too much manual and error-prone work. The fix is to not do this yourself, by relying on the built-in method `Promise.all()`.

25.10.2 Forking and joining computations via `Promise.all()`

`Promise.all(iterable)` takes an iterable over Promises (thenables and other values are converted to Promises via `Promise.resolve()`). Once all of them are fulfilled, it fulfills with an Array of their values. If `iterable` is empty, the Promise returned by `all()` is fulfilled immediately.

```

Promise.all([
  asyncFunc1(),
  asyncFunc2(),
])
  .then(([result1, result2]) => {

```

```

    ...
  })
  .catch(err => {
    // Receives first rejection among the Promises
    ...
  });

```

25.10.3 `map()` via `Promise.all()`

One nice thing about Promises is that many synchronous tools still work, because Promise-based functions return results. For example, you can use the Array method `map()`:

```

const fileUrls = [
  'http://example.com/file1.txt',
  'http://example.com/file2.txt',
];
const promisedTexts = fileUrls.map(httpGet);

```

`promisedTexts` is an Array of Promises. We can use `Promise.all()`, which we have already encountered in the previous section, to convert that Array to a Promise that fulfills with an Array of results.

```

Promise.all(promisedTexts)
  .then(texts => {
    for (const text of texts) {
      console.log(text);
    }
  })
  .catch(reason => {
    // Receives first rejection among the Promises
  });

```

25.10.4 Timing out via `Promise.race()`

`Promise.race(iterable)` takes an iterable over Promises (thenables and other values are converted to Promises via `Promise.resolve()`) and returns a Promise `P`. The first of the input Promises that is settled passes its settlement on to the output Promise. If `iterable` is empty then the Promise returned by `race()` is never settled.

As an example, let's use `Promise.race()` to implement a timeout:

```

Promise.race([
  httpGet('http://example.com/file.txt'),
  delay(5000).then(function () {
    throw new Error('Timed out')
  });
])
  .then(function (text) { ... })
  .catch(function (reason) { ... });

```

25.11 Two useful additional Promise methods

This section describes two useful methods for Promises that many Promise libraries provide. They are only shown to further demonstrate Promises, you should not add them to `Promise.prototype` (this kind of patching should only be done by polyfills).

25.11.1 `done()`

When you chain several Promise method calls, you risk silently discarding errors. For example:

```

function doSomething() {
  asyncFunc()
    .then(f1)
    .catch(r1)
    .then(f2); // (A)
}

```

If `then()` in line A produces a rejection, it will never be handled anywhere. The Promise library Q provides a method `done()`, to be used as the last element in a chain of method calls. It either replaces the last `then()` (and has one to two arguments):

```

function doSomething() {
  asyncFunc()
    .then(f1)
    .catch(r1)
    .done(f2);
}

```

Or it is inserted after the last `then()` (and has zero arguments):

```

function doSomething() {
  asyncFunc()
    .then(f1)
    .catch(r1)
    .then(f2)
    .done();
}

```

Quoting the [Q documentation](#):

The Golden Rule of `done` vs. `then` usage is: either return your promise to someone else, or if the chain ends with you, call `done` to terminate it. Terminating with `catch`

is not sufficient because the catch handler may itself throw an error.

This is how you would implement `done()` in ECMAScript 6:

```
Promise.prototype.done = function (onFulfilled, onRejected) {
  this.then(onFulfilled, onRejected)
    .catch(function (reason) {
      // Throw an exception globally
      setTimeout(() => { throw reason }, 0);
    });
};
```

While `done`'s functionality is clearly useful, it has not been added to ECMAScript 6. The idea was to first explore how much engines can detect automatically. Depending on how well that works, it may be necessary to introduce `done()`.

25.11.2 `finally()`

Sometimes you want to perform an action independently of whether an error happened or not. For example, to clean up after you are done with a resource. That's what the Promise method `finally()` is for, which works much like the `finally` clause in exception handling. Its callback receives no arguments, but is notified of either a resolution or a rejection.

```
createResource(...)
  .then(function (value1) {
    // Use resource
  })
  .then(function (value2) {
    // Use resource
  })
  .finally(function () {
    // Clean up
  });
```

This is how Domenic Denicola [proposes](#) to implement `finally()`:

```
Promise.prototype.finally = function (callback) {
  const P = this.constructor;
  // We don't invoke the callback in here,
  // because we want then() to handle its exceptions
  return this.then(
    // Callback fulfills => continue with receiver's fulfillment or rejection
    // Callback rejects => pass on that rejection (then() has no 2nd parameter!)
    value => P.resolve(callback()).then(() => value),
    reason => P.resolve(callback()).then(() => { throw reason })
  );
};
```

The callback determines how the settlement of the receiver (`this`) is handled:

- If the callback throws an exception or returns a rejected Promise then that becomes/contributes the rejection value.
- Otherwise, the settlement (fulfillment or rejection) of the receiver becomes the settlement of the Promise returned by `finally()`. In a way, we take `finally()` out of the chain of methods.

Example 1 (by [Jake Archibald](#)): using `finally()` to hide a spinner. Simplified version:

```
showSpinner();
fetchGalleryData()
  .then(data => updateGallery(data))
  .catch(showNoDataError)
  .finally(hideSpinner);
```

Example 2 (by [Kris Kowal](#)): using `finally()` to tear down a test.

```
const HTTP = require("q-io/http");
const server = HTTP.Server(app);
return server.listen(0)
  .then(function () {
    // run test
  })
  .finally(server.stop);
```

25.12 Node.js: using callback-based sync functions with Promises

The Promise library Q has [tool functions](#) for interfacing with Node.js-style (`err, result`) callback APIs. For example, `denodeify` converts a callback-based function to a Promise-based one:

```
const readFile = Q.denodeify(FS.readFile);

readFile('foo.txt', 'utf-8')
  .then(function (text) {
    ...
  });
```

`denodify` is a micro-library that only provides the functionality of `Q.denodeify()` and complies with the ECMAScript 6 Promise API.

25.13 ES6-compatible Promise libraries

There are many Promise libraries out there. The following ones conform to the ECMAScript 6 API, which means that you can use them now and easily migrate to native ES6 later.

Minimal polyfills:

- “[ES6-Promises](#)” by Jake Archibald extracts just the ES6 API out of RSVP.js.
- “[Native Promise Only \(NPO\)](#)” by Kyle Simpson is “a polyfill for native ES6 promises, as close as possible (no extensions) to the strict spec definitions”.
- “[Lie](#)” by Calvin Metcalf is “a small, performant, promise library implementing the Promises/A+ spec”.

Larger Promise libraries:

- “[RSVP.js](#)” by Stefan Penner is a superset of the ES6 Promise API.
- “[Bluebird](#)” by Petka Antonov is a popular Promises library that passes the ES2015 tests (Test262) and is thus an alternative to ES6 Promises.
- [Q.Promise](#) by Kris Kowal implements the ES6 API.

ES6 standard library polyfills:

- “[ES6 Shim](#)” by Paul Millr includes `Promise`.
- “[core-js](#)” by Denis Pushkarev, the ES6+ polyfill used by Babel, includes `Promise`.

25.14 Promises and generators

25.15 Next step: using Promises via generators

Implementing asynchronous functions via Promises is more convenient than via events or callbacks, but it’s still not ideal:

- Asynchronous code and synchronous code work completely differently. As a consequence, mixing those execution styles and switching between them for a function or method is cumbersome.
- Conceptually, invoking an asynchronous function is a blocking call: The code making the call is suspended during the asynchronous computation and resumed once the result is in. However, the code does not reflect this as much as it could.

The solution is to bring blocking calls to JavaScript. Generators let us do that, via libraries: In the following code, I use [the control flow library co](#) to asynchronously retrieve two JSON files.

```
co(function* () {
  try {
    const [croftStr, bondStr] = yield Promise.all([ // (A)
      getFile('http://localhost:8000/croft.json'),
      getFile('http://localhost:8000/bond.json'),
    ]);
    const croftJson = JSON.parse(croftStr);
    const bondJson = JSON.parse(bondStr);

    console.log(croftJson);
    console.log(bondJson);
  } catch (e) {
    console.log('Failure to read: ' + e);
  }
});
```

In line A, execution blocks (waits) via `yield` until the result of `Promise.all()` is ready. That means that the code looks synchronous while performing asynchronous operations.

Details are explained in [the chapter on generators](#).

25.16 Promises in depth: a simple implementation

In this section, we will approach Promises from a different angle: Instead of learning how to use the API, we will look at a simple implementation of it. This different angle helped me greatly with making sense of Promises.

The Promise implementation is called `DemoPromise`. In order to be easier to understand, it doesn’t completely match the API. But it is close enough to still give you much insight into the challenges that actual implementations face.



`DemoPromise` is available on GitHub, in the repository [demo_promise](#).

`DemoPromise` is a class with three prototype methods:

- `DemoPromise.prototype.resolve(value)`
- `DemoPromise.prototype.reject(reason)`
- `DemoPromise.prototype.then(onFulfilled, onRejected)`

That is, `resolve` and `reject` are methods (versus functions handed to a callback parameter of the constructor).

25.16.1 A stand-alone Promise

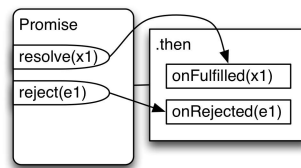
Our first implementation is a stand-alone Promise with minimal functionality:

- You can create a Promise.
- You can resolve or reject a Promise and you can only do it once.
- You can register *reactions* (callbacks) via `then()`. It must work independently of whether the Promise has already been settled or not.
 - This method does not support chaining, yet – it does not return anything.

This is how this first implementation is used:

```
const dp = new DemoPromise();
dp.resolve('abc');
dp.then(function (value) {
  console.log(value); // abc
});
```

The following diagram illustrates how our first `DemoPromise` works:



25.16.1.1 `DemoPromise.prototype.then()`

Let's examine `then()` first. It has to handle two cases:

- If the Promise is still pending, it queues invocations of `onFulfilled` and `onRejected`, to be used when the Promise is settled.
- If the Promise is already fulfilled or rejected, `onFulfilled` or `onRejected` can be invoked right away.

```
then(onFulfilled, onRejected) {
  const self = this;
  const fulfilledTask = function () {
    onFulfilled(self.promiseResult);
  };
  const rejectedTask = function () {
    onRejected(self.promiseResult);
  };
  switch (this.promiseState) {
    case 'pending':
      this.fulfillReactions.push(fulfilledTask);
      this.rejectReactions.push(rejectedTask);
      break;
    case 'fulfilled':
      addToTaskQueue(fulfilledTask);
      break;
    case 'rejected':
      addToTaskQueue(rejectedTask);
      break;
  }
}
```

The previous code snippet uses the following helper function:

```
function addToTaskQueue(task) {
  setTimeout(task, 0);
}
```

25.16.1.2 `DemoPromise.prototype.resolve()`

`resolve()` works as follows: If the Promise is already settled, it does nothing (ensuring that a Promise can only be settled once). Otherwise, the state of the Promise changes to 'fulfilled' and the result is cached in `this.promiseResult`. Next, all fulfillment reactions, that have been enqueued so far, are triggered.

```
resolve(value) {
  if (this.promiseState !== 'pending') return;
  this.promiseState = 'fulfilled';
  this.promiseResult = value;
  this._clearAndEnqueueReactions(this.fulfillReactions);
  return this; // enable chaining
}

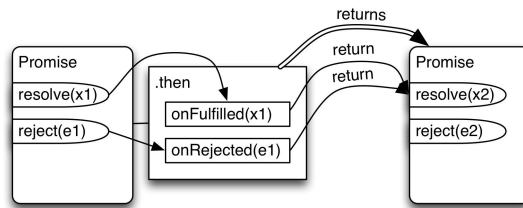
_clearAndEnqueueReactions(reactions) {
  this.fulfillReactions = undefined;
  this.rejectReactions = undefined;
  reactions.map(addToTaskQueue);
}
```

`reject()` is similar to `resolve()`.

25.16.2 Chaining

The next feature we implement is chaining:

- `then()` returns a Promise that is resolved with what either `onFulfilled` or `onRejected` return.
- If `onFulfilled` or `onRejected` are missing, whatever they would have received is passed on to the Promise returned by `then()`.



Obviously, only `then()` changes:

```

then(onFulfilled, onRejected) {
  const returnValue = new Promise(); // (A)
  const self = this;

  let fulfilledTask;
  if (typeof onFulfilled === 'function') {
    fulfilledTask = function () {
      const r = onFulfilled(self.promiseResult);
      returnValue.resolve(r); // (B)
    };
  } else {
    fulfilledTask = function () {
      returnValue.resolve(self.promiseResult); // (C)
    };
  }

  let rejectedTask;
  if (typeof onRejected === 'function') {
    rejectedTask = function () {
      const r = onRejected(self.promiseResult);
      returnValue.resolve(r); // (D)
    };
  } else {
    rejectedTask = function () {
      // 'onRejected' has not been provided
      // => we must pass on the rejection
      returnValue.reject(self.promiseResult); // (E)
    };
  }
  ...
  return returnValue; // (F)
}
  
```

`then()` creates and returns a new Promise (lines A and F). Additionally, `fulfilledTask` and `rejectedTask` are set up differently: After a settlement...

- The result of `onFulfilled` is used to resolve `returnValue` (line B).
 - If `onFulfilled` is missing, we use the fulfillment value to resolve `returnValue` (line C).
- The result of `onRejected` is used to resolve (not reject!) `returnValue` (line D).
 - If `onRejected` is missing, we use pass on the rejection value to `returnValue` (line E).

25.16.3 Flattening

Flattening is mostly about making chaining more convenient: Normally, returning a value from a reaction passes it on to the next `then()`. If we return a Promise, it would be nice if it could be “unwrapped” for us, like in the following example:

```

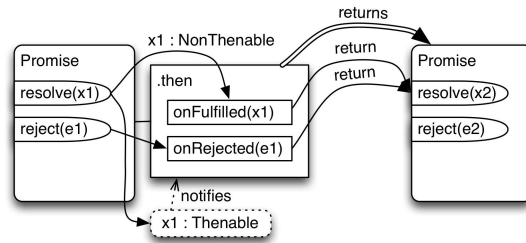
asyncFunc1()
  .then(function (value1) {
    return asyncFunc2(); // (A)
  })
  .then(function (value2) {
    // value2 is fulfillment value of asyncFunc2() Promise
    console.log(value2);
  });
  
```

We returned a Promise in line A and didn't have to nest a call to `then()` inside the current method, we could invoke `then()` on the method's result. Thus: no nested `then()`, everything remains flat.

We implement this by letting the `resolve()` method do the flattening:

- Resolving a Promise P with a Promise Q means that Q's settlement is forwarded to P's reactions.
- P becomes “locked in” on Q: it can't be resolved (incl. rejected), anymore. And its state and result are always the same as Q's.

We can make flattening more generic if we allow Q to be a thenable (instead of only a Promise).



To implement locking-in, we introduce a new boolean flag `this.alreadyResolved`. Once it is true, `this` is locked and can't be resolved anymore. Note that `this` may still be pending, because its state is now the same as the Promise it is locked in on.

```
resolve(value) {
  if (this.alreadyResolved) return;
  this.alreadyResolved = true;
  this._doResolve(value);
  return this; // enable chaining
}
```

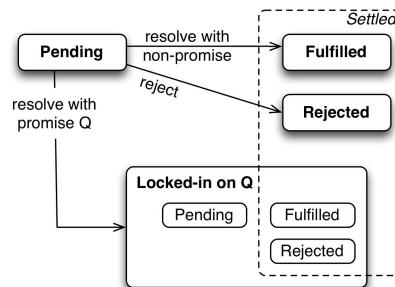
The actual resolution now happens in the private method `_doResolve()`:

```
_doResolve(value) {
  const self = this;
  // Is `value` a thenable?
  if (typeof value === 'object' && value !== null && 'then' in value) {
    // Forward fulfillments and rejections from `value` to `this`.
    // Added as a task (vs. done immediately) to preserve async semantics.
    addTaskQueue(function () { // (A)
      value.then(
        function onFulfilled(result) {
          self._doResolve(result);
        },
        function onRejected(error) {
          self._doReject(error);
        }
      );
    });
  } else {
    this.promiseState = 'fulfilled';
    this.promiseResult = value;
    this._clearAndEnqueueReactions(this.fulfillReactions);
  }
}
```

The flattening is performed in line A: If `value` is fulfilled, we want `self` to be fulfilled and if `value` is rejected, we want `self` to be rejected. The forwarding happens via the private methods `_doResolve` and `_doReject`, to get around the protection via `alreadyResolved`.

25.16.4 Promise states in more detail

With chaining, the states of Promises become more complex (as covered by [Sect. 25.4](#) of the ECMAScript 6 specification):



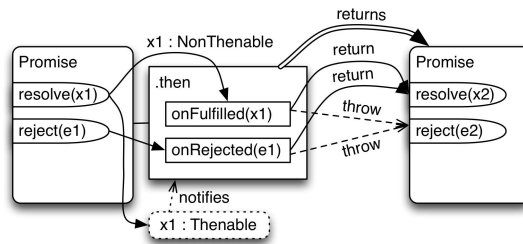
If you are only *using* Promises, you can normally adopt a simplified worldview and ignore locking-in. The most important state-related concept remains “settledness”: a Promise is settled if it is either fulfilled or rejected. After a Promise is settled, it doesn't change, anymore (state and fulfillment or rejection value).

If you want to *implement* Promises then “resolving” matters, too and is now harder to understand:

- Intuitively, “resolved” means “can't be (directly) resolved anymore”. A Promise is resolved if it is either settled or locked in. Quoting the spec: “An unresolved Promise is always in the pending state. A resolved Promise may be pending, fulfilled or rejected.”
- Resolving does not necessarily lead to settling: you can resolve a Promise with another one that is always pending.
- Resolving now includes rejecting (i.e., it is more general): you can reject a Promise by resolving it with a rejected Promise.

25.16.5 Exceptions

As our final feature, we'd like our Promises to handle exceptions in user code as rejections. For now, "user code" means the two callback parameters of `then()`.

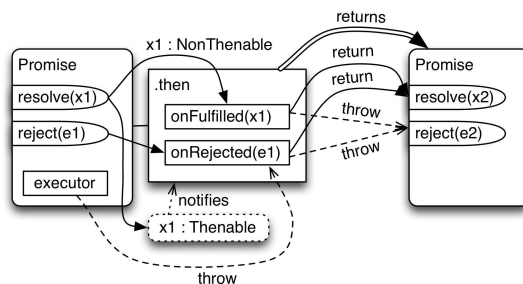


The following excerpt shows how we turn exceptions inside `onFulfilled` into rejections – by wrapping a `try-catch` around its invocation in line A.

```
then(onFulfilled, onRejected) {
  ...
  let fulfilledTask;
  if (typeof onFulfilled === 'function') {
    fulfilledTask = function () {
      try {
        const r = onFulfilled(self.promiseResult); // (A)
        returnValue.resolve(r);
      } catch (e) {
        returnValue.reject(e);
      }
    };
  } else {
    fulfilledTask = function () {
      returnValue.resolve(self.promiseResult);
    };
  }
  ...
}
```

25.16.6 Revealing constructor pattern

If we wanted to turn `DemoPromise` into an actual Promise implementation, we'd still need to implement [the revealing constructor pattern \[5\]](#): ES6 Promises are not resolved and rejected via methods, but via functions that are handed to the *executor*, the callback parameter of the constructor.



If the executor throws an exception then "its" Promise must be rejected.

25.17 Advantages and limitations of Promises

25.17.1 Advantages of Promises

25.17.1.1 Unifying asynchronous APIs

One important advantage of Promises is that they will increasingly be used by asynchronous browser APIs and unify currently diverse and incompatible patterns and conventions. Let's look at two upcoming Promise-based APIs.

The [fetch API](#) is a Promise-based alternative to `XMLHttpRequest`:

```
fetch(url)
  .then(request => request.text())
  .then(str => ...)
```

`fetch()` returns a Promise for the actual request, `text()` returns a Promise for the content as a string.

The [ECMAScript 6 API for programmatically importing modules](#) is based on Promises, too:

```
System.import('some_module.js')
  .then(some_module => {
    ...
  })
```

25.17.1.2 Promises versus events

Compared to events, Promises are better for handling one-off results. It doesn't matter whether you register for a result before or after it has been computed, you will get it. This advantage of Promises is fundamental in nature. On the flip side, you can't use them for handling recurring events. Chaining is another advantage of Promises, but one that could be added to event handling.

25.17.1.3 Promises versus callbacks

Compared to callbacks, Promises have cleaner function (or method) signatures. With callbacks, parameters are used for input and output:

```
fs.readFile(name, opts?, (err, string | Buffer) => void)
```

With Promises, all parameters are used for input:

```
readFilePromisified(name, opts?) : Promise<string | Buffer>
```

Additional Promise advantages include:

- Unified handling of both asynchronous errors and normal exceptions.
- Easier composition, because you can reuse synchronous tools such as `Array.prototype.map()`.
- Chaining of `then()` and `catch()`.
- Guarding against notifying callbacks more than once. Some development environments also warn about rejections that are never handled.

25.17.2 Promises are not always the best choice

Promises work well for single asynchronous results. They are not suited for:

- Recurring events: If you are interested in those, take a look at [reactive programming](#), which add a clever way of chaining to normal event handling.
- Streams of data: A [standard](#) for supporting those is currently in development.

ECMAScript 6 Promises lack two features that are sometimes useful:

- You can't cancel them.
- You can't query them for how far along they are (e.g. to display a progress bar in a client-side user interface).

The Q Promise library has [support](#) for the latter and there are [plans](#) to add both capabilities to Promises/A+.

25.18 Cheat sheet: the ECMAScript 6 Promise API

This section gives an overview of the ECMAScript 6 Promise API, as described in the [specification](#).

25.18.1 Promise constructor

The constructor for Promises is invoked as follows:

```
const p = new Promise(function (resolve, reject) { ... });
```

The callback of this constructor is called an *executor*. The executor can use its parameters to resolve or reject the new Promise `p`:

- `resolve(x)` resolves `p` with `x`:
 - If `x` is thenable, its settlement is forwarded to `p` (which includes triggering reactions registered via `then()`).
 - Otherwise, `p` is fulfilled with `x`.
- `reject(e)` rejects `p` with the value `e` (often an instance of `Error`).

25.18.2 Static Promise methods

25.18.2.1 Creating Promises

The following two static methods create new instances of their receivers:

- `Promise.resolve(x)`: converts arbitrary values to Promises, with an awareness of Promises.
 - If the constructor of `x` is the receiver, `x` is returned unchanged.
 - Otherwise, return a new instance of the receiver that is fulfilled with `x`.
- `Promise.reject(reason)`: creates a new instance of the receiver that is rejected with the value `reason`.

25.18.2.2 Composing Promises

Intuitively, the static methods `Promise.all()` and `Promise.race()` compose iterables of Promises to a single Promise. That is:

- They take an iterable. The elements of the iterable are converted to Promises via `this.resolve()`.
- They return a new Promise. That Promise is a new instance of the receiver.

The methods are:

- `Promise.all(iterable)`: returns a Promise that...
 - is fulfilled if all elements in `iterable` are fulfilled.
Fulfillment value: Array with fulfillment values.
 - is rejected if any of the elements are rejected.
Rejection value: first rejection value.
- `Promise.race(iterable)`: the first element of `iterable` that is settled is used to settle the returned Promise.

25.18.3 `Promise.prototype` methods

25.18.3.1 `Promise.prototype.then(onFulfilled, onRejected)`

- The callbacks `onFulfilled` and `onRejected` are called *reactions*.
- `onFulfilled` is called immediately if the Promise is already fulfilled or as soon as it becomes fulfilled. Similarly, `onRejected` is informed of rejections.
- `then()` returns a new Promise Q (created via the species of the constructor of the receiver):
 - If either of the reactions returns a value, Q is resolved with it.
 - If either of the reactions throws an exception, Q is rejected with it.
- Omitted reactions:
 - If `onFulfilled` has been omitted, a fulfillment of the receiver is forwarded to the result of `then()`.
 - If `onRejected` has been omitted, a rejection of the receiver is forwarded to the result of `then()`.

Default values for omitted reactions could be implemented like this:

```
function defaultOnFulfilled(x) {
  return x;
}
function defaultOnRejected(e) {
  throw e;
}
```

25.18.3.2 `Promise.prototype.catch(onRejected)`

- `p.catch(onRejected)` is the same as `p.then(null, onRejected)`.

25.19 Further reading

[1] “[Promises/A+](#)”, edited by Brian Cavalier and Domenic Denicola (the de-facto standard for JavaScript Promises)

[2] “[The Revealing Constructor Pattern](#)” by Domenic Denicola (this pattern is used by the `Promise` constructor)

VI Miscellaneous

26. Unicode in ES6

This chapter explains the improved support for Unicode that ECMAScript 6 brings. For a general introduction to Unicode, read Chap. “[Unicode and JavaScript](#)” in “Speaking JavaScript”.

26.1 Unicode is better supported in ES6

There are three areas in which ECMAScript 6 has improved support for Unicode:

- Unicode escapes for code points beyond 16 bits: `\u{...}`
Can be used in identifiers, string literals, template literals and regular expression literals. They are explained in the next section.
- [Strings](#):
 - Iteration honors Unicode code points.
 - Read code point values via `String.prototype.codePointAt()`.
 - Create a string from code point values via `String.fromCodePoint()`.
- [Regular expressions](#):
 - New flag `/u` (plus boolean property `unicode`) improves handling of surrogate pairs.

Additionally, ES6 is based on Unicode version 5.1.0, whereas ES5 is based on Unicode version 3.0.

26.2 Escape sequences in ES6

There are three parameterized escape sequences for representing characters in JavaScript:

- Hex escape (exactly two hexadecimal digits): `\xHH`

```
> '\x7A' === 'z'
true
```

- Unicode escape (exactly four hexadecimal digits): `\uHHHH`

```
> '\u007A' === 'z'
true
```

- Unicode code point escape (1 or more hexadecimal digits): `\u{...}`

```
> '\u{7A}' === 'z'
true
```

Unicode code point escapes are new in ES6. They let you specify code points beyond 16 bits. If you wanted to do that in ECMAScript 5, you had to encode each code point as two UTF-16 code units (a *surrogate pair*). These code units could be expressed via Unicode escapes. For example, the following statement logs a rocket (code point 0x1F680) to most consoles:

```
console.log('\uD83D\uDE80');
```

With a Unicode code point escape you can specify code points greater than 16 bits directly:

```
console.log('\u{1F680}');
```

26.2.1 Where can escape sequences be used?

The escape sequences can be used in the following locations:

| | <code>\uHHHH</code> | <code>\u{...}</code> | <code>\xHH</code> |
|-----------------------------|---------------------|--------------------------------|-------------------|
| Identifiers | ✓ | ✓ | |
| String literals | ✓ | ✓ | ✓ |
| Template literals | ✓ | ✓ | ✓ |
| Regular expression literals | ✓ | Only with flag <code>/u</code> | ✓ |

Identifiers:

- A 4-digit Unicode escape `\uHHHH` becomes a single code point.
- A Unicode code point escape `\u{...}` becomes a single code point.

```
> const hello = 123;
> hell\u{6F}
123
```

String literals:

- Strings are internally stored as UTF-16 code units.
- A hex escape `\xHH` contributes a UTF-16 code unit.
- A 4-digit Unicode escape `\uHHHH` contributes a UTF-16 code unit.
- A Unicode code point escape `\u{...}` contributes the UTF-16 encoding of its code point (one or two UTF-16 code units).

Template literals:

- In template literals, escape sequences are handled like in string literals.
- In tagged templates, how escape sequences are interpreted depends on the tag function. It can choose between two interpretations:
 - Cooked: escape sequences are handled like in string literals.
 - Raw: escape sequences are handled as a sequence of characters.

```
> `hell\u{6F}` // cooked
'hello'
> String.raw`hell\u{6F}` // raw
'hell\u{6F}'
```

Regular expressions:

- Unicode code point escapes are only allowed if the flag `/u` is set, because `\u{3}` is interpreted as three times the character `u`, otherwise:

```
> /^u{3}$/.test('uuu')
true
```

26.2.2 Escape sequences in the ES6 spec

Various information:

- The spec treats source code as a sequence of Unicode code points: “[Source Text](#)”
- Unicode escape sequences in identifiers: “[Names and Keywords](#)”

- Strings are internally stored as sequences of UTF-16 code units: “[String Literals](#)”
- Strings – how various escape sequences are translated to UTF-16 code units: “[Static Semantics: SV](#)”
- Template literals – how various escape sequences are translated to UTF-16 code units: “[Static Semantics: TV and TRV](#)”

26.2.2.1 Regular expressions

The spec distinguishes between BMP patterns (flag `/u` not set) and Unicode patterns (flag `/u` set). Sect. “[Pattern Semantics](#)” explains that they are handled differently and how.

As a reminder, here is how grammar rules are parameterized in the spec:

- If a grammar rule R has the subscript $[U]$ then that means there are two versions of it: R and R_U .
- Parts of the rule can pass on the subscript via $[?U]$.
- If a part of a rule has the prefix $[+U]$ it only exists if the subscript $[U]$ is present.
- If a part of a rule has the prefix $[\sim U]$ it only exists if the subscript $[U]$ is not present.

You can see this parameterization in action in Sect. “[Patterns](#)”, where the subscript $[U]$ creates separate grammars for BMP patterns and Unicode patterns:

- **IdentityEscape**: In BMP patterns, many characters can be prefixed with a backslash and are interpreted as themselves (for example: if `\u` is not followed by four hexadecimal digits, it is interpreted as `u`). In Unicode patterns that only works for the following characters (which frees up `\u` for Unicode code point escapes): `^ $ \ . * + ? () [] { } |`
- **RegExpUnicodeEscapeSequence**: “`\u{ " HexDigits " }`” is only allowed in Unicode patterns. In those patterns, lead and trail surrogates are also grouped to help with UTF-16 decoding.

Sect. “[CharacterEscape](#)” explains how various escape sequences are translated to *characters* (roughly: either code units or code points).



Further reading

“[JavaScript has a Unicode problem](#)” (by Mathias Bynens) explains new Unicode features in ES6.

27. Tail call optimization

ECMAScript 6 offers *tail call optimization*, where you can make some function calls without growing the call stack. This chapter explains how that works and what benefits it brings.

27.1 What is tail call optimization?

Roughly, whenever the last thing a function does is to call another function then the latter does not need to return to its caller. As a consequence, no information needs to be stored on the call stack and the function call is more of a goto (a jump). This kind of call is named *tail call*; not growing the stack is named *tail call optimization* (TCO).

Let’s look at an example to better understand TCO. I’ll first explain how it is executed without TCO and then with TCO.

```
function id(x) {
  return x; // (A)
}
function f(a) {
  const b = a + 1;
  return id(b); // (B)
}
console.log(f(2)); // (C)
```

27.1.1 Normal execution

Let’s assume there is a JavaScript engine that manages function calls by storing local variables and return addresses on a stack. How would such an engine execute the code?

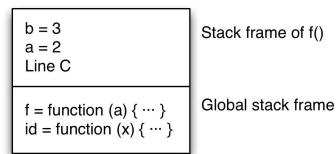
Step 1. Initially, there are only the global variables `id` and `f` on the stack.

```
f = function (a) { ... }
id = function (x) { ... }
```

Global stack frame

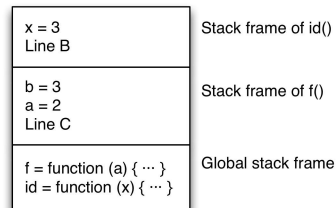
The block of stack entries encodes the state (local variables, including parameters) of the current scope and is called a *stack frame*.

Step 2. In line C, `f()` is called: First, the location to return to is saved on the stack. Then `f`'s parameters are allocated and execution jumps to its body. The stack now looks as follows.



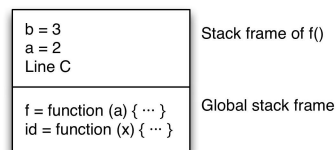
There are now two frames on the stack: One for the global scope (bottom) and one for `f()` (top). `f`'s stack frame includes the return address, line C.

Step 3. `id()` is called in line B. Again, a stack frame is created that contains the return address and `id`'s parameter.

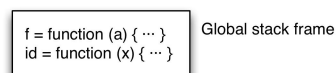


Step 4. In line A, the result `x` is returned. `id`'s stack frame is removed and execution jumps to the return address, line B. (There are several ways in which returning a value could be handled. Two common solutions are to leave the result on a stack or to hand it over in a register. I ignore this part of execution here.)

The stack now looks as follows:



Step 5. In line B, the value that was returned by `id` is returned to `f`'s caller. Again, the topmost stack frame is removed and execution jumps to the return address, line C.



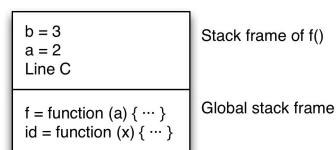
Step 6. Line C receives the value `3` and logs it.

27.1.2 Tail call optimization

```
function id(x) {  
  return x; // (A)  
}  
function f(a) {  
  const b = a + 1;  
  return id(b); // (B)  
}  
console.log(f(2)); // (C)
```

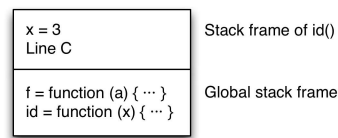
If you look at the previous section then there is one step that is unnecessary – step 5. All that happens in line B is that the value returned by `id()` is passed on to line C. Ideally, `id()` could do that itself and the intermediate step could be skipped.

We can make this happen by implementing the function call in line B differently. Before the call happens, the stack looks as follows.



If we examine the call we see that it is the very last action in `f()`. Once `id()` is done, the only remaining action performed by `f()` is to pass `id`'s result to `f`'s caller. Therefore, `f`'s variables are not needed, anymore and its stack frame can be removed before making

the call. The return address given to `id()` is `f`'s return address, line C. During the execution of `id()`, the stack looks like this:



Then `id()` returns the value 3. You could say that it returns that value for `f()`, because it transports it to `f`'s caller, line C.

Let's review: The function call in line B is a tail call. Such a call can be done with zero stack growth. To find out whether a function call is a tail call, we must check whether it is in a *tail position* (i.e., the last action in a function). How that is done is explained in the next section.

27.2 Checking whether a function call is in a tail position

We have just learned that tail calls are function calls that can be executed more efficiently. But what counts as a tail call?

First, the way in which you call a function does not matter. The following calls can all be optimized if they appear in a tail position:

- Function call: `func(...)`
- Dispatched method call: `obj.method(...)`
- Direct method call via `call()`: `func.call(...)`
- Direct method call via `apply()`: `func.apply(...)`

27.2.1 Tail calls in expressions

Arrow functions can have expressions as bodies. For tail call optimization, we therefore have to figure out where function calls are in tail positions in expressions. Only the following expressions can contain tail calls:

- The conditional operator (`? :`)
- The logical Or operator (`||`)
- The logical And operator (`&&`)
- The comma operator (`,`)

Let's look at an example for each one of them.

27.2.1.1 The conditional operator (`? :`)

```
const a = x => x ? f() : g();
```

Both `f()` and `g()` are in tail position.

27.2.1.2 The logical Or operator (`||`)

```
const a = () => f() || g();
```

`f()` is not in a tail position, but `g()` is in a tail position. To see why, take a look at the following code, which is equivalent to the previous code:

```
const a = () => {
  const fResult = f(); // not a tail call
  if (fResult) {
    return fResult;
  } else {
    return g(); // tail call
  }
};
```

The result of the logical Or operator depends on the result of `f()`, which is why that function call is not in a tail position (the caller does something with it other than returning it). However, `g()` is in a tail position.

27.2.1.3 The logical And operator

```
const a = () => f() && g();
```

`f()` is not in a tail position, but `g()` is in a tail position. To see why, take a look at the following code, which is equivalent to the previous code:

```
const a = () => {
  const fResult = f(); // not a tail call
  if (!fResult) {
    return fResult;
  } else {
    return g(); // tail call
  }
};
```

The result of the logical And operator depends on the result of `f()`, which is why that function call is not in a tail position (the caller does something with it other than returning it). However, `g()` is in a tail position.

27.2.1.4 The comma operator (,)

```
const a = () => (f(), g());
```

`f()` is not in a tail position, but `g()` is in a tail position. To see why, take a look at the following code, which is equivalent to the previous code:

```
const a = () => {
  f();
  return g();
}
```

27.2.2 Tail calls in statements

For statements, the following rules apply.

Only these compound statements can contain tail calls:

- Blocks (as delimited by `{}`), with or without a label
- `if`: in either the “then” clause or the “else” clause.
- `do-while`, `while`, `for`: in their bodies.
- `switch`: in its body.
- `try-catch`: only in the `catch` clause. The `try` clause has the `catch` clause as a context that can’t be optimized away.
- `try-finally`, `try-catch-finally`: only in the `finally` clause, which is a context of the other clauses that can’t be optimized away.

Of all the atomic (non-compound) statements, only `return` can contain a tail call. All other statements have context that can’t be optimized away. The following statement contains a tail call if `expr` contains a tail call.

```
return «expr»;
```

27.2.3 Tail call optimization can only be made in strict mode

In non-strict mode, most engines have the following two properties that allow you to examine the call stack:

- `func.arguments`: contains the arguments of the most recent invocation of `func`.
- `func.caller`: refers to the function that most recently called `func`.

With tail call optimization, these properties don’t work, because the information that they rely on may have been removed. Therefore, strict mode forbids these properties (as described in the language specification) and tail call optimization only works in strict mode.

27.2.4 Pitfall: solo function calls are never in tail position

The function call `bar()` in the following code is not in tail position:

```
function foo() {
  bar(); // this is not a tail call in JS
}
```

The reason is that the last action of `foo()` is not the function call `bar()`, it is (implicitly) returning `undefined`. In other words, `foo()` behaves like this:

```
function foo() {
  bar();
  return undefined;
}
```

Callers can rely on `foo()` always returning `undefined`. If `bar()` were to return a result for `foo()`, due to tail call optimization, then that would change `foo`’s behavior.

Therefore, if we want `bar()` to be a tail call, we have to change `foo()` as follows.

```
function foo() {
  return bar(); // tail call
}
```

27.3 Tail-recursive functions

A function is *tail-recursive* if the main recursive calls it makes are in tail positions.

For example, the following function is not tail recursive, because the main recursive call in line A is not in a tail position:

```
function factorial(x) {
  if (x <= 0) {
    return 1;
  } else {
    return x * factorial(x-1); // (A)
  }
}
```

`factorial()` can be implemented via a tail-recursive helper function `facRec()`. The main recursive call in line A is in a tail position.

```
function factorial(n) {
  return facRec(n, 1);
}
function facRec(x, acc) {
  if (x <= 1) {
    return acc;
  } else {
    return facRec(x-1, x*acc); // (A)
  }
}
```

That is, some non-tail-recursive functions can be transformed into tail-recursive functions.

27.3.1 Tail-recursive loops

Tail call optimization makes it possible to implement loops via recursion without growing the stack. The following are two examples.

27.3.1.1 `forEach()`

```
function forEach(arr, callback, start = 0) {
  if (0 <= start && start < arr.length) {
    callback(arr[start], start, arr);
    return forEach(arr, callback, start+1); // tail call
  }
}
forEach(['a', 'b'], (elem, i) => console.log(`${i}. ${elem}`));

// Output:
// 0. a
// 1. b
```

27.3.1.2 `findIndex()`

```
function findIndex(arr, predicate, start = 0) {
  if (0 <= start && start < arr.length) {
    if (predicate(arr[start])) {
      return start;
    }
    return findIndex(arr, predicate, start+1); // tail call
  }
}
findIndex(['a', 'b'], x => x === 'b'); // 1
```

28. Metaprogramming with proxies

28.1 Overview

Proxies enable you to intercept and customize operations performed on objects (such as getting properties). They are a *metaprogramming* feature.

In the following example, `proxy` is the object whose operations we are intercepting and `handler` is the object that handles the interceptions. In this case, we are only intercepting a single operation, `get` (getting properties).

```
const target = {};
const handler = {
  get(target, propKey, receiver) {
    console.log('get ' + propKey);
    return 123;
  }
};
const proxy = new Proxy(target, handler);
```

When we get the property `proxy.foo`, the handler intercepts that operation:

```
> proxy.foo
get foo
123
```

Consult [the reference for the complete API](#) for a list of operations that can be intercepted.

28.2 Programming versus metaprogramming

Before we can get into what proxies are and why they are useful, we first need to understand what *metaprogramming* is.

In programming, there are levels:

- At the *base level* (also called: *application level*), code processes user input.
- At the *meta level*, code processes base level code.

Base and meta level can be different languages. In the following meta program, the metaprogramming language is JavaScript and the base programming language is

Java.

```
const str = 'Hello' + '!'.repeat(3);
console.log('System.out.println="'+str+'"');
```

Metaprogramming can take different forms. In the previous example, we have printed Java code to the console. Let's use JavaScript as both metaprogramming language and base programming language. The classic example for this is the `eval()` function, which lets you evaluate/compile JavaScript code on the fly. There are [not that many actual use cases](#) for `eval()`. In the interaction below, we use it to evaluate the expression `5 + 2`.

```
> eval('5 + 2')
7
```

Other JavaScript operations may not look like metaprogramming, but actually are, if you look closer:

```
// Base level
const obj = {
  hello() {
    console.log('Hello!');
  }
};

// Meta level
for (const key of Object.keys(obj)) {
  console.log(key);
}
```

The program is examining its own structure while running. This doesn't look like metaprogramming, because the separation between programming constructs and data structures is fuzzy in JavaScript. All of the `Object.* methods` can be considered metaprogramming functionality.

28.2.1 Kinds of metaprogramming

Reflective metaprogramming means that a program processes itself. [Kiczales et al. \[2\]](#) distinguish three kinds of reflective metaprogramming:

- **Introspection:** you have read-only access to the structure of a program.
- **Self-modification:** you can change that structure.
- **Intercession:** you can redefine the semantics of some language operations.

Let's look at examples.

Example: introspection. `Object.keys()` performs introspection (see previous example).

Example: self-modification. The following function `moveProperty` moves a property from a source to a target. It performs self-modification via the bracket operator for property access, the assignment operator and the `delete` operator. (In production code, you'd probably use [property descriptors](#) for this task.)

```
function moveProperty(source, propertyName, target) {
  target[propertyName] = source[propertyName];
  delete source[propertyName];
}
```

Using `moveProperty()`:

```
> const obj1 = { prop: 'abc' };
> const obj2 = {};
> moveProperty(obj1, 'prop', obj2);

> obj1
{}
> obj2
{ prop: 'abc' }
```

ECMAScript 5 doesn't support intercession; proxies were created to fill that gap.

28.3 A first look at proxies

ECMAScript 6 proxies bring intercession to JavaScript. They work as follows. There are many operations that you can perform on an object `obj`. For example:

- Getting the property `prop` of an object `obj` (`obj.prop`)
- Checking whether an object `obj` has a property `prop` (`'prop' in obj`)

Proxies are special objects that allow you customize some of these operations. A proxy is created with two parameters:

- **handler:** For each operation, there is a corresponding handler method that – if present – performs that operation. Such a method *intercepts* the operation (on its way to the target) and is called a *trap* (a term borrowed from the domain of operating systems).
- **target:** If the handler doesn't intercept an operation then it is performed on the target. That is, it acts as a fallback for the handler. In a way, the proxy wraps the target.

In the following example, the handler intercepts the operations `get` and `has`.

```
const target = {};
const handler = {
  /** Intercepts: getting properties */
  get(target, propKey, receiver) {
    console.log(`GET ${propKey}`);
    return 123;
  },

  /** Intercepts: checking whether properties exist */
  has(target, propKey) {
    console.log(`HAS ${propKey}`);
    return true;
  }
};
const proxy = new Proxy(target, handler);
```

When we get property `foo`, the handler intercepts that operation:

```
> proxy.foo
GET foo
123
```

Similarly, the `in` operator triggers `has`:

```
> 'hello' in proxy
HAS hello
true
```

The handler doesn't implement the trap `set` (setting properties). Therefore, setting `proxy.bar` is forwarded to `target` and leads to `target.bar` being set.

```
> proxy.bar = 'abc';
> target.bar
'abc'
```

28.3.1 Function-specific traps

If the target is a function, two additional operations can be intercepted:

- `apply`: Making a function call, triggered via
 - `proxy(...)`
 - `proxy.call(...)`
 - `proxy.apply(...)`
- `construct`: Making a constructor call, triggered via
 - `new proxy(...)`

The reason for only enabling these traps for function targets is simple: You wouldn't be able to forward the operations `apply` and `construct`, otherwise.

28.3.2 Intercepting method calls

If you want to intercept method calls via a proxy, there is one challenge: you can intercept the operation `get` (getting property values) and you can intercept the operation `apply` (calling a function), but there is no single operation for method calls that you could intercept. That's because method calls are viewed as two separate operations: First a `get` to retrieve a function, then an `apply` to call that function.

Therefore, you must intercept `get` and return a function that intercepts the function call. The following code demonstrates how that is done.

```
function traceMethodCalls(obj) {
  const handler = {
    get(target, propKey, receiver) {
      const origMethod = target[propKey];
      return function (...args) {
        const result = origMethod.apply(this, args);
        console.log(propKey + JSON.stringify(args)
          + ' -> ' + JSON.stringify(result));
        return result;
      };
    }
  };
  return new Proxy(obj, handler);
}
```

I'm not using a Proxy for the latter task, I'm simply wrapping the original method with a function.

Let's use the following object to try out `traceMethodCalls()`:

```
const obj = {
  multiply(x, y) {
    return x * y;
  },
  squared(x) {
    return this.multiply(x, x);
  },
};
```

`tracedObj` is a traced version of `obj`. The first line after each method call is the output of `console.log()`, the second line is the result of the method call.

```

> const tracedObj = traceMethodCalls(obj);
> tracedObj.multiply(2,7)
multiply[2,7] -> 14
14
> tracedObj.squared(9)
multiply[9,9] -> 81
squared[9] -> 81
81

```

The nice thing is that even the call `this.multiply()` that is made inside `obj.squared()` is traced. That's because `this` keeps referring to the proxy.

This is not the most efficient solution. One could, for example, cache methods. Furthermore, Proxies themselves have an impact on performance.

28.3.3 Revocable proxies

ECMAScript 6 lets you create proxies that can be *revoked* (switched off):

```
const {proxy, revoke} = Proxy.revocable(target, handler);
```

On the left hand side of the assignment operator (`=`), we are using destructuring to access the properties `proxy` and `revoke` of the object returned by `Proxy.revocable()`.

After you call the function `revoke` for the first time, any operation you apply to `proxy` causes a `TypeError`. Subsequent calls of `revoke` have no further effect.

```

const target = {}; // Start with an empty object
const handler = {}; // Don't intercept anything
const {proxy, revoke} = Proxy.revocable(target, handler);

```

```

proxy.foo = 123;
console.log(proxy.foo); // 123

```

```
revoke();
```

```
console.log(proxy.foo); // TypeError: Revoked
```

28.3.4 Proxies as prototypes

A proxy `proto` can become the prototype of an object `obj`. Some operations that begin in `obj` may continue in `proto`. One such operation is `get`.

```

const proto = new Proxy({}, {
  get(target, propertyKey, receiver) {
    console.log('GET ' + propertyKey);
    return target[propertyKey];
  }
});

```

```

const obj = Object.create(proto);
obj.bla;

```

```

// Output:
// GET bla

```

The property `bla` can't be found in `obj`, which is why the search continues in `proto` and the trap `get` is triggered there. There are more operations that affect prototypes; they are listed at the end of this chapter.

28.3.5 Forwarding intercepted operations

Operations whose traps the handler doesn't implement are automatically forwarded to the target. Sometimes there is some task you want to perform in addition to forwarding the operation. For example, a handler that intercepts all operations and logs them, but doesn't prevent them from reaching the target:

```

const handler = {
  deleteProperty(target, propKey) {
    console.log('DELETE ' + propKey);
    return delete target[propKey];
  },
  has(target, propKey) {
    console.log('HAS ' + propKey);
    return propKey in target;
  },
  // Other traps: similar
}

```

For each trap, we first log the name of the operation and then forward it by performing it manually. ECMAScript 6 has the module-like object `Reflect` that helps with forwarding: for each trap

```
handler.trap(target, arg_1, ..., arg_n)
```

`Reflect` has a method

```
Reflect.trap(target, arg_1, ..., arg_n)
```

If we use `Reflect`, the previous example looks as follows.

```

const handler = {
  deleteProperty(target, propKey) {
    console.log('DELETE ' + propKey);

```

```

    return Reflect.deleteProperty(target, propKey);
  },
  has(target, propKey) {
    console.log('HAS ' + propKey);
    return Reflect.has(target, propKey);
  },
  // Other traps: similar
}

```

Now what each of the traps does is so similar that we can implement the handler via a proxy:

```

const handler = new Proxy({}, {
  get(target, trapName, receiver) {
    // Return the handler method named trapName
    return function (...args) {
      // Don't log args[0]
      console.log(trapName.toUpperCase() + ' ' + args.slice(1));
      // Forward the operation
      return Reflect[trapName](...args);
    }
  }
});

```

For each trap, the proxy asks for a handler method via the `get` operation and we give it one. That is, all of the handler methods can be implemented via the single meta method `get`. It was one of the goals for the proxy API to make this kind of virtualization simple.

Let's use this proxy-based handler:

```

> const target = {};
> const proxy = new Proxy(target, handler);
> proxy.foo = 123;
SET foo,123,[object Object]
> proxy.foo
GET foo,[object Object]
123

```

The following interaction confirms that the `set` operation was correctly forwarded to the target:

```

> target.foo
123

```

28.4 Use cases for proxies

This section demonstrates what proxies can be used for. That will give you the opportunity to see the API in action.

28.4.1 Tracing property accesses (`get`, `set`)

Let's assume we have a function `tracePropAccess(obj, propKeys)` that logs whenever a property of `obj`, whose key is in the Array `propKeys`, is set or got. In the following code, we apply that function to an instance of the class `Point`:

```

class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  toString() {
    return `Point(${this.x}, ${this.y})`;
  }
}
// Trace accesses to properties `x` and `y`
const p = new Point(5, 7);
p = tracePropAccess(p, ['x', 'y']);

```

Getting and setting properties of the traced object `p` has the following effects:

```

> p.x
GET x
5
> p.x = 21
SET x=21
21

```

Intriguingly, tracing also works whenever `Point` accesses the properties, because `this` now refers to the traced object, not to an instance of `Point`.

```

> p.toString()
GET x
GET y
'Point(21, 7)'

```

In ECMAScript 5, you'd implement `tracePropAccess()` as follows. We replace each property with a getter and a setter that traces accesses. The setters and getters use an extra object, `propData`, to store the data of the properties. Note that we are destructively changing the original implementation, which means that we are metaprogramming.

```

function tracePropAccess(obj, propKeys) {
  // Store the property data here
  const propData = Object.create(null);
  // Replace each property with a getter and a setter
  propKeys.forEach(function (propKey) {
    propData[propKey] = obj[propKey];
    Object.defineProperty(obj, propKey, {

```



```

    get: function () {
      console.log('GET '+propKey);
      return propData[propKey];
    },
    set: function (value) {
      console.log('SET '+propKey+'='+value);
      propData[propKey] = value;
    },
  });
});
return obj;
}

```

In ECMAScript 6, we can use a simpler, proxy-based solution. We intercept property getting and setting and don't have to change the implementation.

```

function tracePropAccess(obj, propKeys) {
  const propKeySet = new Set(propKeys);
  return new Proxy(obj, {
    get(target, propKey, receiver) {
      if (propKeySet.has(propKey)) {
        console.log('GET '+propKey);
      }
      return Reflect.get(target, propKey, receiver);
    },
    set(target, propKey, value, receiver) {
      if (propKeySet.has(propKey)) {
        console.log('SET '+propKey+'='+value);
      }
      return Reflect.set(target, propKey, value, receiver);
    },
  });
}

```

28.4.2 Warning about unknown properties (get, set)

When it comes to accessing properties, JavaScript is very forgiving. For example, if you try to read a property and misspell its name, you don't get an exception, you get the result `undefined`. You can use proxies to get an exception in such a case. This works as follows. We make the proxy a prototype of an object.

If a property isn't found in the object, the `get` trap of the proxy is triggered. If the property doesn't even exist in the prototype chain after the proxy, it really is missing and we throw an exception. Otherwise, we return the value of the inherited property. We do so by forwarding the `get` operation to the target (the prototype of the target is also the prototype of the proxy).

```

const PropertyChecker = new Proxy({}, {
  get(target, propKey, receiver) {
    if (!(propKey in target)) {
      throw new ReferenceError('Unknown property: '+propKey);
    }
    return Reflect.get(target, propKey, receiver);
  }
});

```

Let's use `PropertyChecker` for an object that we create:

```

> const obj = { __proto__: PropertyChecker, foo: 123 };
> obj.foo // own
123
> obj.foo
ReferenceError: Unknown property: fo
> obj.toString() // inherited
'[object Object]'

```

If we turn `PropertyChecker` into a constructor, we can use it for ECMAScript 6 classes via `extends`:

```

function PropertyChecker() { }
PropertyChecker.prototype = new Proxy(...);

class Point extends PropertyChecker {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
}

const p = new Point(5, 7);
console.log(p.x); // 5
console.log(p.z); // ReferenceError

```

If you are worried about accidentally *creating* properties, you have two options: You can either wrap a proxy around objects that traps `set`. Or you can make an object `obj` non-extensible via `Object.preventExtensions(obj)`, which means that JavaScript doesn't let you add new (own) properties to `obj`.

28.4.3 Negative Array indices (get)

Some Array methods let you refer to the last element via `-1`, to the second-to-last element via `-2`, etc. For example:

```

> ['a', 'b', 'c'].slice(-1)
[ 'c' ]

```

Alas, that doesn't work when accessing elements via the bracket operator (`[]`). We can, however, use proxies to add that capability. The following function `createArray()` creates Arrays that support negative indices. It does so by wrapping proxies around Array instances. The proxies intercept the `get` operation that is triggered by the bracket operator.

```
function createArray(...elements) {
  const handler = {
    get(target, propKey, receiver) {
      // Sloppy way of checking for negative indices
      const index = Number(propKey);
      if (index < 0) {
        propKey = String(target.length + index);
      }
      return Reflect.get(target, propKey, receiver);
    }
  };
  // Wrap a proxy around an Array
  const target = [];
  target.push(...elements);
  return new Proxy(target, handler);
}
const arr = createArray('a', 'b', 'c');
console.log(arr[-1]); // c
```

Acknowledgement: The idea for this example comes from a [blog post](#) by hemanth.hm.

28.4.4 Data binding (set)

Data binding is about syncing data between objects. One popular use case are widgets based on the MVC (Model View Controller) pattern: With data binding, the *view* (the widget) stays up-to-date if you change the *model* (the data visualized by the widget).

To implement data binding, you have to observe and react to changes made to an object. In the following code snippet, I sketch how observing changes could work for an Array.

```
function createObservedArray(callback) {
  const array = [];
  return new Proxy(array, {
    set(target, propertyKey, value, receiver) {
      callback(propertyKey, value);
      return Reflect.set(target, propertyKey, value, receiver);
    }
  });
}
const observedArray = createObservedArray(
  (key, value) => console.log(` ${key}=${value}`));
observedArray.push('a');
```

Output:

```
0=a
length=1
```

28.4.5 Accessing a restful web service (method calls)

A proxy can be used to create an object on which arbitrary methods can be invoked. In the following example, the function `createWebService` creates one such object, `service`. Invoking a method on `service` retrieves the contents of the web service resource with the same name. Retrieval is handled via an ECMAScript 6 Promise.

```
const service = createWebService('http://example.com/data');
// Read JSON data in http://example.com/data/employees
service.employees().then(json => {
  const employees = JSON.parse(json);
  ...
});
```

The following code is a quick and dirty implementation of `createWebService` in ECMAScript 5. Because we don't have proxies, we need to know beforehand what methods will be invoked on `service`. The parameter `propKeys` provides us with that information, it holds an Array with method names.

```
function createWebService(baseUrl, propKeys) {
  const service = {};
  propKeys.forEach(function(propKey) {
    service[propKey] = function() {
      return httpGet(baseUrl+'/' + propKey);
    };
  });
  return service;
}
```

The ECMAScript 6 implementation of `createWebService` can use proxies and is simpler:

```
function createWebService(baseUrl) {
  return new Proxy({}, {
    get(target, propKey, receiver) {
      // Return the method to be called
      return () => httpGet(baseUrl+'/' + propKey);
    }
  });
}
```

Both implementations use the following function to make HTTP GET requests (how it works is explained in [the chapter on Promises](#)).

```
function httpGet(url) {
  return new Promise(
    (resolve, reject) => {
      const request = new XMLHttpRequest();
      Object.assign(request, {
        onload() {
          if (this.status === 200) {
            // Success
            resolve(this.response);
          } else {
            // Something went wrong (404 etc.)
            reject(new Error(this.statusText));
          }
        },
        onerror() {
          reject(new Error(
            'XMLHttpRequest Error: '+this.statusText));
        }
      });
      request.open('GET', url);
      request.send();
    }
  );
}
```

28.4.6 Revocable references

Revocable references work as follows: A client is not allowed to access an important resource (an object) directly, only via a reference (an intermediate object, a wrapper around the resource). Normally, every operation applied to the reference is forwarded to the resource. After the client is done, the resource is protected by *revoking* the reference, by switching it off. Henceforth, applying operations to the reference throws exceptions and nothing is forwarded, anymore.

In the following example, we create a revocable reference for a resource. We then read one of the resource's properties via the reference. That works, because the reference grants us access. Next, we revoke the reference. Now the reference doesn't let us read the property, anymore.

```
const resource = { x: 11, y: 8 };
const {reference, revoke} = createRevocableReference(resource);

// Access granted
console.log(reference.x); // 11

revoke();

// Access denied
console.log(reference.x); // TypeError: Revoked
```

Proxies are ideally suited for implementing revocable references, because they can intercept and forward operations. This is a simple proxy-based implementation of `createRevocableReference`:

```
function createRevocableReference(target) {
  let enabled = true;
  return {
    reference: new Proxy(target, {
      get(target, propKey, receiver) {
        if (!enabled) {
          throw new TypeError('Revoked');
        }
        return Reflect.get(target, propKey, receiver);
      },
      has(target, propKey) {
        if (!enabled) {
          throw new TypeError('Revoked');
        }
        return Reflect.has(target, propKey);
      },
      ...
    }),
    revoke() {
      enabled = false;
    },
  };
}
```

The code can be simplified via the proxy-as-handler technique from the previous section. This time, the handler basically is the `Reflect` object. Thus, the `get` trap normally returns the appropriate `Reflect` method. If the reference has been revoked, a `TypeError` is thrown, instead.

```
function createRevocableReference(target) {
  let enabled = true;
  const handler = new Proxy({}, {
    get(dummyTarget, trapName, receiver) {
      if (!enabled) {
        throw new TypeError('Revoked');
      }
      return Reflect[trapName];
    }
  });
  return {
    reference: new Proxy(target, handler),
    revoke() {
      enabled = false;
    },
  };
}
```

```

    };
}

```

However, you don't have to implement revocable references yourself, because ECMAScript 6 lets you create proxies that can be revoked. This time, the revoking happens in the proxy, not in the handler. All the handler has to do is forward every operation to the target. As we have seen that happens automatically if the handler doesn't implement any traps.

```

function createRevocableReference(target) {
  const handler = {}; // forward everything
  const { proxy, revoke } = Proxy.revocable(target, handler);
  return { reference: proxy, revoke };
}

```

28.4.6.1 Membranes

Membranes build on the idea of revocable references: Environments that are designed to run untrusted code, wrap a membrane around that code to isolate it and keep the rest of the system safe. Objects pass the membrane in two directions:

- The code may receive objects (“dry objects”) from the outside.
- Or it may hand objects (“wet objects”) to the outside.

In both cases, revocable references are wrapped around the objects. Objects returned by wrapped functions or methods are also wrapped. Additionally, if a wrapped wet object is passed back into a membrane, it is unwrapped.

Once the untrusted code is done, all of the revocable references are revoked. As a result, none of its code on the outside can be executed anymore and outside objects that it has cease to work, as well. The [Caja Compiler](#) is “a tool for making third party HTML, CSS and JavaScript safe to embed in your website”. It uses membranes to achieve this task.

28.4.7 Implementing the DOM in JavaScript

The browser Document Object Model (DOM) is usually implemented as a mix of JavaScript and C++. Implementing it in pure JavaScript is useful for:

- Emulating a browser environment, e.g. to manipulate HTML in Node.js. [jsdom](#) is one library that does that.
- Speeding the DOM up (switching between JavaScript and C++ costs time).

Alas, the standard DOM can do things that are not easy to replicate in JavaScript. For example, most DOM collections are live views on the current state of the DOM that change dynamically whenever the DOM changes. As a result, pure JavaScript implementations of the DOM are not very efficient. One of the reasons for adding proxies to JavaScript was to help write more efficient DOM implementations.

28.4.8 Other use cases

There are more use cases for proxies. For example:

- Remoting: Local placeholder objects forward method invocations to remote objects. This use case is similar to the web service example.
- Data access objects for databases: Reading and writing to the object reads and writes to the database. This use case is similar to the web service example.
- Profiling: Intercept method invocations to track how much time is spent in each method. This use case is similar to the tracing example.
- Type checking: Nicholas Zakas has used proxies to [type-check objects](#).

28.5 The design of the proxy API

In this section, we go deeper into how proxies work and why they work that way.

28.5.1 Stratification: keeping base level and meta level separate

Firefox has allowed you to do some interceptive metaprogramming for a while: If you define a method whose name is `__noSuchMethod__`, it is notified whenever a method is called that doesn't exist. The following is an example of using `__noSuchMethod__`.

```

const obj = {
  __noSuchMethod__: function (name, args) {
    console.log(name + ': ' + args);
  }
};
// Neither of the following two methods exist,
// but we can make it look like they do
obj.foo(1); // Output: foo: 1
obj.bar(1, 2); // Output: bar: 1,2

```

Thus, `__noSuchMethod__` works similarly to a proxy trap. In contrast to proxies, the trap is an own or inherited method of the object whose operations we want to intercept. The problem with that approach is that base level (normal methods) and meta level (`__noSuchMethod__`) are mixed. Base-level code may accidentally invoke or see a meta level method and there is the possibility of accidentally defining a meta level method.

Even in standard ECMAScript 5, base level and meta level are sometimes mixed. For example, the following metaprogramming mechanisms can fail, because they exist at the base level:

- `obj.hasOwnProperty(propKey)`: This call can fail if a property in the prototype chain overrides the built-in implementation. For example, it fails if `obj` is:

```
{ hasOwnProperty: null }
```

A safe way to call this method is:

```
Object.prototype.hasOwnProperty.call(obj, propKey)

// Abbreviated version:
{}.hasOwnProperty.call(obj, propKey)
```

- `func.call(...), func.apply(...)`: For each of these two methods, problem and solution are the same as with `hasOwnProperty`.
- `obj.__proto__`: In most JavaScript engines, `__proto__` is a special property that lets you get and set the prototype of `obj`. Hence, when you use objects as dictionaries, you must be careful to [avoid `__proto__` as a property key](#).

By now, it should be obvious that making (base level) property keys special is problematic. Therefore, proxies are *stratified* – base level (the proxy object) and meta level (the handler object) are separate.

28.5.2 Virtual objects versus wrappers

Proxies are used in two roles:

- As *wrappers*, they *wrap* their targets, they control access to them. Examples of wrappers are: revocable resources and tracing proxies.
- As *virtual objects*, they are simply objects with special behavior and their targets don't matter. An example is a proxy that forwards method calls to a remote object.

An earlier design of the proxy API conceived proxies as purely virtual objects. However, it turned out that even in that role, a target was useful, to enforce invariants (which is explained later) and as a fallback for traps that the handler doesn't implement.

28.5.3 Transparent virtualization and handler encapsulation

Proxies are shielded in two ways:

- It is impossible to determine whether an object is a proxy or not (*transparent virtualization*).
- You can't access a handler via its proxy (*handler encapsulation*).

Both principles give proxies considerable power for impersonating other objects. One reason for enforcing *invariants* (as explained later) is to keep that power in check.

If you do need a way to tell proxies apart from non-proxies, you have to implement it yourself. The following code is a module `lib.js` that exports two functions: one of them creates proxies, the other one determines whether an object is one of those proxies.

```
// lib.js
const proxies = new WeakSet();

export function createProxy(obj) {
  const handler = {};
  const proxy = new Proxy(obj, handler);
  proxies.add(proxy);
  return proxy;
}

export function isProxy(obj) {
  return proxies.has(obj);
}
```

This module uses the ECMAScript 6 data structure `WeakSet` for keeping track of proxies. `WeakSet` is ideally suited for this purpose, because it doesn't prevent its elements from being garbage-collected.

The next example shows how `lib.js` can be used.

```
// main.js
import { createProxy, isProxy } from './lib.js';

const p = createProxy({});
console.log(isProxy(p)); // true
console.log(isProxy({})); // false
```

28.5.4 The meta object protocol and proxy traps

This section examines how JavaScript is structured internally and how the set of proxy traps was chosen.

In the context of programming languages and API design, a *protocol* is a set of interfaces plus rules for using them. The ECMAScript specification describes how to execute JavaScript code. It includes a [protocol for handling objects](#). This protocol operates at a meta level and is sometimes called the meta object protocol (MOP). The

JavaScript MOP consists of own internal methods that all objects have. “Internal” means that they exist only in the specification (JavaScript engines may or may not have them) and are not accessible from JavaScript. The names of internal methods are written in double square brackets.

The internal method for getting properties is called `[[Get]]`. If we pretend that property names with square brackets are legal, this method would roughly be implemented as follows in JavaScript.

```
// Method definition
[[Get]](propKey, receiver) {
  const desc = this.ownPropertyDescriptor(propKey);
  if (desc === undefined) {
    const parent = this.getPrototypeOf();
    if (parent === null) return undefined;
    return parent.ownPropertyDescriptor(propKey); // (A)
  }
  if ('value' in desc) {
    return desc.value;
  }
  const getter = desc.get;
  if (getter === undefined) return undefined;
  return getter.call(receiver, []);
}
```

The MOP methods called in this code are:

- `ownPropertyDescriptor` (trap `ownPropertyDescriptor`)
- `getPrototypeOf` (trap `getPrototypeOf`)
- `[[Get]]` (trap `get`)
- `[[Call]]` (trap `apply`)

In line (A) you can see why proxies in a prototype chain find out about `get` if a property isn't found in an “earlier” object: If there is no own property whose key is `propKey`, the search continues in the prototype `parent` of `this`.

Fundamental versus derived operations. You can see that `[[Get]]` calls other MOP operations. Operations that do that are called *derived*. Operations that don't depend on other operations are called *fundamental*.

28.5.4.1 The MOP of proxies

The [meta object protocol of proxies](#) is different from that of normal objects. For normal objects, derived operations call other operations. For proxies, each operation (regardless of whether it is fundamental or derived) is either intercepted by a handler method or forwarded to the target.

What operations should be interceptable via proxies? One possibility is to only provide traps for fundamental operations. The alternative is to include some derived operations. The advantage of doing so is that it increases performance and is more convenient. For example, if there weren't a trap for `get`, you'd have to implement its functionality via `ownPropertyDescriptor`. One problem with derived traps is that they can lead to proxies behaving inconsistently. For example, `get` may return a value that is different from the value in the descriptor returned by `ownPropertyDescriptor`.

28.5.4.2 Selective intercession: what operations should be interceptable?

Intercession by proxies is *selective*: you can't intercept every language operation. Why were some operations excluded? Let's look at two reasons.

First, stable operations are not well suited for intercession. An operation is *stable* if it always produces the same results for the same arguments. If a proxy can trap a stable operation, it can become unstable and thus unreliable. [Strict equality](#) (`===`) is one such stable operation. It can't be trapped and its result is computed by treating the proxy itself as just another object. Another way of maintaining stability is by applying an operation to the target instead of the proxy. As explained later, when we look at how invariants are enforced for proxies, this happens when `Object.getPrototypeOf()` is applied to a proxy whose target is non-extensible.

A second reason for not making more operations interceptable is that intercession means executing custom code in situations where that normally isn't possible. The more this interleaving of code happens, the harder it is to understand and debug a program. It also affects performance negatively.

28.5.4.3 Traps: `get` versus `invoke`

If you want to create virtual methods via ECMAScript 6 proxies, you have to return functions from a `get` trap. That raises the question: why not introduce an extra trap for method invocations (e.g. `invoke`)? That would enable us to distinguish between:

- Getting properties via `obj.prop` (trap `get`)
- Invoking methods via `obj.prop()` (trap `invoke`)

There are two reasons for not doing so.

First, not all implementations distinguish between `get` and `invoke`. For example, [Apple's JavaScriptCore](#) doesn't.

Second, extracting a method and invoking it later via `call()` or `apply()` should have the same effect as invoking the method via `dispatch`. In other words, the following two variants should work equivalently. If there was an extra trap `invoke` then that equivalence would be harder to maintain.

```
// Variant 1: call via dynamic dispatch
const result = obj.m();

// Variant 2: extract and call directly
const m = obj.m;
const result = m.call(obj);
```

28.5.4.3.1 Use cases for `invoke`

Some things can only be done if you are able to distinguish between `get` and `invoke`. Those things are therefore impossible with the current proxy API. Two examples are: auto-binding and intercepting missing methods. Let's examine how one would implement them if proxies supported `invoke`.

Auto-binding. By making a proxy the prototype of an object `obj`, you can automatically bind methods:

- Retrieving the value of a method `m` via `obj.m` returns a function whose `this` is bound to `obj`.
- `obj.m()` performs a method call.

Auto-binding helps with using methods as callbacks. For example, variant 2 from the previous example becomes simpler:

```
const boundMethod = obj.m;
const result = boundMethod();
```

Intercepting missing methods. `invoke` lets a proxy emulate the previously mentioned `__noSuchMethod__` mechanism that Firefox supports. The proxy would again become the prototype of an object `obj`. It would react differently depending on how an unknown property `foo` is accessed:

- If you read that property via `obj.foo`, no interception happens and `undefined` is returned.
- If you make the method call `obj.foo()` then the proxy intercepts and, e.g., notifies a callback.

28.5.5 Enforcing invariants for proxies

Before we look at what invariants are and how they are enforced for proxies, let's review how objects can be protected via non-extensibility and non-configurability.

28.5.5.1 Protecting objects

There are two ways of protecting objects:

- Non-extensibility protects objects
- Non-configurability protects properties (or rather, their attributes)

Non-extensibility. If an object is non-extensible, you can't add properties and you can't change its prototype:

```
'use strict'; // switch on strict mode to get TypeErrors

const obj = Object.preventExtensions({});
console.log(Object.isExtensible(obj)); // false
obj.foo = 123; // TypeError: object is not extensible
Object.setPrototypeOf(obj, null); // TypeError: object is not extensible
```

Non-configurability. All the data of a property is stored in *attributes*. A property is like a record and attributes are like the fields of that record. Examples of attributes:

- The attribute `value` holds the value of a property.
- The boolean attribute `writable` controls whether a property's value can be changed.
- The boolean attribute `configurable` controls whether a property's attributes can be changed.

Thus, if a property is both non-writable and non-configurable, it is read-only and remains that way:

```
'use strict'; // switch on strict mode to get TypeErrors

const obj = {};
Object.defineProperty(obj, 'foo', {
  value: 123,
  writable: false,
  configurable: false
});
console.log(obj.foo); // 123
obj.foo = 'a'; // TypeError: Cannot assign to read only property

Object.defineProperty(obj, 'foo', {
  configurable: true
}); // TypeError: Cannot redefine property
```

For more details on these topics (including how `Object.defineProperty()` works) consult the following sections in “Speaking JavaScript”:

- [Property Attributes and Property Descriptors](#)
- [Protecting Objects](#)

28.5.5.2 Enforcing invariants

Traditionally, non-extensibility and non-configurability are:

- Universal: they work for all objects.
- Monotonic: once switched on, they can't be switched off again.

These and other characteristics that remain unchanged in the face of language operations are called *invariants*. With proxies, it is easy to violate invariants, as they are not intrinsically bound by non-extensibility etc.

The proxy API prevents proxies from violating invariants by checking the parameters and results of handler methods. The following are four examples of invariants (for an arbitrary object `obj`) and how they are enforced for proxies (an exhaustive list is given at the end of this chapter).

The first two invariants involve non-extensibility and non-configurability. These are enforced by using the target object for bookkeeping: results returned by handler methods have to be mostly in sync with the target object.

- Invariant: If `Object.preventExtensions(obj)` returns `true` then all future calls must return `false` and `obj` must now be non-extensible.
 - Enforced for proxies by throwing a `TypeError` if the handler returns `true`, but the target object is not extensible.
- Invariant: Once an object has been made non-extensible, `Object.isExtensible(obj)` must always return `false`.
 - Enforced for proxies by throwing a `TypeError` if the result returned by the handler is not the same (after coercion) as `Object.isExtensible(target)`.

The remaining two invariants are enforced by checking return values:

- Invariant: `Object.isExtensible(obj)` must return a boolean.
 - Enforced for proxies by coercing the value returned by the handler to a boolean.
- Invariant: `Object.getOwnPropertyDescriptor(obj, ...)` must return an object or `undefined`.
 - Enforced for proxies by throwing a `TypeError` if the handler doesn't return an appropriate value.

Enforcing invariants has the following benefits:

- Proxies work like all other objects with regard to extensibility and configurability. Therefore, universality is maintained. This is achieved without preventing proxies from virtualizing (impersonating) protected objects.
- A protected object can't be misrepresented by wrapping a proxy around it. Misrepresentation can be caused by bugs or by malicious code.

The next two sections give examples of invariants being enforced.

28.5.5.3 Example: the prototype of a non-extensible target must be represented faithfully

In response to the `getPrototypeOf` trap, the proxy must return the target's prototype if the target is non-extensible.

To demonstrate this invariant, let's create a handler that returns a prototype that is different from the target's prototype:

```
const fakeProto = {};  
const handler = {  
  getPrototypeOf(t) {  
    return fakeProto;  
  }  
};
```

Faking the prototype works if the target is extensible:

```
const extensibleTarget = {};  
const ext = new Proxy(extensibleTarget, handler);  
console.log(Object.getPrototypeOf(ext) === fakeProto); // true
```

We do, however, get an error if we fake the prototype for a non-extensible object.

```
const nonExtensibleTarget = {};  
Object.preventExtensions(nonExtensibleTarget);  
const nonExt = new Proxy(nonExtensibleTarget, handler);  
Object.getPrototypeOf(nonExt); // TypeError
```

28.5.5.4 Example: non-writable non-configurable target properties must be represented faithfully

If the target has a non-writable non-configurable property then the handler must return that property's value in response to a `get` trap. To demonstrate this invariant, let's create a handler that always returns the same value for properties.

```
const handler = {
  get(target, propKey) {
    return 'abc';
  }
};
const target = Object.defineProperties(
  {}, {
    foo: {
      value: 123,
      writable: true,
      configurable: true
    },
    bar: {
      value: 456,
      writable: false,
      configurable: false
    }
  }
);
const proxy = new Proxy(target, handler);
```

Property `target.foo` is not both non-writable and non-configurable, which means that the handler is allowed to pretend that it has a different value:

```
> proxy.foo
'abc'
```

However, property `target.bar` is both non-writable and non-configurable. Therefore, we can't fake its value:

```
> proxy.bar
TypeError: Invariant check failed
```

28.6 FAQ: proxies

28.6.1 Where is the `enumerate` trap?

ES6 originally had a trap `enumerate` that was triggered by `for-in` loops. But it was recently removed, to simplify proxies. `Reflect.enumerate()` was removed, as well. ([Source: TC39 notes](#))

28.7 Reference: the proxy API

This section serves as a quick reference for the proxy API: the global objects `Proxy` and `Reflect`.

28.7.1 Creating proxies

There are two ways to create proxies:

- `const proxy = new Proxy(target, handler)`
Creates a new proxy object with the given target and the given handler.
- `const {proxy, revoke} = Proxy.revocable(target, handler)`
Creates a proxy that can be revoked via the function `revoke`. `revoke` can be called multiple times, but only the first call has an effect and switches proxy off. Afterwards, any operation performed on `proxy` leads to a `TypeError` being thrown.

28.7.2 Handler methods

This subsection explains what traps can be implemented by handlers and what operations trigger them. Several traps return boolean values. For the traps `has` and `isExtensible`, the boolean is the result of the operation. For all other traps, the boolean indicates whether the operation succeeded or not.

Traps for all objects:

- `defineProperty(target, propKey, propDesc) : boolean`
 - `Object.defineProperty(proxy, propKey, propDesc)`
- `deleteProperty(target, propKey) : boolean`
 - `delete proxy[propKey]`
 - `delete proxy.foo // propKey = 'foo'`
- `get(target, propKey, receiver) : any`
 - `receiver[propKey]`
 - `receiver.foo // propKey = 'foo'`
- `getOwnPropertyDescriptor(target, propKey) : PropDesc|Undefined`
 - `Object.getOwnPropertyDescriptor(proxy, propKey)`
- `getPrototypeOf(target) : Object|Null`
 - `Object.getPrototypeOf(proxy)`
- `has(target, propKey) : boolean`
 - `propKey in proxy`
- `isExtensible(target) : boolean`
 - `Object.isExtensible(proxy)`
- `ownKeys(target) : Array<PropertyKey>`

- `Object.getOwnPropertyNames(proxy)` (only uses string keys)
- `Object.getOwnPropertySymbols(proxy)` (only uses symbol keys)
- `Object.keys(proxy)` (only uses enumerable string keys; enumerability is checked via `Object.getOwnPropertyDescriptor`)
- `preventExtensions(target)` : boolean
 - `Object.preventExtensions(proxy)`
- `set(target, propKey, value, receiver)` : boolean
 - `receiver[propKey] = value`
 - `receiver.foo = value // propKey = 'foo'`
- `setPrototypeOf(target, proto)` : boolean
 - `Object.setPrototypeOf(proxy, proto)`

Traps for functions (available if target is a function):

- `apply(target, thisArgument, argumentsList)` : any
 - `proxy.apply(thisArgument, argumentsList)`
 - `proxy.call(thisArgument, ...argumentsList)`
 - `proxy(...argumentsList)`
- `construct(target, argumentsList, newTarget)` : Object
 - `new proxy(...argumentsList)`

28.7.2.1 Fundamental operations versus derived operations

The following operations are *fundamental*, they don't use other operations to do their work: `apply`, `defineProperty`, `deleteProperty`, `getOwnPropertyDescriptor`, `getPrototypeOf`, `isExtensible`, `ownKeys`, `preventExtensions`, `setPrototypeOf`

All other operations are *derived*, they can be implemented via fundamental operations. For example, for data properties, `get` can be implemented by iterating over the prototype chain via `getPrototypeOf` and calling `getOwnPropertyDescriptor` for each chain member until either an own property is found or the chain ends.

28.7.3 Invariants of handler methods

Invariants are safety constraints for handlers. This subsection documents what invariants are enforced by the proxy API and how. Whenever you read “the handler must do X” below, it means that a `TypeError` is thrown if it doesn't. Some invariants restrict return values, others restrict parameters. The correctness of a trap's return value is ensured in two ways: Normally, an illegal value means that a `TypeError` is thrown. But whenever a boolean is expected, coercion is used to convert non-booleans to legal values.

This is the complete list of invariants that are enforced:

- `apply(target, thisArgument, argumentsList)`
 - No invariants are enforced.
- `construct(target, argumentsList, newTarget)`
 - The result returned by the handler must be an object (not `null` or a primitive value).
- `defineProperty(target, propKey, propDesc)`
 - If the target is not extensible then you can't add properties and `propKey` must be one of the own keys of the target.
 - If `propDesc` sets the attribute `configurable` to `false` then the target must have a non-configurable own property whose key is `propKey`.
 - If `propDesc` were to be used to (re)define an own property for the target then that must not cause an exception. An exception is thrown if a change is forbidden by the attributes `writable` and `configurable` (non-extensibility is handled by the first rule).
- `deleteProperty(target, propKey)`
 - Non-configurable own properties of the target can't be deleted.
- `get(target, propKey, receiver)`
 - If the target has an own, non-writable, non-configurable data property whose key is `propKey` then the handler must return that property's value.
 - If the target has an own, non-configurable, getter-less accessor property then the handler must return `undefined`.
- `getOwnPropertyDescriptor(target, propKey)`
 - The handler must return either an object or `undefined`.
 - Non-configurable own properties of the target can't be reported as non-existent by the handler.
 - If the target is non-extensible then exactly the target's own properties must be reported by the handler as existing.
 - If the handler reports a property as non-configurable then that property must be a non-configurable own property of the target.
 - If the result returned by the handler were used to (re)define an own property for the target then that must not cause an exception. An exception is thrown if the change is not allowed by the attributes `writable` and `configurable` (non-extensibility is handled by the third rule). Therefore, the handler can't report a non-configurable property as configurable and it can't report a different value for a non-configurable non-writable property.
- `getPrototypeOf(target)`
 - The result must be either an object or `null`.
 - If the target object is not extensible then the handler must return the

prototype of the target object.

- `has(target, propKey)`
 - A handler must not hide (report as non-existent) a non-configurable own property of the target.
 - If the target is non-extensible then no own property of the target may be hidden.
- `isExtensible(target)`
 - The result returned by the handler is coerced to boolean.
 - After coercion to boolean, the value returned by the handler must be the same as `target.isExtensible()`.
- `ownKeys(target)`
 - The handler must return an object, which treated as Array-like and converted into an Array.
 - Each element of the result must be either a string or a symbol.
 - The result must contain the keys of all non-configurable own properties of the target.
 - If the target is not extensible then the result must contain exactly the keys of the own properties of the target (and no other values).
- `preventExtensions(target)`
 - The result returned by the handler is coerced to boolean.
 - If the handler returns a truthy value (indicating a successful change) then `target.isExtensible()` must be `false` afterwards.
- `set(target, propKey, value, receiver)`
 - If the target has an own, non-writable, non-configurable data property whose key is `propKey` then `value` must be the same as the value of that property (i.e., the property can't be changed).
 - If the target has an own, non-configurable, setter-less accessor property then a `TypeError` is thrown (i.e., such a property can't be set).
- `setPrototypeOf(target, proto)`
 - The result returned by the handler is coerced to boolean.
 - If the target is not extensible, the prototype can't be changed. This is enforced as follows: If the target is not extensible and the handler returns a truthy value (indicating a successful change) then `proto` must be the same as the prototype of the target. Otherwise, a `TypeError` is thrown.



In the spec, the invariants are listed in the section "[Proxy Object Internal Methods and Internal Slots](#)".

28.7.4 Operations that affect the prototype chain

The following operations of normal objects perform operations on objects in the prototype chain. Therefore, if one of the objects in that chain is a proxy, its traps are triggered. The specification implements the operations as internal own methods (that are not visible to JavaScript code). But in this section, we pretend that they are normal methods that have the same names as the traps. The parameter `target` becomes the receiver of the method call.

- `target.get(propertyKey, receiver)`
If `target` has no own property with the given key, `get` is invoked on the prototype of `target`.
- `target.has(propertyKey)`
Similarly to `get`, `has` is invoked on the prototype of `target` if `target` has no own property with the given key.
- `target.set(propertyKey, value, receiver)`
Similarly to `get`, `set` is invoked on the prototype of `target` if `target` has no own property with the given key.

All other operations only affect own properties, they have no effect on the prototype chain.



In the spec, these (and other) operations are described in the section "[Ordinary Object Internal Methods and Internal Slots](#)".

28.7.5 Reflect

The global object `Reflect` implements all interceptable operations of the JavaScript meta object protocol as methods. The names of those methods are the same as those of the handler methods, which, [as we have seen](#), helps with forwarding operations from the handler to the target.

- `Reflect.apply(target, thisArgument, argumentsList) : any`
Same as `Function.prototype.apply()`.
- `Reflect.construct(target, argumentsList, newTarget=target) : Object`
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. More information on how constructor calls are chained in ES6 is given in [the chapter on classes](#).

- `Reflect.defineProperty(target, propertyKey, propDesc) : boolean`
Similar to `Object.defineProperty()`.
- `Reflect.deleteProperty(target, propertyKey) : boolean`
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. The only way to protect properties from deletion is by making them non-configurable. In sloppy mode, the `delete` operator returns the same results. But in strict mode, it throws a `TypeError` instead of returning `false`.
- `Reflect.get(target, propertyKey, receiver=target) : any`
A function that gets properties. The optional parameter `receiver` is needed when `get` reaches a getter later in the prototype chain. Then it provides the value for `this`.
- `Reflect.getOwnPropertyDescriptor(target, propertyKey) : PropDesc|Undefined`
Same as `Object.getOwnPropertyDescriptor()`.
- `Reflect.getPrototypeOf(target) : Object|Null`
Same as `Object.getPrototypeOf()`.
- `Reflect.has(target, propertyKey) : boolean`
The `in` operator as a function.
- `Reflect.isExtensible(target) : boolean`
Same as `Object.isExtensible()`.
- `Reflect.ownKeys(target) : Array<PropertyKey>`
Returns all own property keys (strings and symbols!) in an Array.
- `Reflect.preventExtensions(target) : boolean`
Similar to `Object.preventExtensions()`.
- `Reflect.set(target, propertyKey, value, receiver=target) : boolean`
A function that sets properties.
- `Reflect.setPrototypeOf(target, proto) : boolean`
The new standard way of setting the prototype of an object. The current non-standard way, that works in most engines, is to set the special property `__proto__`.

Several methods have boolean results. For `has` and `isExtensible`, they are the results of the operation. For the remaining methods, they indicate whether the operation succeeded.

28.7.5.1 Use cases for `Reflect` besides forwarding

Apart from forwarding operations, [why is `Reflect` useful \[4\]](#)?

- 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`
 - `Object.preventExtensions(obj) : Object`
 - `Object.setPrototypeOf(obj, proto) : Object`
- Operators as functions: The following `Reflect` methods implement functionality that is otherwise only available via operators:
 - `Reflect.construct(target, argumentsList, newTarget=target) : Object`
 - `Reflect.deleteProperty(target, propertyKey) : boolean`
 - `Reflect.get(target, propertyKey, receiver=target) : any`
 - `Reflect.has(target, propertyKey) : boolean`
 - `Reflect.set(target, propertyKey, value, receiver=target) : boolean`
- Shorter version of `apply()`: If you want to be completely safe about invoking the method `apply()` on a function, you 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 you try to delete a non-configurable own property.
`Reflect.deleteProperty()` returns `false` in that case.

28.7.5.2 `Object.*` versus `Reflect.*`

Going forward, `Object` will host operations that are of interest to normal applications, while `Reflect` will host operations that are more low-level.

28.8 Conclusion

This concludes our in-depth look at the proxy API. For each application, you have to take performance into consideration and – if necessary – measure. Proxies may not always be fast enough. On the other hand, performance is often not crucial and it is nice

to have the metaprogramming power that proxies give us. As we have seen, there are numerous use cases they can help with.

28.9 Further reading

[1] “[On the design of the ECMAScript Reflection API](#)” by Tom Van Cutsem and Mark Miller. Technical report, 2012. [Important source of this chapter.]

[2] “[The Art of the Metaobject Protocol](#)” by Gregor Kiczales, Jim des Rivieres and Daniel G. Bobrow. Book, 1991.

[3] “[Putting Metaclasses to Work: A New Dimension in Object-Oriented Programming](#)” by Ira R. Forman and Scott H. Danforth. Book, 1999.

[4] “[Harmony-reflect: Why should I use this library?](#)” by Tom Van Cutsem. [Explains why Reflect is useful.]

29. Coding style tips for ECMAScript 6

This chapter lists a few ideas related to ES6 coding style:

- `var` versus `let` versus `const` (details are explained in [the chapter on variables](#)):
 - Prefer `const`. You can use it for all variables whose values never change.
 - Otherwise, use `let` — for variables whose values do change.
 - Avoid `var`.
- An arrow function is the superior solution whenever a function fits into a single line:

```
readFilePromisified(filename)
  .then(text => console.log(text))
```

For multi-line functions, traditional functions work well, too (with the caveat of this not being lexical):

```
readFilePromisified(filename)
  .then(function(text) {
    const obj = JSON.parse(text);
    console.log(JSON.stringify(obj, null, 4));
  });
```

Single-line functions tend to be throw-away. If a function isn't then a traditional function has the advantage that you can name it, which is useful for documentation and debugging.

- Properties in object literals: As soon as an object literal spans multiple lines, I add a comma after the last entry. Such a trailing comma has been legal since ES5. It makes adding, removing and rearranging entries simpler. As a consequence, method definitions always end with `,`:

```
const obj = {
  foo() {
  },
  bar() {
  },
};
```

- Modules: don't mix default exports and named exports. Your module should either specialize on a single thing or export multiple, named, things. Details are explained in [the chapter on modules](#).
- Format generators as follows:

```
// Generator function declaration
function* genFunc() { ... }

// Generator function expression
const genFunc = function* () { ... };

// Generator method definition in an object literal
const obj = {
  * generatorMethod() {
    ...
  }
};

// Generator method definition in a class definition
class MyClass {
  * generatorMethod() {
    ...
  }
}
```

Details are explained in [the chapter on generators](#).

- The chapter on parameter handling has [style tips for function signatures](#):

```
// Mark optional parameters via the parameter default value `undefined`
function foo(optional = undefined) { ... }
```

```
// Mark required parameters via a function that throws an exception
function foo(required = throwException()) { ... }

// Enforcing a maximum arity (variant 1 of 2)
function f(x, y, ...empty) { // max arity: 2
  if (empty.length > 0) {
    throw new Error();
  }
}
// Enforcing a maximum arity (variant 2 of 2)
function f(x, y) { // max arity: 2
  if (arguments.length > 2) {
    throw new Error();
  }
}
```

- In the [chapter on callable entities](#) (traditional functions, arrow functions, classes, etc.) there is a section that gives recommendations (when to use which one etc.).

Additionally, the [ES5 coding style tips](#) in “Speaking JavaScript” are still relevant for ES6.

30. An overview of what’s new in ES6

This chapter collects the overview sections of all the chapters in this book.

30.1 Categories of ES6 features

The introduction of the ES6 specification lists all new features:

Some of [ECMAScript 6’s] major enhancements include modules, class declarations, lexical block scoping, iterators and generators, promises for asynchronous programming, destructuring patterns, and proper tail calls. The ECMAScript library of built-ins has been expanded to support additional data abstractions including maps, sets, and arrays of binary numeric values as well as additional support for Unicode supplemental characters in strings and regular expressions. The built-ins are now extensible via subclassing.

There are three major categories of features:

- Better syntax for features that already exist (e.g. via libraries). For example:
 - [Classes](#)
 - [Modules](#)
- New functionality in the standard library. For example:
 - New methods for [strings](#) and [Arrays](#)
 - [Promises](#)
 - [Maps, Sets](#)
- Completely new features. For example:
 - [Generators](#)
 - [Proxies](#)
 - [WeakMaps](#)

30.2 New number and `Math` features

30.2.1 New integer literals

You can now specify integers in binary and octal notation:

```
> 0xFF // ES5: hexadecimal
255
> 0b11 // ES6: binary
3
> 0o10 // ES6: octal
8
```

30.2.2 New `Number` properties

The global object `Number` gained a few new properties. Among others:

- `Number.EPSILON` for comparing floating point numbers with a tolerance for rounding errors.
- A method and constants for determining whether a JavaScript integer is *safe* (within the signed 53 bit range in which there is no loss of precision):
 - `Number.isSafeInteger(number)`
 - `Number.MIN_SAFE_INTEGER`
 - `Number.MAX_SAFE_INTEGER`

30.2.3 New `Math` methods

The global object `Math` has new methods for numerical, trigonometric and bitwise operations. Let’s look at four examples.

`Math.sign()` returns the sign of a number:

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

`Math.trunc()` removes the decimal fraction of a number:

```
> Math.trunc(3.1)
3
> Math.trunc(3.9)
3
> Math.trunc(-3.1)
-3
> Math.trunc(-3.9)
-3
```

`Math.log10()` computes the logarithm to base 10:

```
> Math.log10(100)
2
```

`Math.hypot()` Computes the square root of the sum of the squares of its arguments (Pythagoras' theorem):

```
> Math.hypot(3, 4)
5
```

30.3 New string features

New string methods:

```
> 'hello'.startsWith('hell')
true
> 'hello'.endsWith('ello')
true
> 'hello'.includes('ell')
true
> 'doo '.repeat(3)
'doo doo doo '
```

ES6 has a new kind of string literal, the *template literal*:

```
// String interpolation via template literals (in backticks)
const first = 'Jane';
const last = 'Doe';
console.log(`Hello ${first} ${last}!`);
// Hello Jane Doe!

// Template literals also let you create strings with multiple lines
const multiLine = `
This is
a string
with multiple
lines`;
```

30.4 Symbols

Symbols are a new primitive type in ECMAScript 6.

30.4.1 Use case 1: unique property keys

Symbols are mainly used as unique property keys – a symbol never clashes with any other property key (symbol or string). For example, you can make an object *iterable* (usable via the `for-of` loop and other language mechanisms), by using the symbol stored in `Symbol.iterator` as the key of a method (more information on iterables is given in [the chapter on iteration](#)):

```
const iterableObject = {
  [Symbol.iterator]() { // (A)
    const data = ['hello', 'world'];
    let index = 0;
    return {
      next() {
        if (index < data.length) {
          return { value: data[index++] };
        } else {
          return { done: true };
        }
      }
    };
  }
}

for (const x of iterableObject) {
  console.log(x);
}

// Output:
// hello
// world
```

In line A, a symbol is used as the key of the method. This unique marker makes the object iterable and enables us to use the `for-of` loop.

30.4.2 Use case 2: constants representing concepts

In ECMAScript 5, you may have used strings to represent concepts such as colors. In ES6, you can use symbols and be sure that they are always unique:

```
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);
  }
}
```

30.4.3 Pitfall: you can't coerce symbols to strings

Coercing (implicitly converting) symbols to strings throws exceptions:

```
const sym = Symbol('desc');

const str1 = '' + sym; // TypeError
const str2 = `${sym}`; // TypeError
```

The only solution is to convert explicitly:

```
const str2 = String(sym); // 'Symbol(desc)'
const str3 = sym.toString(); // 'Symbol(desc)'
```

Forbidding coercion prevents some errors, but also makes working with symbols more complicated.

30.4.4 Which operations related to property keys are aware of symbols?

The following operations are aware of symbols as property keys:

- `Reflect.ownKeys()`
- Property access via `[]`
- `Object.assign()`

The following operations ignore symbols as property keys:

- `Object.keys()`
- `Object.getOwnPropertyNames()`
- `for-in` loop

30.5 Template literals

ES6 has two new kinds of literals: *template literals* and *tagged template literals*. These two literals have similar names and look similar, but they are quite different. It is therefore important to distinguish:

- Web templates (data): HTML with blanks to be filled in
- Template literals (code): multi-line string literals plus interpolation
- Tagged template literals (code): function calls

Template literals are string literals that can stretch across multiple lines and include interpolated expressions:

```
const firstName = 'Jane';
console.log(`Hello ${firstName}!
How are you
today?`);

// Output:
// Hello Jane!
// How are you
// today?
```

Tagged template literals (short: *tagged templates*) are created by mentioning a function before a template literal:

```
> String.raw`A \tagged\ template`
'A \\tagged\\ template'
```

Tagged templates are function calls. In the previous example, the method `String.raw` is called to produce the result of the tagged template.

30.6 Variables and scoping

ES6 provides two new ways of declaring variables: `let` and `const`, which mostly replace the ES5 way of declaring variables, `var`.

30.6.1 `let`

`let` works similarly to `var`, but the variable it declares is *block-scoped*, it only exists within the current block. `var` is *function-scoped*.

In the following code, you can see that the `let`-declared variable `tmp` only exists with the block that starts in line A:

```
function order(x, y) {  
  if (x > y) { // (A)  
    let tmp = x;  
    x = y;  
    y = tmp;  
  }  
  console.log(tmp===x); // ReferenceError: tmp is not defined  
  return [x, y];  
}
```

30.6.2 `const`

`const` works like `let`, but the variable you declare must be immediately initialized, with a value that can't be changed afterwards.

```
const foo;  
// SyntaxError: missing = in const declaration  
  
const bar = 123;  
bar = 456;  
// TypeError: `bar` is read-only
```

Since `for-of` creates one *binding* (storage space for a variable) per loop iteration, it is OK to `const`-declare the loop variable:

```
for (const x of ['a', 'b']) {  
  console.log(x);  
}  
// Output:  
// a  
// b
```

30.6.3 Ways of declaring variables

The following table gives an overview of six ways in which variables can be declared in ES6:

| | Hoisting | Scope | Creates global properties |
|-----------------------|--------------------|---------------|---------------------------|
| <code>var</code> | Declaration | Function | Yes |
| <code>let</code> | Temporal dead zone | Block | No |
| <code>const</code> | Temporal dead zone | Block | No |
| <code>function</code> | Complete | Block | Yes |
| <code>class</code> | No | Block | No |
| <code>import</code> | Complete | Module-global | No |

30.7 Destructuring

Destructuring is a convenient way of extracting values from data stored in (possibly nested) objects and Arrays. It can be used in locations that receive data (such as the left-hand side of an assignment). How to extract the values is specified via patterns (read on for examples).

30.7.1 Object destructuring

Destructuring objects:

```
const obj = { first: 'Jane', last: 'Doe' };  
const {first: f, last: l} = obj;  
// f = 'Jane'; l = 'Doe'  
  
// {prop} is short for {prop: prop}  
const {first, last} = obj;  
// first = 'Jane'; last = 'Doe'
```

Destructuring helps with processing return values:

```
const obj = { foo: 123 };  
  
const {writable, configurable} =  
  Object.getOwnPropertyDescriptor(obj, 'foo');  
  
console.log(writable, configurable); // true true
```

30.7.2 Array destructuring

Array destructuring (works for all iterable values):

```
const iterable = ['a', 'b'];
const [x, y] = iterable;
// x = 'a'; y = 'b'
```

Destructuring helps with processing return values:

```
const [all, year, month, day] =
  /^(\d\d\d\d)-(\d\d)-(\d\d)$/
  .exec('2999-12-31');
```

30.7.3 Where can destructuring be used?

Destructuring can be used in the following locations:

```
// Variable declarations:
const [x] = ['a'];
let [x] = ['a'];
var [x] = ['a'];

// Assignments:
[x] = ['a'];

// Parameter definitions:
function f([x]) { ... }
f(['a']);
```

You can also destructure in a `for-of` loop:

```
const arr1 = ['a', 'b'];
for (const [index, element] of arr1.entries()) {
  console.log(index, element);
}
// Output:
// 0 a
// 1 b

const arr2 = [
  {name: 'Jane', age: 41},
  {name: 'John', age: 40},
];
for (const {name, age} of arr2) {
  console.log(name, age);
}
// Output:
// Jane 41
// John 40
```

30.8 Parameter handling

Parameter handling has been significantly upgraded in ECMAScript 6. It now supports parameter default values, rest parameters (varargs) and destructuring.

Default parameter values:

```
function findClosestShape(x=0, y=0) {
  // ...
}
```

Rest parameters:

```
function format(pattern, ...params) {
  return params;
}
console.log(format('a', 'b', 'c')); // ['b', 'c']
```

Named parameters via destructuring:

```
function selectEntries({ start=0, end=-1, step=1 } = {}) {
  // The object pattern is an abbreviation of:
  // { start: start=0, end: end=-1, step: step=1 }

  // Use the variables `start`, `end` and `step` here
  ...

}

selectEntries({ start: 10, end: 30, step: 2 });
selectEntries({ step: 3 });
selectEntries({});
selectEntries();
```

30.8.1 Spread operator (...)

In function and constructor calls, the spread operator turns iterable values into arguments:

```
> Math.max(-1, 5, 11, 3)
11
> Math.max(...[-1, 5, 11, 3])
11
> Math.max(-1, ...[-1, 5, 11], 3)
11
```

In Array literals, the spread operator turns iterable values into Array elements:

```
> [1, ...[2,3], 4]
[1, 2, 3, 4]
```

30.9 Callable entities in ECMAScript 6

In ES5, a single construct, the (traditional) function, played three roles:

- Real (non-method) function
- Method
- Constructor

In ES6, there is more specialization. The three duties are now handled as follows (a class definition is either one of the two constructs for creating classes – a class declaration or a class expression):

- Real (non-method) function:
 - Arrow functions (only have an expression form)
 - Traditional functions (created via function expressions and function declarations)
 - Generator functions (created via generator function expressions and generator function declarations)
- Method:
 - Methods (created by method definitions in object literals and class definitions)
 - Generator methods (created by generator method definitions in object literals and class definitions)
- Constructor:
 - Classes (created via class definitions)

This list is a simplification. There are quite a few libraries that use `this` as an implicit parameter for callbacks. Then you have to use traditional functions.

Note that I distinguish:

- The entity: e.g. traditional function
- The syntax that creates the entity: e.g. function expression and function declaration

Even though their behaviors differ (as explained later), all of these entities are functions. For example:

```
> typeof (() => {}) // arrow function
'function'
> typeof function* () {} // generator function
'function'
> typeof class {} // class
'function'
```

30.10 Arrow functions

There are two benefits to arrow functions.

First, they are less verbose than traditional function expressions:

```
const arr = [1, 2, 3];
const squares = arr.map(x => x * x);

// Traditional function expression:
const squares = arr.map(function (x) { return x * x });
```

Second, their `this` is picked up from surroundings (*lexical*). Therefore, you don't need `bind()` or `that = this`, anymore.

```
function UiComponent() {
  const button = document.getElementById('myButton');
  button.addEventListener('click', () => {
    console.log('CLICK');
    this.handleClick(); // lexical `this`
  });
}
```

The following variables are all lexical inside arrow functions:

- arguments
- super
- this
- new.target

30.11 New OOP features besides classes

30.11.1 New object literal features

Method definitions:

```
const obj = {
  myMethod(x, y) {
    ...
  }
};
```

Property value shorthands:

```
const first = 'Jane';
const last = 'Doe';

const obj = { first, last };
// Same as:
const obj = { first: first, last: last };
```

Computed property keys:

```
const propKey = 'foo';
const obj = {
  [propKey]: true,
  ['b'+ 'ar']: 123
};
```

This new syntax can also be used for method definitions:

```
const obj = {
  ['h'+ 'ello']() {
    return 'hi';
  }
};
console.log(obj.hello()); // hi
```

The main use case for computed property keys is to make it easy to use symbols as property keys.

30.11.2 New methods in Object

The most important new method of `Object` is `assign()`. Traditionally, this functionality was called `extend()` in the JavaScript world. In contrast to how this classic operation works, `Object.assign()` only considers *own* (non-inherited) properties.

```
const obj = { foo: 123 };
Object.assign(obj, { bar: true });
console.log(JSON.stringify(obj));
// {"foo":123,"bar":true}
```

30.12 Classes

A class and a subclass:

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  toString() {
    return `${this.x}, ${this.y}`;
  }
}

class ColorPoint extends Point {
  constructor(x, y, color) {
    super(x, y);
    this.color = color;
  }
  toString() {
    return super.toString() + ' in ' + this.color;
  }
}
```

Using the classes:

```
> const cp = new ColorPoint(25, 8, 'green');

> cp.toString();
' (25, 8) in green'

> cp instanceof ColorPoint
true
> cp instanceof Point
true
```

Under the hood, ES6 classes are not something that is radically new: They mainly provide more convenient syntax to create old-school constructor functions. You can see that if you use `typeof`:

```
> typeof Point
'function'
```

30.13 Modules

JavaScript has had modules for a long time. However, they were implemented via libraries, not built into the language. ES6 is the first time that JavaScript has built-in modules.

ES6 modules are stored in files. There is exactly one module per file and one file per module. You have two ways of exporting things from a module. [These two ways can be mixed](#), but it is usually better to use them separately.

30.13.1 Multiple named exports

There can be multiple *named exports*:

```
//----- lib.js -----
export const sqrt = Math.sqrt;
export function square(x) {
  return x * x;
}
export function diag(x, y) {
  return sqrt(square(x) + square(y));
}

//----- main.js -----
import { square, diag } from 'lib';
console.log(square(11)); // 121
console.log(diag(4, 3)); // 5
```

You can also import the complete module:

```
//----- main.js -----
import * as lib from 'lib';
console.log(lib.square(11)); // 121
console.log(lib.diag(4, 3)); // 5
```

30.13.2 Single default export

There can be a single *default export*. For example, a function:

```
//----- myFunc.js -----
export default function () { ... } // no semicolon!

//----- main1.js -----
import myFunc from 'myFunc';
myFunc();
```

Or a class:

```
//----- MyClass.js -----
export default class { ... } // no semicolon!

//----- main2.js -----
import MyClass from 'MyClass';
const inst = new MyClass();
```

Note that there is no semicolon at the end if you default-export a function or a class (which are anonymous declarations).

30.13.3 Browsers: scripts versus modules

| | Scripts | Modules |
|--|---------------|------------------------|
| HTML element | <script> | <script type="module"> |
| Default mode | non-strict | strict |
| Top-level variables are | global | local to module |
| Value of this at top level | window | undefined |
| Executed | synchronously | asynchronously |
| Declarative imports (import statement) | no | yes |
| Programmatic imports (Promise-based API) | yes | yes |
| File extension | .js | .js |

30.14 The for-of loop

for-of is a new loop in ES6 that replaces both for-in and forEach() and supports the new iteration protocol.

Use it to loop over *iterable* objects (Arrays, strings, Maps, Sets, etc.; see Chap. “Iterables and iterators”):

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

// Output:
// a
// b
```

break and continue work inside for-of loops:

```
for (const x of ['a', '', 'b']) {
  if (x.length === 0) break;
  console.log(x);
}

// Output:
// a
```

Access both elements and their indices while looping over an Array (the square brackets before `of` mean that we are using [destructuring](#)):

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

// Output:
// 0. a
// 1. b
```

Looping over the [key, value] entries in a Map (the square brackets before `of` mean that we are using [destructuring](#)):

```
const map = new Map([
  [false, 'no'],
  [true, 'yes'],
]);
for (const [key, value] of map) {
  console.log(`${key} => ${value}`);
}

// Output:
// false => no
// true => yes
```

30.15 New Array features

New static Array methods:

- `Array.from(arrayLike, mapFunc?, thisArg?)`
- `Array.of(...items)`

New `Array.prototype` methods:

- **Iterating:**
 - `Array.prototype.entries()`
 - `Array.prototype.keys()`
 - `Array.prototype.values()`
- **Searching for elements:**
 - `Array.prototype.find(predicate, thisArg?)`
 - `Array.prototype.findIndex(predicate, thisArg?)`
- `Array.prototype.copyWithIn(target, start, end=this.length)`
- `Array.prototype.fill(value, start=0, end=this.length)`

30.16 Maps and Sets

Among others, the following four data structures are new in ECMAScript 6: `Map`, `WeakMap`, `Set` and `WeakSet`.

30.16.1 Maps

The keys of a `Map` can be arbitrary values:

```
> const map = new Map(); // create an empty Map
> const KEY = {};

> map.set(KEY, 123);
> map.get(KEY)
123
> map.has(KEY)
true
> map.delete(KEY);
true
> map.has(KEY)
false
```

You can use an Array (or any iterable) with [key, value] pairs to set up the initial data in the `Map`:

```
const map = new Map([
  [1, 'one'],
  [2, 'two'],
  [3, 'three'], // trailing comma is ignored
]);
```

30.16.2 Sets

A `Set` is a collection of unique elements:

```
const arr = [5, 1, 5, 7, 7, 5];
const unique = [...new Set(arr)]; // [ 5, 1, 7 ]
```

As you can see, you can initialize a `Set` with elements if you hand the constructor an iterable (`arr` in the example) over those elements.

30.16.3 WeakMaps

A `WeakMap` is a `Map` that doesn't prevent its keys from being garbage-collected. That means that you can associate data with objects without having to worry about memory leaks. For example:

```
//----- Manage listeners

const _objToListeners = new WeakMap();

function addListener(obj, listener) {
  if (!_objToListeners.has(obj)) {
    _objToListeners.set(obj, new Set());
  }
  _objToListeners.get(obj).add(listener);
}

function triggerListeners(obj) {
  const listeners = _objToListeners.get(obj);
  if (listeners) {
    for (const listener of listeners) {
      listener();
    }
  }
}

//----- Example: attach listeners to an object

const obj = {};
addListener(obj, () => console.log('hello'));
addListener(obj, () => console.log('world'));

//----- Example: trigger listeners

triggerListeners(obj);

// Output:
// hello
// world
```

30.17 Typed Arrays

Typed Arrays are an ECMAScript 6 API for handling binary data.

Code example:

```
const typedArray = new Uint8Array([0,1,2]);
console.log(typedArray.length); // 3
typedArray[0] = 5;
const normalArray = [...typedArray]; // [5,1,2]

// The elements are stored in typedArray.buffer.
// Get a different view on the same data:
const dataView = new DataView(typedArray.buffer);
console.log(dataView.getUint8(0)); // 5
```

Instances of `ArrayBuffer` store the binary data to be processed. Two kinds of *views* are used to access the data:

- Typed Arrays (`Uint8Array`, `Int16Array`, `Float32Array`, etc.) interpret the `ArrayBuffer` as an indexed sequence of elements of a single type.
- Instances of `DataView` let you access data as elements of several types (`Uint8`, `Int16`, `Float32`, etc.), at any byte offset inside an `ArrayBuffer`.

The following browser APIs support Typed Arrays ([details are mentioned in a dedicated section](#)):

- File API
- XMLHttpRequest
- Fetch API
- Canvas
- WebSockets
- And more

30.18 Iterables and iterators

ES6 introduces a new mechanism for traversing data: *iteration*. Two concepts are central to iteration:

- An *iterable* is a data structure that wants to make its elements accessible to the public. It does so by implementing a method whose key is `Symbol.iterator`. That method is a factory for *iterators*.
- An *iterator* is a pointer for traversing the elements of a data structure (think cursors in databases).

Expressed as interfaces in TypeScript notation, these roles look like this:

```
interface Iterable {
  [Symbol.iterator]() : Iterator;
}
interface Iterator {
  next() : IteratorResult;
}
interface IteratorResult {
  value: any;
  done: boolean;
}
```

30.18.1 Iterable values

The following values are iterable:

- Arrays
- Strings
- Maps
- Sets
- DOM data structures (work in progress)

Plain objects are not iterable (why is explained in [a dedicated section](#)).

30.18.2 Constructs supporting iteration

Language constructs that access data via iteration:

- Destructuring via an Array pattern:

```
const [a,b] = new Set(['a', 'b', 'c']);
```

- for-of loop:

```
for (const x of ['a', 'b', 'c']) {
  console.log(x);
}
```

- Array.from():

```
const arr = Array.from(new Set(['a', 'b', 'c']));
```

- Spread operator (...):

```
const arr = [...new Set(['a', 'b', 'c'])];
```

- Constructors of Maps and Sets:

```
const map = new Map([[false, 'no'], [true, 'yes']]);
const set = new Set(['a', 'b', 'c']);
```

- Promise.all(), Promise.race():

```
Promise.all(iterableOverPromises).then(...);
Promise.race(iterableOverPromises).then(...);
```

- yield*:

```
yield* anIterable;
```

30.19 Generators

Generators, a new feature of ES6, are functions that can be paused and resumed (think cooperative multitasking or coroutines). That helps with many applications. Two important ones are:

1. Implementing iterables
2. Blocking on asynchronous function calls

The following subsections give brief overviews of these applications.

30.19.1 Implementing iterables via generators

The following function returns an iterable over the properties of an object, one [key, value] pair per property:

```
// The asterisk after `function` means that
// `objectEntries` is a generator
function* objectEntries(obj) {
  const propKeys = Reflect.ownKeys(obj);

  for (const propKey of propKeys) {
    // `yield` returns a value and then pauses
    // the generator. Later, execution continues
    // where it was previously paused.
    yield [propKey, obj[propKey]];
  }
}
```

objectEntries() is used like this:

```
const jane = { first: 'Jane', last: 'Doe' };
for (const [key,value] of objectEntries(jane)) {
  console.log(` ${key}: ${value}`);
}
// Output:
// first: Jane
// last: Doe
```

How exactly objectEntries() works is explained in [a dedicated section](#). Implementing the same functionality without generators is much more work.

30.19.2 Blocking on asynchronous function calls

In the following code, I use [the control flow library co](#) to asynchronously retrieve two JSON files. Note how, in line A, execution blocks (waits) via yield until the result of

`Promise.all()` is ready. That means that the code looks synchronous while performing asynchronous operations.

```
co(function* () {
  try {
    const [croftStr, bondStr] = yield Promise.all([ // (A)
      getFile('http://localhost:8000/croft.json'),
      getFile('http://localhost:8000/bond.json'),
    ]);
    const croftJson = JSON.parse(croftStr);
    const bondJson = JSON.parse(bondStr);

    console.log(croftJson);
    console.log(bondJson);
  } catch (e) {
    console.log('Failure to read: ' + e);
  }
});
```

The Promise-based function `getFile(url)` retrieves the file pointed to by `url`. Its implementation is shown later. I'll also explain how `co` works. For more info on Promises, consult Chap. “[Promises for asynchronous programming](#)”.

30.20 New regular expression features

The following regular expression features are new in ECMAScript 6:

- The new flag `/y` (sticky) anchors each match of a regular expression to the end of the previous match.
- The new flag `/u` (unicode) handles surrogate pairs (such as `\uD83D\uDE80`) as code points and lets you use Unicode code point escapes (such as `\u{1F680}`) in regular expressions.
- The new data property `flags` gives you access to the flags of a regular expression, just like `source` already gives you access to the pattern in ES5:

```
> /abc/ig.source // ES5
'abc'
> /abc/ig.flags // ES6
'gi'
```

- You can use the constructor `RegExp()` to make a copy of a regular expression:

```
> new RegExp(/abc/ig).flags
'gi'
> new RegExp(/abc/ig, 'i').flags // change flags
'i'
```

30.21 Promises for asynchronous programming

Promises are an alternative to callbacks for delivering the results of an asynchronous computation. They require more effort from implementors of asynchronous functions, but provide several benefits for users of those functions.

The following function returns a result asynchronously, via a Promise:

```
function asyncFunc() {
  return new Promise(
    function (resolve, reject) {
      ...
      resolve(value); // success
      ...
      reject(error); // failure
    }
  );
}
```

You call `asyncFunc()` as follows:

```
asyncFunc()
  .then(value => { /* success */ })
  .catch(error => { /* failure */ });
```

30.21.1 Chaining `then()` calls

`then()` always returns a Promise, which enables you to chain method calls:

```
asyncFunc1()
  .then(value1 => {
    // Success: use value1
    return asyncFunc2(); // (A)
  })
  .then(value2 => { // (B)
    // Success: use value2
  })
  .catch(error => {
    // Failure: handle errors of
    // asyncFunc1() and asyncFunc2()
  });
```

How the Promise `P` returned by `then()` is settled depends on what its callback does:

- If it returns a Promise (as in line A), the settlement of that Promise is forwarded to `P`. That's why the callback from line B can pick up the settlement of `asyncFunc2`'s Promise.
- If it returns a different value, that value is used to settle `P`.
- If it throws an exception then `P` is rejected with that exception.

Furthermore, note how `catch()` handles the errors of two asynchronous function calls (`asyncFunction1()` and `asyncFunction2()`). That is, uncaught errors are passed on until there is an error handler.

30.21.2 Executing asynchronous functions in parallel

If you chain asynchronous function calls via `then()`, they are executed sequentially, one at a time:

```
asyncFunc1()
  .then(() => asyncFunc2());
```

If you don't do that and call all of them immediately, they are basically executed in parallel (a *fork* in Unix process terminology):

```
asyncFunc1();
asyncFunc2();
```

`Promise.all()` enables you to be notified once all results are in (a *join* in Unix process terminology). Its input is an Array of Promises, its output a single Promise that is fulfilled with an Array of the results.

```
Promise.all([
  asyncFunc1(),
  asyncFunc2(),
])
  .then(([result1, result2]) => {
    ...
  })
  .catch(err => {
    // Receives first rejection among the Promises
    ...
  });
```

30.21.3 Glossary: Promises

The Promise API is about delivering results asynchronously. A *Promise object* (short: Promise) is a stand-in for the result, which is delivered via that object.

States:

- A Promise is always in one of three mutually exclusive states:
 - Before the result is ready, the Promise is *pending*.
 - If a result is available, the Promise is *fulfilled*.
 - If an error happened, the Promise is *rejected*.
- A Promise is *settled* if “things are done” (if it is either fulfilled or rejected).
- A Promise is settled exactly once and then remains unchanged.

Reacting to state changes:

- *Promise reactions* are callbacks that you register with the Promise method `then()`, to be notified of a fulfillment or a rejection.
- A *thenable* is an object that has a Promise-style `then()` method. Whenever the API is only interested in being notified of settlements, it only demands thenables (e.g. the values returned from `then()` and `catch()`; or the values handed to `Promise.all()` and `Promise.race()`).

Changing states: There are two operations for changing the state of a Promise. After you have invoked either one of them once, further invocations have no effect.

- *Rejecting* a Promise means that the Promise becomes rejected.
- *Resolving* a Promise has different effects, depending on what value you are resolving with:
 - Resolving with a normal (non-thenable) value fulfills the Promise.
 - Resolving a Promise P with a thenable T means that P can't be resolved anymore and will now follow T's state, including its fulfillment or rejection value. The appropriate P reactions will get called once T settles (or are called immediately if T is already settled).

30.22 Metaprogramming with proxies

Proxies enable you to intercept and customize operations performed on objects (such as getting properties). They are a *metaprogramming* feature.

In the following example, `proxy` is the object whose operations we are intercepting and `handler` is the object that handles the interceptions. In this case, we are only intercepting a single operation, `get` (getting properties).

```
const target = {};
const handler = {
  get(target, propKey, receiver) {
    console.log('get ' + propKey);
    return 123;
  }
};
const proxy = new Proxy(target, handler);
```

When we get the property `proxy.foo`, the handler intercepts that operation:

```
> proxy.foo  
get foo  
123
```

Consult [the reference for the complete API](#) for a list of operations that can be intercepted.

Notes

Footnotes:

Background

1 This is not completely true: there are a few minor breaking changes that don't affect code on the web. These are detailed in [section D.1](#) and [section E.1](#) of the ES6 specification.↵

2 Source: Tweet by Allen Wirfs-Brock.
<https://twitter.com/awbjs/status/574649464687734785>↵

Data

1 [Speaking JS] `parseFloat()` in [“Speaking JavaScript”](#).↵

2 [Speaking JS] `parseInt()` in [“Speaking JavaScript”](#).↵

3 [Speaking JS] The details of rounding errors are explained in [“Speaking JavaScript”](#).↵

4 Internally, JavaScript has [two zeros](#). `Math.sign(-0)` produces the result `-0` and `Math.sign(+0)` produces the result `+0`.↵

5 [Speaking JS] Details are explained in [the chapter on JSON](#).↵

6 [Speaking JS] The details are explained in [the chapter on JSON](#).↵

7 [Speaking JS]
http://speakingjs.com/es5/ch23.html#_dynamically_evaluating_javascript_code_via_eval_and_new_function↵

8 Iterables are explained in [another chapter](#).↵

9 Explained in [the chapter on Arrays](#).↵

Modularity

1 The exceptions are function expressions and object literals, which you have to put in parentheses, because they look like function declarations and code blocks.↵

2 [Spec] Sect. [“Imports”](#) starts with grammar rules and continues with semantics.↵

3 [Spec] The specification method `GetExportedNames()` collects the exports of a module. In step (7.d.i), a check prevents other modules' default exports from being re-exported.↵

4 [Spec] Sect. [“Exports”](#) starts with grammar rules and continues with semantics.↵

Collections

1 [Speaking JS] [“Pitfalls: Using an Object as a Map”](#)↵

2 Based on [“Closing iterators”](#), slides by David Herman.↵

3 `throw()` is also an optional method, but is practically never used for iterators and therefore explained in [the chapter on generators](#)↩

4 “[Combinator](#)” (in HaskellWiki) describes what combinators are.↩

5 Or rather, the function counts up until the number `start` overflows and becomes `Infinity`, at which point it doesn't change, anymore.↩