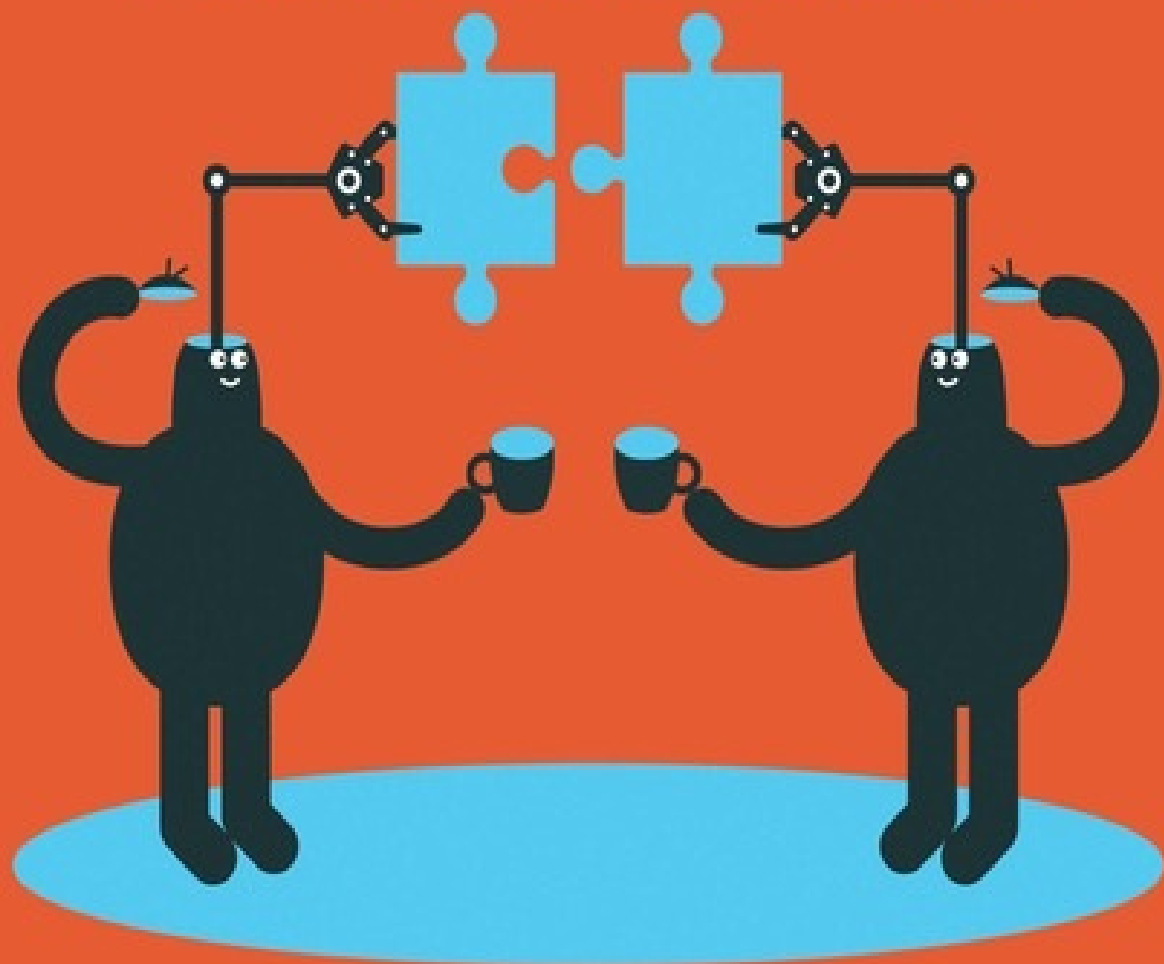


GET PROGRAMMING WITH JAVASCRIPT NEXT

New features of ECMAScript 2015, 2016, and beyond



JD Isaacks



MANNING

Get Programming with JavaScript Next: New features of ECMAScript 2015, 2016, and beyond

JD Isaacks



Copyright

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

©2018 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

♻️ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editors: Candace West, Dan Maharry
Technical development editor: Nick Watts
Review editor: Aleksandar Dragosavljević
Project editor: David Novak
Copy editor: Benjamin Berg
Proofreader: Melody Dolab
Technical proofreader: Jon Borgman
Typesetter: Dottie Marsico
Cover designer: Monica Kamsvaag

ISBN 9781617294204

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 – EBM – 23 22 21 20 19 18

DEDICATION

For my brother, Jon Potts

Brief Table of Contents

Copyright

Brief Table of Contents

Table of Contents

Preface

Acknowledgments

About this book

About the author

Lesson 1. ECMAScript specification and the proposal process

Lesson 2. Transpiling with Babel

Lesson 3. Bundling modules with Browserify

Unit 1. Variables and strings

Lesson 4. Declaring variables with let

Lesson 5. Declaring constants with const

Lesson 6. New string methods

Lesson 7. Template literals

Lesson 8. Capstone: Building a domain-specific language

Unit 2. Objects and arrays

Lesson 9. New array methods

Lesson 10. Object.assign

Lesson 11. Destructuring

Lesson 12. New object literal syntax

Lesson 13. Symbol—a new primitive

Lesson 14. Capstone: Simulating a lock and key

Unit 3. Functions

Lesson 15. Default parameters and rest

Lesson 16. Destructuring parameters

Lesson 17. Arrow functions

Lesson 18. Generator functions

Lesson 19. Capstone: The prisoner's dilemma

Unit 4. Modules

Lesson 20. Creating modules

Lesson 21. Using modules

Lesson 22. Capstone: Hangman game

Unit 5. Iterables

Lesson 23. Iterables

Lesson 24. Sets

Lesson 25. Maps

Lesson 26. Capstone: Blackjack

Unit 6. Classes

Lesson 27. Classes

Lesson 28. Extending classes

Lesson 29. Capstone: Comets

Unit 7. Working asynchronously

Lesson 30. Promises

[Lesson 31. Advanced promises](#)

[Lesson 32. Async functions](#)

[Lesson 33. Observables](#)

[Lesson 34. Capstone: Canvas image gallery](#)

[Exercise answers](#)

[Here's a preview of some of the new syntaxes you'll learn
in unit 2](#)

[Here's a preview of using promises and async functions
from unit 7](#)

[Index](#)

[List of Figures](#)

[List of Tables](#)

[List of Listings](#)

Table of Contents

Copyright

Brief Table of Contents

Table of Contents

Preface

Acknowledgments

About this book

About the author

Lesson 1. ECMAScript specification and the proposal process

1.1. A brief history of ECMAScript

1.2. Why ES2015 adds so much

1.3. Who decides what gets added?

1.3.1. Specification stages

1.3.2. Picking a stage

1.4. What this book will cover

Summary

Lesson 2. Transpiling with Babel

2.1. What is transpiling?

2.1.1. Compiling to JavaScript languages

2.1.2. Where Babel fits in

2.2. Setting up Babel 6

2.3. The Babel configuration needed for this book

2.3.1. A note on source maps

2.3.2. Set up Babel as NPM script

Summary

Lesson 3. Bundling modules with Browserify

3.1. What's a module?

3.2. How modules work in Node.js

3.3. What is Browserify?

3.4. How does Browserify help with ES6 modules?

3.5. Setting up Browserify with Babel

3.5.1. Installing Browserify

3.5.2. Setting up a project using babelify

3.6. Alternatives to Browserify

Summary

Unit 1. Variables and strings

Lesson 4. Declaring variables with let

4.1. How scope works with let

4.1.1. Why the block scope of let is preferred

4.2. How hoisting works with let

4.3. Should I use let instead of var from now on?

Summary

Lesson 5. Declaring constants with const

5.1. How constants work

5.2. When to use constants

Summary

Lesson 6. New string methods

6.1. Searching strings

6.2. Padding strings

Summary

Lesson 7. Template literals

7.1. What are template literals?

7.1.1. String interpolation with template literals

7.1.2. Multiline strings with template literals

7.2. Template literals are not reusable templates

7.3. Custom processing with tagged template literals

Summary

Lesson 8. Capstone: Building a domain-specific language

8.1. Creating some helper functions

8.2. Create an HTML-escaping DSL

8.3. Create a DSL for converting arrays into HTML

Summary

Unit 2. Objects and arrays

Lesson 9. New array methods

9.1. Constructing arrays with Array.from

9.2. Constructing arrays with Array.of

9.3. Constructing Arrays with Array.prototype.fill

9.4. Searching in arrays with

Array.prototype.includes

9.5. Searching in arrays with Array.prototype.find

Summary

Lesson 10. Object.assign

10.1. Setting default values with Object.assign

10.2. Extending objects with Object.assign

10.3. Preventing mutations when using
Object.assign

10.4. How Object.assign assigns values

Summary

Lesson 11. Destructuring

11.1. Destructuring objects

11.2. Destructuring arrays

11.3. Combining array and object destructuring

11.4. What types can be destructured

Summary

Lesson 12. New object literal syntax

12.1. Shorthand property names

12.2. Shorthand method names

12.3. Computed property names

Summary

Lesson 13. Symbol—a new primitive

13.1. Using symbols as constants

13.2. Using symbols as object keys

13.3. Creating behavior hooks with global symbols

13.4. Modifying object behavior with well-known symbols

13.5. Symbol gotchas

Summary

Lesson 14. Capstone: Simulating a lock and key

14.1. Creating the lock and key system

14.2. Creating a Choose the Door game

Summary

Unit 3. Functions

Lesson 15. Default parameters and rest

15.1. Default parameters

15.2. Using default params to skip recalculating values

15.3. Gathering parameters with the rest operator

15.4. Using rest to pass arguments between functions

Summary

Lesson 16. Destructuring parameters

16.1. Destructuring array parameters

16.2. Destructuring object parameters

16.3. Simulating named parameters

16.4. Creating aliased parameters

Summary

Lesson 17. Arrow functions

17.1. Succinct code with arrow functions

17.2. Maintaining context with arrow functions

17.3. Arrow function gotchas

Summary

Lesson 18. Generator functions

18.1. Defining generator functions

18.2. Using generator functions

18.3. Creating infinite lists with generator functions

Summary

Lesson 19. Capstone: The prisoner's dilemma

19.1. Generating prisoners

19.2. Getting prisoners to interact

19.3. Getting and storing the results

19.4. Putting the simulation together

19.5. Which prisoner does best?

Summary

Unit 4. Modules

Lesson 20. Creating modules

20.1. Module rules

20.2. Creating modules

20.3. When does a JavaScript file become a module?

Summary

Lesson 21. Using modules

21.1. Specifying a module's location

21.2. Importing values from modules

21.3. How imported values are bound

21.4. Importing side effects

21.5. Breaking apart and organizing modules

Summary

Lesson 22. Capstone: Hangman game

22.1. Planning

22.2. The words module

22.3. The status module

22.4. The game's interface modules

22.5. The index

Summary

Unit 5. Iterables

Lesson 23. Iterables

23.1. Iterables—what are they?

23.2. The for..of statement

23.3. Spread

23.3.1. Using spread as an immutable push

23.4. Iterators—looking under the hood of iterables

Summary

Lesson 24. Sets

24.1. Creating sets

24.2. Using sets

24.3. What about the WeakSet?

Summary

Lesson 25. Maps

25.1. Creating maps

25.2. Using maps

25.3. When to use maps

25.4. What about the WeakMap?

Summary

Lesson 26. Capstone: Blackjack

26.1. The cards and the deck

26.2. Making the CPU's turn slow enough to see

26.3. Putting the pieces together

Summary

Unit 6. Classes

Lesson 27. Classes

27.1. Class declarations

27.2. Instantiating classes

27.3. Exporting classes

27.4. Class methods are not bound

27.5. Setting instance properties in class definitions

27.6. Static properties

Summary

Lesson 28. Extending classes

28.1. Extends

28.2. Super

28.3. A common gotcha when extending classes

Summary

Lesson 29. Capstone: Comets

29.1. Creating a controllable sprite

29.2. Adding comets

29.3. Shooting rockets

29.4. When things collide

29.5. Adding explosions

Summary

Unit 7. Working asynchronously

Lesson 30. Promises

30.1. Using promises

30.2. Error handling

30.3. Promise helpers

Summary

Lesson 31. Advanced promises

31.1. Creating promises

31.2. Nested promises

31.3. Catching errors

Summary

Lesson 32. Async functions

32.1. Asynchronous code with generators

32.2. Async functions

32.3. Error handling in async functions

Summary

Lesson 33. Observables

33.1. Creating observables

33.2. Composing observables

33.3. Creating observable combinators

Summary

Lesson 34. Capstone: Canvas image gallery

34.1. Fetching images

34.2. Painting the images on the canvas

34.3. Repeating the process

Summary

Exercise answers

Lesson 4

Lesson 5

Lesson 6

Lesson 7

Lesson 9

Lesson 10

Lesson 11

Lesson 12

Lesson 13

[Lesson 15](#)

[Lesson 16](#)

[Lesson 17](#)

[Lesson 18](#)

[Lesson 20](#)

[Lesson 21](#)

[Lesson 23](#)

[Lesson 24](#)

[Lesson 25](#)

[Lesson 27](#)

[Lesson 28](#)

[Lesson 30](#)

[Lesson 31](#)

[Lesson 32](#)

[Lesson 33](#)

[Here's a preview of some of the new syntaxes you'll learn
in unit 2](#)

[Here's a preview of using promises and async functions
from unit 7](#)

[Index](#)

[List of Figures](#)

[List of Tables](#)

[List of Listings](#)

Preface

I've been using ECMAScript-based languages for about 15 years. In fact, the first programming language I ever learned, ActionScript, was based on ECMAScript. I kind of fell into programming, but I'm so glad that I did. Originally I wanted to be a graphic designer. I was always drawing, and by middle school, I was making complex drawings pixel by pixel using Microsoft Paint. In high school, I took a class on interactive multimedia, and was introduced to Adobe Photoshop and Macromedia Flash. Once I discovered the power of Photoshop, I never wanted to go back and draw pixel by pixel ever again. With Flash, I could take it even further: no longer confined to creating still images, I could create rich animations.

This was the time of eBaum's World and Newgrounds— websites that showcased the communities' Flash games. I would visit these sites and wonder how these games were created. I tried to teach myself ActionScript (Flash's internal language) purely by experimentation, but it wasn't until I bought my first programming book that I really learned. After that, I was hooked. Being able to add interaction was a whole new dimension over creating animations, just as creating animations was a whole new dimension over creating still images. But this was just the start of my journey.

Sometimes in my career, I learn something new that's so much more powerful than how I've been doing things that I never want to go back. Most recently, after learning the power of all the new features packed into ES2015 and later, I never want to go back to how I did things before—

and I hope you agree.

Acknowledgments

I would like to thank my wife Christina and our children, Talan and Jonathan “Jam,” for all the time they sacrificed spending with me as I worked hard on writing this book. I love you all.

I would also like to thank Manning, especially my editors Dan Maharry, Nick Watts, and Candace West. I would like to extend a special thank you to all the reviewers that made this book even better: Aïmen Saïhi, Ali Naqvi, Brian Norquist, Casey Childers, Ethien Daniel Salinas Domínguez, Fasih Khatib, Francesco Strazzullo, Giancarlo Massari, Laurence Giglio, Matteo Gildone, Michael J. Haller, Michael Jensen, Miguel Paraz, Pierfrancesco D’Orsogna, Richard Ward, Sean Lindsay, and Ticean Bennett.

About this book

Get Programming with JavaScript Next was written for JavaScript programmers looking to learn the modern features introduced in 2015 and later. Instead of focusing on a specific version such as ES2015 or ES2016, I wanted to focus on the best new features that a developer would run into and be expected to understand when thrown into a modern JavaScript development environment.

WHO SHOULD READ THIS BOOK

Any programmer, no matter their skill level, should get a lot out of this book. This book doesn't teach "how to program." Readers are expected to be somewhat comfortable programming with classic JavaScript, but you don't need to be an expert in JavaScript to follow along.

HOW THIS BOOK IS ORGANIZED

This book is broken into cohesive units. Each unit follows a specific theme such as functions or asynchronous coding. Each unit is broken into lessons on a specific topic, and each lesson starts with a priming question designed to get your gears spinning in the right way before we start the lesson. Throughout each lesson, there will be quick checks to make sure you understand the core idea of a section before moving on. At the end of each lesson is an exercise to help you take what you've learned and apply it. At the end of each unit is a capstone project that you'll build using everything you've learned throughout the unit.

ABOUT THE CODE

This book contains many examples of source code both in numbered listings and inline with normal text. In both cases, source code is formatted in a `fixed-width font like this` to separate it from ordinary text. Sometimes code is also in bold to highlight code that has changed from previous steps in the chapter, such as when a new feature adds to an existing line of code.

In many cases, the original source code has been reformatted; we've added line breaks and reworked indentation to accommodate the available page space in the book. In rare cases, even this was not enough, and listings include line-continuation markers (`\`). Additionally, comments in the source code have often been removed from the listings when the code is described in the text. Code annotations accompany many of the listings, highlighting important concepts.

The code used in this book may be accessed at the publisher's website (<https://www.manning.com/books/get-programming-with-javascript-next>) or GitHub (<https://github.com/jisaacks/get-programming-jsnext>).

BOOK FORUM

Purchase of *Get Programming with JavaScript Next* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum, go to <https://forums.manning.com/forums/get-programming-with-javascript-next>. You can also learn more about Manning's forums and the rules of conduct at <https://forums.manning.com/forums/about>.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions, lest his interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

ONLINE RESOURCES

You can keep up to date with the features of JavaScript at <https://github.com/tc39/ecma262>. Here you can see what features are at what stage in the proposal process. We talk about how these stages and proposals work later in this book.

About the author



JD ISAACKS has been programming for 15 years; his focus has primarily been on ECMAScript-based languages. He was previously a JavaScript instructor for The Iron Yard coding academy. He loves open source and has contributed to many popular projects such as React, Backbone, and D3. He is also a member of both the Bower.js and Moment.js teams, and the creator of GitGutter, one of the most popular Sublime Text packages.

Lesson 1. ECMAScript specification and the proposal process

In this lesson you will learn about the origins of JavaScript and the difference between JavaScript and ECMAScript. Since this book is about the new features introduced starting with ES2015 and later, in this lesson you'll learn about how those new features get proposed for the language and the process of how those proposals become part of the language specification.

1.1. A BRIEF HISTORY OF ECMASCRIPT

JavaScript was originally created at Netscape in 1995. Later the language was submitted to Ecma International for standardization, and ECMAScript First Edition was published in 1997. Ecma International used to be known as The European Computer Manufacturers Association (ECMA), but changed its name to “Ecma International” to reflect its global position. Even though Ecma is no longer an acronym, ECMAScript still uses the uppercase ECMA. After ECMAScript was published, the following two years each saw an updated addition, with ECMAScript Third Edition, commonly referred to as *ES3*, being published in December of 1999.

ECMAScript Fourth Edition (ES4) was intended to be a radical change. It introduced many new concepts, including classes and interfaces, and was statically typed. It also wasn’t backwards-compatible with ES3. This meant that if implemented, it had the potential to break existing JavaScript applications in the wild. Needless to say, ES4 was controversial and split the Ecma technical committee, resulting in a subcommittee formed to work on a much smaller update dubbed *ECMAScript 3.1*. ECMAScript Fourth Edition was eventually abandoned and ECMAScript 3.1 was renamed to *ES5 (Fifth Edition)* and published in 2009.

If you’ve been keeping score, that was 10 years, a full decade before a new version of the language was published—and even though the version number jumped by two, the update was really rather small.

1.2. WHY ES2015 ADDS SO MUCH

ECMAScript Sixth Edition was finalized in June of 2015. The Sixth Edition was the first major update to the language in over 15 years. The landscape of the web and the way websites and web applications were built had changed drastically, so naturally there were many new ideas, leading to new syntax, operators, primitives, and objects with enhancements to existing ones, as well as a treasure trove of new concepts. All of this equates to the Sixth Edition being a major revision.

Initially (and commonly still) referred to as *ES6*, the Sixth Edition was renamed to *ES2015* to coincide with the original strategy of releasing a new version each year. As such, *ECMAScript Seventh Edition*, originally dubbed *ES7* and then renamed *ES2016*, was finalized in June 2016.

The idea behind releasing a new version each year is that the language can gradually and continuously mature without going through a stagnant phase like it did in the early 2000s. This should also make new editions easier and faster to adopt for developers.

1.3. WHO DECIDES WHAT GETS ADDED?

A task group within Ecma International known as *TC39* (<http://www.ecma-international.org/memento/TC39.htm>) is responsible for developing and maintaining the ECMAScript specification. The members of the group are mostly from companies that build web browsers, such as Mozilla, Google, Microsoft, and Apple, as they have a vested interest in and must implement, the specification. You can see a full list of the TC39 members at <http://tc39wiki.calculist.org/about/people/>. Additions to the ECMAScript specification go through a five-stage process ranging from stages 0 through 4.

1.3.1. Specification stages

- *Stage 0: Strawman*—This stage is informal and can be in any form; it allows anyone to add their input into the further development of the language. To add your input, you must be either a member of TC39 or registered with Ecma International. You can register at <https://tc39.github.io/agreements/contributor/>. Once registered you can propose your idea via the esdiscuss mailing list. You can also view these discussions at <https://esdiscuss.org/>.
- *Stage 1: Proposal*—After a strawman has been made, a member of TC39 must champion the addition to advance it to the next stage. This means the TC39 member must explain why the addition is useful and describe how the addition will behave and look once implemented.
- *Stage 2: Draft*—In this stage, the addition gets fully spec'd out and is considered experimental. If the addition reaches this stage, it means the committee expects the feature to eventually make it into the language.
- *Stage 3: Candidate*—At this stage, the solution is considered complete and is signed off on. Changes after this stage are rare, and are generally for critical discoveries after implementation and significant usage. After a suitable period of deployment, the addition may be safely bumped to stage 4.

- *Stage 4: Finished*—This is the final stage; if an addition reaches this stage it is ready to be included in the formal ECMAScript standard specification.

For further reading about these specific stages and other information about the TC39 process, see <https://tc39.github.io/process-document/>.

1.3.2. Picking a stage

There are projects such as Babel (see [lesson 2](#)) that allow you to use future JavaScript features today. If you're going to use a tool like this, it's probably a good idea to pick an appropriate stage at the beginning of your project. If you only want features that are guaranteed to be in the next release, stage 4 would be appropriate. Choosing stage 3 is also considered safe because most likely any features included in stage 3 will end up staying, and with few changes. Choose a stage lower than that and you run the risk of having features changed or even revoked in the future. You may find a particular feature helpful enough to make that risk worth taking.

You can also decide what stage to choose based on what features you want to use. Of course, you may not want to use any features that aren't officially included in the ECMAScript specification yet, and that is fine. If you do want to pick a stage, you can review what features are in what stages at the following URLs:

- Stage 0—<https://github.com/tc39/ecma262/blob/master/stage0.md>
- Stages 1–3—<https://github.com/tc39/proposals/blob/master/README.md>
- Stage 4—<https://github.com/tc39/proposals/blob/master/finished-proposals.md>

1.4. WHAT THIS BOOK WILL COVER

This book is intended for existing JavaScript developers looking to get up to speed and become productive with the newest editions of JavaScript, including ES2015, ES2016, and later. This book focuses on the most important and widely used features of these editions and proposals. This book is not intended to teach JavaScript or programming fundamentals. But you don't need to be an expert JavaScript programmer to benefit from this book, either.

Because this book covers a mixture of ES2015, ES2016, and proposed/staged features, I'll define some terms to make keeping track of all this easier. Going forward, I may refer to ES2015 interchangeably with ES6, and when I do, I'm always referring to the same thing: the Sixth Edition to ECMAScript. Likewise, I may refer to ES7 and ES2016 interchangeably. I may use the term *ESNext* as a blanket term to refer to ES2015 and later—basically everything that's new to JavaScript after ES5.

SUMMARY

In this lesson you learned that ECMAScript is the official specification for JavaScript and how the proposal process works. In the next lesson you'll learn how to transpile features that aren't implemented yet, so you can use them today.

Lesson 2. Transpiling with Babel

When new features are added to JavaScript, the browsers always have to play a game of catch-up. It takes time after an update to JavaScript's specification before all modern browsers have everything fully implemented and supported. In order to use all of the features covered in this book, you'll make use of the technique covered in this lesson: transpiling.

2.1. WHAT IS TRANSPILING?

Transpile is a portmanteau of the words *translate* and *compile*. Compilers typically compile a written programming language into incomprehensible^[1] machine code. A transpiler is a special kind of compiler that translates from the source code of one programming language to the source code of another.

¹

To a human, anyway.

2.1.1. Compiling to JavaScript languages

Transpilers have been around for some time, but they exploded onto the JavaScript scene in 2009 with the introduction of *CoffeeScript* (<http://coffeescript.org>).

CoffeeScript is a compile-to-JavaScript language created by Jeremy Ashkenas, who is also known for creating the popular JavaScript libraries Underscore and Backbone. It takes many cues from Ruby, Python, and Haskell, and focuses on the “good parts” of JavaScript made popular by Douglas Crockford in his book *JavaScript: The Good Parts* (O’Reilly Media, 2008). CoffeeScript achieved this by hiding many of what users refer to as the warts of JavaScript and only exposing the safer parts.

CoffeeScript, however, is neither a subset nor a superset of JavaScript. It exposes a new syntax and many new concepts, some of which became inspiration for features in ES2015, such as arrow functions. Following CoffeeScript’s success, many other compile-to-JavaScript languages started showing up, such as ClojureScript, PureScript, TypeScript, and Elm, to name just a few.

JavaScript isn’t necessarily the best target for a language

to compile to, but in order to run code on the web, there is no other choice. Recently a new technology was announced called *WebAssembly* (often shortened to *wasm*). WebAssembly promises to be a better compile-to-target for frontend development than JavaScript, and if successful, could pave the way for even more diversity when choosing a language to run in the browser.

2.1.2. Where Babel fits in

At this point, you may be thinking, “Transpilers sound cool and all, but who cares? I’m reading a book on JavaScript, not a language that compiles to it.” Well, transpilers aren’t just used for languages that compile to JavaScript. They can also help you write ESNext code and use it in the browser today. Think about it: when another language compiles to JavaScript, it not only targets JavaScript, but a specific version of JavaScript. CoffeeScript, for example, targets ES3. So if a completely differently language can be transpiled into a specific version of JavaScript, shouldn’t another version of JavaScript be able to as well?

There are several transpilers for converting ESNext JavaScript into a suitable version that can be executed in the browser today. Two of the most commonly used are *Traceur* and *Babel*. Babel used to be called ES6to5 because it transpiled ES6 code into ES5 code, but after it started supporting all future JavaScript features, and considering that ES6’s name officially changed to ES2015, the team behind ES6to5 decided to change the project’s name to Babel.

2.2. SETTING UP BABEL 6

Babel is available as an NPM (<https://www.npmjs.com/>) package and comes bundled with Node.js (<https://nodejs.org/en/>). You can download an installer for Node.js from their website. This book assumes you have Node.js version 4 or later installed and NPM version 3 or later. NPM comes bundled with Node.js and thus doesn't require being installed separately.

In order to use Babel, you'll set up a node.js package so you can install your required dependencies. With Node.js and NPM installed, open a command line program (Terminal .app in OSX or cmd.exe in Windows) and execute the following shell commands to initialize a new project (make sure you replace the placeholder `project_name` with the name of your project):^[1]

1

The \$ at the beginning indicates that this is a shell command meant to be executed in a command line program. The \$ is not part of the actual command and should not be entered.

```
$ mkdir project_name
$ cd project_name
$ npm init -y
```

Let's break this command down. The line `mkdir project_name` will make a new directory (folder) with the name you provide. Then `cd project_name` will change to the newly created directory for the project. Finally, `npm init` will initialize this as a new project. The `-y` flag tells NPM to skip asking questions and go with all the defaults.

You should now see a new file called `package.json` in your project indicating that this is a Node.js project. Now

that you have a project initialized, you can set up Babel. Execute the following shell command to install Babel's command line interface:^[2]

2

The current version of Babel at the time of this writing is version 6.5.2. The instructions in this book are for Babel version 6.x which is a major change from version 5.x. You can constrain to some version of Babel 6 using the version range `>=6.0.0 <7.0.0` e.g. `npm install babel@">=6.0.0 <7.0.0"`; see <https://docs.npmjs.com/cli/install>.

```
$ npm install babel-cli --save-dev
```

Beginning in version 6, Babel doesn't do any transforming by default, and you must install plugins or presets for any transformations you want to apply. To use a plugin or preset, you must both install it in your project and specify its use in Babel's configuration.

Babel uses a special file called `.babelrc` for its configuration. You must put this file in the project's root and the contents of the file must be valid JSON. To specify that you want Babel to transpile all ES2015 features, you can use the ES2015 preset. Edit the `.babelrc` file so its contents are

```
{
  "presets": ["es2015"]
}
```

Now that you've told Babel to use the ES2015 preset, you must also install it:

```
$ npm install babel-preset-es2015 --save-dev
```

You should now be ready to transpile some ES6 code! Test it out. First add a new folder in your project named `src` and add a new `index.js` file in it. Your project structure should now look like this:

```
project_name
└─ src
   └─ index.js
```

Now add some ES2015 code to transpile. Add the following code to your `index.js` file:

```
let foo = "bar";
```

You can now tell Babel to transpile your source code. In your terminal, run the following command.

Listing 2.1. Compiling from `src` folder to a `dist` folder

```
$ babel src -d dist
```

After running this command, a new directory named `dist` should be created with the transpiled code inside. Let's break the command down. When you specify `babel src` you are telling Babel to operate on the contents of the `src` directory. By default, Babel outputs the transpiled code to the terminal. When you add the `-d` `<directory_name>`, you can tell Babel to output the transpiled code to a directory instead.

The structure of your project should now look like this:

```
project_name
├─ dist
│   └─ index.js
└─ src
   └─ index.js
```

The `dist/index.js` file contains the following transpiled code:

```
"use strict";

var foo = "bar";
```

2.3. THE BABEL CONFIGURATION NEEDED FOR THIS BOOK

Each of the TC39 stages has a preset. You can include a preset for any of the five stages and Babel will be able to compile code that has reached that stage (or higher). For example, if you use the stage-2 preset, you could use features that have reached stages 2, 3, or 4, but not stages 0 or 1.

Because I can't predict what stages each proposal will be at when you read this, please consult the TC39 stages links from [lesson 1](#) to determine what presets you will need.

Alternatively you can go with stage 0 to grab everything, using the following .babelrc preset.

Listing 2.2. Babel stage-0 presets

```
{
  "presets": ["es2015", "stage-0"],
  "plugins": ["transform-decorators-legacy"],
  "sourceMaps": "inline"
}
```

In [listing 2.2](#), you're using the ES2015 and stage-0 presets to include all of ES2015 and all of the existing proposed features. You also need to include the transform-decorators-legacy plugin in order to transpile decorators. And finally you tell Babel to include inline source maps to make debugging easier. Now in order for Babel to use those plugins and presets, you need to install them:^[1]

1

You can install them all at once instead of one at a time like I did. I did it this way for legibility in the small margins of a book.

```
$ npm install babel-preset-es2015 --save-dev
$ npm install babel-preset-stage-0 --save-dev
```

```
$ npm install babel-plugin-transform-decorators-legacy --save-dev
```

2.3.1. A note on source maps

In your Babel configuration you added a section for *source maps*. If you're unfamiliar with source maps, they're a technology invented to make it easier to debug minified code. Most production applications ship with their code minified to save bandwidth so that the app will load faster. This minified code can be a nightmare to debug, though, so source maps were invented to map code back to its original form. Compile-to-JavaScript languages started using source maps to show the original language's source instead of the transpiled JavaScript, and Babel does as well. To learn more about source maps, see

<http://www.html5rocks.com/en/tutorials/developertools/source-maps/>.

2.3.2. Set up Babel as NPM script

You may not want to repeatedly tell Babel what folder to transpile from and to (like you did in [listing 2.1](#)). You can make your life easier by setting up an NPM script to do that for you. If you're unfamiliar with how NPM scripts work, it's really simple. In your NPM configuration file named `package.json`, there's a special `scripts` section that allows you to specify shell commands that can be executed by their name. See

<https://docs.npmjs.com/misc/scripts> for further information about NPM scripts.

There should already be a test script added to your `package.json` by default. If you open your `package.json` file and locate the `scripts` section, it should look like this:

```
{  
  "scripts": {  
    "test": "echo \"Error: no test specified\"; exit 1"
```

Depending on your operating system, it could look different.

```
"scripts": {  
  "test": "echo \"Error: no test specified\" && exit 1"  
},
```

You can add our Babel command as a script like so:

```
"scripts": {  
  "test": "echo \"Error: no test specified\" && exit 1",  
  "babel": "babel src -d dist",  
},
```

Don't forget to add the comma after the test command!
You should now be able to execute this NPM script by
executing the following shell command:

```
$ npm run babel
```

This is not only simpler and easier to remember, but you
can also modify your Babel script as your needs evolve,
and your command always remains the same.

SUMMARY

In this lesson you learned what transpiling is and how it can be used to start using the ESNext features. You also learned how set up Babel to transpile the code in this book.

Lesson 3. Bundling modules with Browserify

Modules are a big part of the additions to JavaScript. As you'll learn in this lesson, transpiling alone is not enough for modules. This is because the most defining aspect of modules is that they break your code into separate files. This means that you'll need to *bundle* them into one file. There are several popular tools for bundling JavaScript modules; two popular up-and-coming options are Webpack and Rollup. In this lesson you'll use one of the first ones that hit the scene, Browserify.

3.1. WHAT'S A MODULE?

Many programming languages support modularized code. Ruby calls these pieces of modularized code *gems*, Python calls them *eggs*, and Java calls them *packages*. JavaScript never had official support for this concept until ES2015 with the introduction of *modules*. A module is an individual file of modularized JavaScript code. Because JavaScript was so late to this party, many community solutions for modules in JavaScript were created, the most ubiquitous being node.js modules.

3.2. HOW MODULES WORK IN NODE.JS

Node.js has a fantastic module system with NPM. NPM is the Node package manager that comes bundled with Node.js. With roughly a quarter of a million packages published to NPM's registry and billions of downloads a month, NPM is one of the richest ecosystems of code in the world.

There is a specific way to import and export modules in Node.js. When referring to Node.js-style modules, many people use the term *CommonJS*, which is a specification originally called ServerJS that was created so that many server-side JavaScript implementations could all share a compatible module definition. Only one server-side JavaScript implementation took off—Node.js—so it wasn't necessary to standardize their module definition. So while Node's module definition is similar to and often called CommonJS, strictly speaking it is not.

The module system in Node.js allows developers to separate their programs into modules that encapsulate logic and only expose necessary APIs. Because a module only exposes what is explicitly exported, there is no need to wrap everything in an immediately invoked function expression. Better yet, because the modules are only available where imported, they do not pollute the global namespace, protecting against accidental naming collisions.

3.3. WHAT IS BROWSERIFY?

Browserify is a tool that lets you define modules, the same way Node.js does during development, and then bundle them into a single file. Browserify operates on an entry point, your main JavaScript file, and analyzes what scripts are imported. It then runs on all those scripts as well, eventually building a tree of all the dependencies that are needed. Browserify then generates a single JavaScript file with all the required modules bundled into it while maintaining their proper scoping and namespacing. This bundled JavaScript file can then be included in a webpage on the frontend. This allows frontend developers to write modularized JavaScript and even utilize the rich ecosystem of all the packages published to NPM. Browserify was named after its ability to write code in a Node.js fashion and use Node.js modules in the browser.

3.4. HOW DOES BROWSERIFY HELP WITH ES6 MODULES?

You learned in [lesson 2](#) that Babel transpiles your ESNext code so you can execute it in a browser. But Babel doesn't supply you with a module system. It simply translates your ESNext source to an ES5 destination and leaves the developer with the task of solving the bundling. ES2015, on the other hand, does define a formal module specification for JavaScript. So how can you use ES2015 modules today if Babel doesn't expose a module loader? What if you could get Browserify and Babel to work together so that Babel could transpile the ES2015 modules into the kind of modules that Browserify likes to work with, and then Browserify could take it from there? Luckily Browserify has the concept of *transforms*. Transforms allow the code to be transformed before being operated on by Browserify. There is a transform for Babel called *babelify*. When you use *babelify*, each file will be transpiled before being sent to Browserify, allowing you to use ES2015 modules.

3.5. SETTING UP BROWSERIFY WITH BABEL

Now that you understand what Browserify is and the role it plays, let's install it.

3.5.1. Installing Browserify

First globally install Browserify. Execute the following shell command:

```
$ npm install browserify --global
```

This installs Browserify globally, so it can be used for any project. You won't need to install it again unless you want to upgrade to a newer version.

3.5.2. Setting up a project using babelify

Now that you have Browserify, you can get started by creating a new project called `babelify_example`. Create a new folder called `babelify_example` containing a `.babelrc` file, a `dist` folder, and a `src` folder, with the `src` folder containing an `index.js` and an `app.js` so that your project structure looks like this:

```
babelify_example
├── dist
├── src
│   ├── app.js
│   └── index.js
└── .babelrc
```

Now in your terminal, `cd` (change directory) to your project's root folder, initialize as an NPM project with `babelify` and the other Babel presets and plugins you used in the previous chapter:

```
$ cd babelify_example
$ npm init -y
$ npm install babelify --save-dev
$ npm install babel-preset-es2015 --save-dev
$ npm install babel-preset-stage-0 --save-dev
$ npm install babel-plugin-transform-decorators-legacy --save-dev
```

Notice that you did not install the babel-cli package. This is because you're now using Browserify with babelify and thus no longer need the Babel CLI (command line interface). Go ahead and add the same Babel config from your previous lesson to the .babelrc file:

```
{
  "presets": ["es2015", "stage-0"],
  "plugins": ["transform-decorators-legacy"],
  "sourceMaps": "inline"
}
```

OK, you should now be all set to start using Browserify with babelify! Test it out by writing a small module to check for you. In the app.js file, add the following code:

```
const MyModule = {
  check() {
    console.log('Yahoo! modules are working!!!');
  }
}

export default MyModule;
```

Now in the index.js file, add the following code:

```
import MyModule from './app';

MyModule.check();
```

Fantastic. Now bundle this with the following shell command:

```
$ browserify src/index.js --transform babelify --outfile
dist/bundle.js
➡ --debug
```

If everything was set up correctly, your `dist` folder should now contain a new `bundle.js` file with some transpiled JavaScript code inside of it. Let's break down that command. The first argument to Browserify is `src/index.js`. This tells Browserify that this is the *entry point* of your application—entry point meaning the root JavaScript file that imports other modules. Then the `--transform babelify` tells Browserify to use the `babelify` transform to transpile your code before bundling. The `--outfile dist/bundle.js` specifies what the destination or *output file* is for your bundled and transpiled source code. Finally, the `--debug` flag is necessary to include source maps, which wouldn't be included without it.

You can see a list of the available arguments to Browserify by running

```
$ browserify help
```

Now test your code with Node.js. If you've never used Node.js to execute JavaScript before, don't fret. You already have it installed, and telling it to execute JavaScript is as simple as pointing it to a JavaScript file. So tell Node to execute your transpiled `bundle.js` file by executing the following shell script:

```
$ node dist/bundle.js
```

You should be greeted with an enthusiastic

```
Yahoo! modules are working!!
```

Don't worry about understanding the semantics of how modules work just yet. We will cover that in [lessons 20](#)

and 21. For now, get your bundle working in the browser instead of Node. Create an `index.html` file at the root of your project with the following contents:

```
<!DOCTYPE html>
<html>
<head>
  <title>Babelify Example</title>
</head>
<body>
  <h1>Hello, ES6!</h1>

  <script src="dist/bundle.js"></script>
</body>
</html>
```

Now open your `index.html` in your web browser. Check the console; if using Google Chrome, select Menu > More Tools > Developer Tools, then select the Console tab. You should see the same quote in your console: *Yahoo! modules are working!!*

Let's recap what you've done so far:

1. You created a module with a `check` method to log a message in `app.js`.
2. In your `index.js`, you imported the module and invoked the `check` method.
3. You used `Browserify` and `Babel` to transpile and bundle your JavaScript.
4. You then included your bundled code in an HTML page and saw that it works!

That encompasses all the steps necessary to execute all the code in this book. The remaining chapters will assume that you are set up and able to execute your examples.

Don't forget to recompile (via executing the `browserify` command) whenever you make a change to your source files so that your `bundle.js` reflects your latest code. (Look into `watchify` for automatic bundling.)

You can add the Browserify shell command as an NPM script in your package.json to make it easier to run.

3.6. ALTERNATIVES TO BROWSERIFY

There are plenty of other ways to transpile and bundle your ESNext code. Webpack and Rollup are currently very popular options. Which one works best for your project will depend a lot on the details of your project. Babel has good setup examples for different scenarios that you can check out here: <http://babeljs.io/docs/setup>.

SUMMARY

In this lesson, you learned how to set up Browserify to bundle your ES2015 modules. For more information on modules, see [lessons 20–21](#).

Unit 1. Variables and strings

One of the most familiar statements in JavaScript is the `var` statement. But the addition of `let` and `const` means that `var` will soon be used much less. The `var` statement isn't going away, and can still be used, but most programmers will soon opt to use `const` to declare variables anytime they don't need to be reassigned and `let` anytime they do. In the first two lessons in this unit, you'll discover why that is.

The following two lessons will cover new string methods and a new type of string called a *template*. Templates are handy and will make the old, tedious task of concatenating large strings at author-time (at the time the code is being written) a thing of the past. Templates also have a less-known feature called *tagged templates*, which allows for custom processing and opens the door for creating domain-specific languages. You'll wrap up this unit by creating a couple of your own domain-specific languages using tagged templates.

Lesson 4. Declaring variables with let

After reading [lesson 4](#), you will

- Understand how scope works with `let` and how it differs from `var`
- Understand the difference between block scope and function scope
- Understand how `let` variables are hoisted

In the history of JavaScript, variables have always been declared using the keyword `var`.^[1] ES6 introduces two new ways to declare variables, with the `let` and `const` keywords.^[2] Both of these work slightly differently than variables declared with `var`. There are two primary differences with `let`:

1

Actually, in non-strict mode, it was possible to create a new variable while completely omitting the `var` declaration altogether. This, however, created a global, usually unbeknownst to the author, leading to some pretty fun bugs. This is why `var` is required in strict mode.

2

Technically `const`s are not variables but constants.

- `let` variables have different scoping rules.
- `let` variables behave differently when hoisted.



Consider this

Consider the following two `for` statements. The only difference is that one is using an iterator declared with `var` and the other with `let`. But the resulted outcome is much different. What do you think will happen when each one runs?

```
for (var i = 0; i < 5; i++) {  
  setTimeout(function () {  
    console.log(i);  
  }, 1);  
}
```

```
};  
for (let n = 0; n < 5; n++) {  
  setTimeout(function () {  
    console.log(n);  
  }, 1);  
};
```



4.1. HOW SCOPE WORKS WITH LET

Variables declared with `let` have *block scope*, meaning they can only be accessed within the block (or sub-blocks) they are declared in:

```
if (true) {  
  let foo = 'bar';  
}  
  
console.log(foo);           1
```

- **1 An error is thrown because `foo` does not exist outside the block it was declared in.**

This makes variables much more predictable and won't lead to bugs being introduced because the variable leaks outside of the block it is used within. A *block* is the body of a statement or function. It is the area between the opening and closing curly braces, `{` and `}`. You can even use curly braces to create a free-standing block that isn't tied to a statement:

Listing 4.1. Using a free-standing block to keep a variable private

```
let read, write;  
{  
  let data = {};  
  
  write = function (key, val) {  
    data[key] = val;  
  }  
  
  read = function (key) {  
    return data[key];  
  }  
}  
  
write('message', 'Welcome to ES6!');  
read('message');  
console.log(data);
```

- **1 Opening a free-standing block**
- **2 `data` is effectively a private variable.**

- **3 Closing a free-standing block**
- **4 “Welcome to ES6!”**
- **5 Referencing data outside of the block results in an error.**

In this example, `read` and `write` are declared outside the block, but their values are assigned inside the block that `data` is declared in. This gives them access to the `data` variable. But `data` isn't accessible outside of the block and so becomes a private variable that `write` and `read` can use to store their internal data.

Normally for a variable declared with `let` to be scoped to a specific block, the variable has to be declared within that block. There is one exception to that rule, though: in a `for` loop, a variable declared with a `let` inside the `for` loop's clause will be in the scope of the `for` loop's block:

```
for (let i = 0; i < 5; i++) {           1
  console.log(i);                      2
}
console.log(i);                        3
```

- **1 `i` is declared in the clause of the `for` statement.**
- **2 `i` is in the scope of the `for` loop's block.**
- **3 `i` cannot be referenced outside the `for` loop and an error is thrown.**

4.1.1. Why the block scope of `let` is preferred

Variables declared with `var` have *function scope*, meaning they can be accessed anywhere in their containing function:

```
(function() {
  if (true) {
    var foo = 'bar';
  }

  console.log(foo);          1
}())
```

- **1** `foo` is being referenced outside of the `if` statement it was declared in.

This has historically confused developers and led to bugs due to false assumptions. Let's take a look at a classic example in the next listing:

Listing 4.2. There is a scope problem here, but what is it?

```
<ul>
  <li>one</li>
  <li>two</li>
  <li>three</li>
  <li>four</li>
  <li>five</li>
</ul>

...
<script type="javascript">
  var items = document.querySelectorAll('li');           1
  for (var i = 0; i < 5; i++) {
    var li = items[i];
    li.addEventListener('click', function() {           2
      alert(li.textContent + ':' + i);
    });
  };
</script>
```

- **1** `document.querySelectorAll` is a standard web API method that allows selecting all the DOM nodes that match a specified query.
- **2** `addEventListener` is a method on DOM nodes that allows attaching an event listener.

This code looks like it's attaching an event listener to each list item so that when it's clicked, its text and index will be alerted. In other words, if the first list item is clicked, the expectation is that `one:0` is alerted. The reality is that no matter which list item is clicked, the alerted value will always be the same, `five:5`. That is because this code has a bug in it, a bug that has bitten many developers due to function level scoping.

Did you spot the bug? Since the variables aren't scoped to the `for` statement, each iteration is using the same

variables. Here's the breakdown of what's going on:

- The variable `i` is declared as 0.
- The first iteration of the `for` loop runs.
 - `i` is 0 and `li` is the first list item.
- The second iteration of the `for` loop runs.
 - `i` is 1 and `li` is the second list item.
- The third iteration of the `for` loop runs.
 - `i` is 2 and `li` is the third list item.
- The fourth iteration of the `for` loop runs.
 - `i` is 3 and `li` is the fourth list item.
- The fifth iteration of the `for` loop runs.
 - `i` is 4 and `li` is the fifth list item.
- `i` increments to 5 and the `for` loop stops.

This means after the `for` loop completes, `i` and `li` are set to 5 and the fifth list item, respectively. If the `for` loop alerted the text immediately, no bug would be observed; however, the alert statements are set to be alerted in an event listener which won't be triggered until after the `for` loop completes. So by the time any of the events trigger and alert `i` and `li.textContent`, you get 5 and five.

Previously you used a free-standing block to create a scope around some code to keep a variable private. Typically a scope like this is created by using an *immediately invoked function expression (IIFE)* like so:

```
(function () {  
  var foo = 'bar';  
})();  
  
console.log(foo); // ReferenceError
```

If you rewrite [listing 4.1](#) using an immediately invoked

function expression, it will look like this:

```
var read, write;

(function () {
  var data = {};

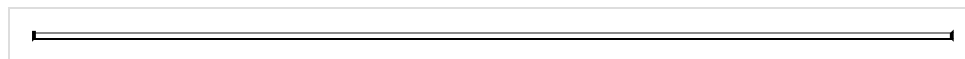
  write = function write(key, val) {
    data[key] = val;
  }

  read = function read(key) {
    return data[key];
  }
})();

write('message', 'Back in ES5 land.');// 1
read('message');// 2
console.log(data);
```

- **1 “Back in ES5 land.”**
- **2 data is out of scope so an error is thrown.**

This code is much more complex looking and harder to understand at a glance because it’s creating and invoking a function for the purpose of creating a scope. Using a function merely as a means to create a scope is a bit of overkill, but before ES6, that was the only option. With block scope, this is no longer the case and IIFEs can be replaced with free-standing blocks.



Quick check 4.1

Q1:

Given the following code, what will be logged to the console?

```
var words = ["function", "scope"];
for(var i = 0; i < words.length; i++) {
  var word = words[i];
  for(var i = 0; i < word.length; i++) {
    var char = word[i]
    console.log('char', i, char);
  }
};
```

```
};
```

QC 4.1 answer

A1:

This would cause the first word *function* to be processed but not the second word *scope*, because by the time the inner loop finishes, *i* is equal to 7, causing the outer loop to stop before processing the second word.

Technically this problem could be solved by declaring both *i* variables with `let` instead of `var`. But this is called *variable shadowing*,^[3] and is generally considered a bad practice so I would still advise using a different variable name for the inner variable.

3

See https://en.wikipedia.org/wiki/Variable_shadowing

4.2. HOW HOISTING WORKS WITH LET

Variables declared with `let` and `var` both have a behavior called *hoisting*. This means that within the entire scope of where the variable is declared, whether the entire block for `let` or the entire function for `var`, the variable consumes the entire scope. This happens no matter where in the scope the variable is declared:

```
if (condition) {  
  // ----- scope of myData starts here -----  
  doSomePrework();  
  //... more code  
  let myData = getData();  
  //... more code  
  doSomePostwork();  
  // ----- scope of myData ends here -----  
}
```

This means that in this section of code, the variable is in scope and can be accessed even before it is declared! This is true and somewhat counterintuitive if another variable by the same name exists just outside the scope. Consider this example:

```
// ----- scope of outer myData starts here -----  
let myData = getDefaultData();  
if (condition) {  
  // ----- scope of inner myData starts here -----  
  doSomePrework(myData); // <-- What variable is this???  
  //... more code  
  let myData = getData();  
  //... more code  
  doSomePostwork();  
  // ----- scope of inner myData ends here -----  
}  
// ----- scope of outer myData ends here -----
```

There are actually two `myData` variables here: one that's in scope only in the `if` statement, and another that's in the containing scope. This gets tricky because intuitively, you might think the outer-scoped `myData` with the

default value is being passed to the `doSomePrework` function because the inner variable hasn't been declared yet. But that isn't the case. Because the inner variable consumes the entire scope, it's the variable that is passed to the `doSomePrework` function. It's hoisted and used before it's even declared.

This concept of variables being in scope before they are declared, called hoisting, isn't actually new. `let` hoists to the top of the block, and `var` hoists to the top of the function. There is, however, a more important distinction here, namely what happens when a variable declared with `let` is accessed before it's declared, in contrast to `var`.

When a `let` variable is accessed in scope before it is declared, it throws a reference error. This is unlike `var`, which allows the use but the value will always be undefined. This area or *zone*, in which the `let` variable can be accessed at a time before it is declared but will throw an error if it actually is, is called a *temporal dead zone*. More specifically, a temporal dead zone refers to the area in which the variable is in scope before it's declared. Any references to variables in a temporal dead zone will throw a reference error:

```
{
  console.log(foo);      1
  let foo = 2;
}
```

- **1 Error thrown, as foo is not declared yet**

Take a moment to look over this code. What do you think will be logged when executed?

```
let num = 10;
function getNum() {
```

```
    if (!num) {  
      let num = 1;  
    }  
    return num;  
  }  
  console.log( getNum() );
```

If you think the answer is 10, you're correct. But there's something else going on here that's easy to miss. By modifying the example ever so slightly, you can see what's going on:

```
let num = 0;  
function getNum() {  
  if (!num) {  
    let num = 1;  
  }  
  return num;  
}  
console.log( getNum() );
```

What would you expect to be logged now? If you answer 1, you're incorrect. Why is this?

```
let num = 0;  
function getNum() {  
  if (!num) {  
    // ----- scope of inner num starts here ----- 1  
    let num = 1;  
    // ----- scope of inner num end here ----- 2  
  }  
  return num; 3  
}  
console.log( getNum() );
```

- **1 num is 0 so the if statement runs.**
- **2 You're declaring a new let variable with the value 1.**
- **3 Oops, when you declared a new variable as 1, it was scoped only to the if statement so this variable still remains 0.**

To fix this, remove the `let` when setting `num` to 1:

```
let num = 0;  
function getNum() {  
  if (!num) {
```

```
    num = 1;
  }
  return num;
}
console.log( getNum() );
```

Quick check 4.2

Q1:

Given the following code, what will be logged to the console?

```
{
  console.log('My lucky number is', luckNumber);
  let luckNumber = 2;
  console.log('My lucky number is', luckNumber);
}
```

QC 4.2 answer

A1:

Nothing would be logged, as the first `console.log` statement will throw an error because it attempts to access the variable before it's declared.

4.3. SHOULD I USE LET INSTEAD OF VAR FROM NOW ON?

This is a controversial question. Some developers believe yes, and some think no. I happen to be in the former camp.^[4]

4

In the case of `var`, however, as we'll see in the next lesson, `const` is often preferred to `let`.

The argument for using `var` is that if a variable is declared at the root of a function, a `var` should be used to express that this variable is scoped to the entire function. I disagree with this argument. I think this is due to developers wanting to continue to think in function scope, and needing to embrace block scope. A function is just another block as far as a `let` is concerned, and you certainly don't need a different kind of declaration if the `let` is declared inside of an `if` versus a `for`, or a `while` or any other block-level statement, so why should a function be special?

A `let` declared at the beginning of a function is scoped the same way as a `var`, but it hoists differently. A `var` will be undefined if it's accessed before it's declared; a `let` throws an exception. I think the exception is the better behavior because using a variable before it's declared leads to tricky bugs. I have also never come across a compelling reason why that is ever a good idea. This is another reason why I'm in favor of no longer using `var`.

But as I'm making the argument to completely replace `var` with `let`, be wary of doing a find and replace of

vars to lets in existing code. Making a blanket change to an existing code base can lead to errors.

SUMMARY

In this lesson, you learned how to declare variables with `let` and how that differs from variables declared with `var`:

- Variables declared with `let` use block scope.
- Block scope means the variable is only in scope within its containing block.
- Variables declared with `var` use function scope.
- Function scope means the variable is in scope within its entire containing function.
- Unlike `var` variables, `let` variables can't be referenced before they are declared.

Let's see if you got this:

Q4.1

The following code creates a function that generates an array containing a range of values. It uses `var` and several IIFEs to prevent access to variables outside of the context in which they are used:

- An all-encompassing IIFE hides `DEFAULT_START` and `DEFAULT_STEP`.
- An IIFE prevents `tmp` from escaping the `if` statement it is used in.
- Another IIFE prevents `i` from being accessed outside the `for` loop.

Rewrite this code to use `let` and remove the need for any of the IIFEs:

```
(function (namespace) {  
  
    var DEFAULT_START = 0;  
    var DEFAULT_STEP = 1;  
    var range = function (start, stop, step) {
```

```
var arr = [];  
  
if (!step) {  
    step = DEFAULT_STEP;  
}  
  
if (!stop) {  
    stop = start;  
    start = DEFAULT_START;  
}  
  
if (stop < start) {  
    (function () {  
        // reverse values  
        var tmp = start;  
        start = stop;  
        stop = tmp;  
    })();  
}  
  
(function () {  
    var i;  
    for (i = start; i < stop; i += step) {  
        arr.push(i);  
    }  
})();  
  
return arr;  
}  
  
namespace.range = range;  
  
}(window.mylib));
```

Lesson 5. Declaring constants with `const`

After reading [lesson 5](#), you will

- Understand what constants are and how they work.
- Know when to use constants.

The keyword `const` stands for *constant*, as in *never changing*. Many programs have values that never change, whether by intention or by happenstance. Values declared with `const`, referred to as *constants*, have the same characteristics as those declared with `let`, which you learned about in the previous chapter, with the additional feature of reassignment being forbidden.

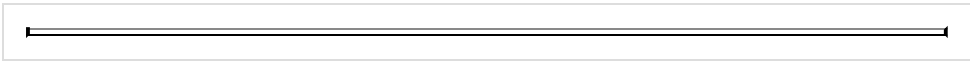
Consider this

Consider the following `switch` statement, which uses flags to determine what type of action is being performed. The uppercasing of `ADD_ITEM` and `DEL_ITEM` indicates that these are values that are never expected to change,^[a] but what would happen if they did change? How would that affect the behavior of the program? How could you author the application to protect from that?

^a

This uppercasing is purely by convention and not a required syntax.

```
switch (action.type) {
  case ADD_ITEM:
    // handle adding a new item
    break;
  case DEL_ITEM:
    // handle deleting an item
    break;
};
```



5.1. HOW CONSTANTS WORK

Constants cannot be reassigned. This means that once you assign the value of a constant, any attempt to assign a new value will result in an error:

```
const myConst = 5;  
  
myConst = 6;           1
```

- **1 Error**

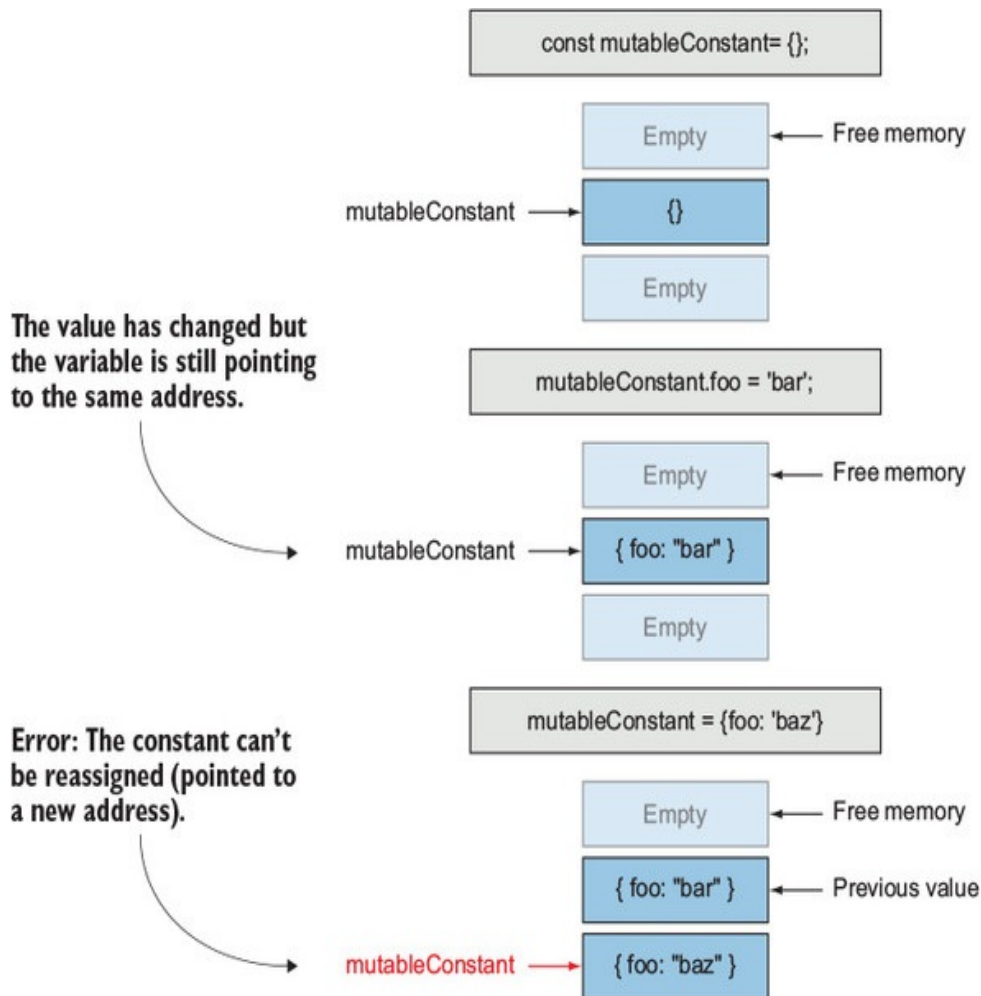
Because of the inability to reassign constants with new values, they quickly became confused with being immutable. Constants are not immutable. So what's the difference between not being able to be reassigned and being immutable? Assignment has to do with variable bindings, binding a name to a piece of data. Immutability or mutability is a property that belongs to the actual data that the binding contains. All primitives (strings, numbers, and so on) are immutable, whereas objects are mutable.

Let's take a look at an example where you assign a mutable object to a constant and are free to mutate it:

```
const mutableConstant = {};  
  
mutableConstant.foo = 'bar';           1  
mutableConstant.foo = 'baz';           2  
  
mutableConstant = {foo: 'bat'}         3
```

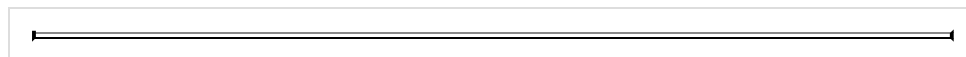
- **1 When you set foo to bar you're modifying (mutating) the existing object.**
- **2 When you set foo to baz you're modifying (mutating) the existing object.**
- **3 You're trying to set foo to baz by assigning an entirely new object, which is forbidden.**

Figure 5.1. Mutating the value



Because you created a constant and assigned a mutable value to it, you were able to mutate that value. But you could not assign a new value to the constant; the only way you were able to change the value was by modifying the value itself, as shown in [figure 5.1](#).

If you assign an immutable value like a number to a constant, then the constant becomes immutable because it contains a value that can't be modified and the constant can't be reassigned, so they become frozen.



Quick check 5.1

Q1:

What will happen when the following code is

executed?

```
const i = 0;  
i++;
```

QC 5.1 answer

A1:

Because the increment operator assigns a new value to the variable it operates on, this code will throw an error because constants cannot be reassigned.

The claim that primitives, like strings and numbers, are immutable may be a tough pill to swallow if you aren't already familiar with it. You may be thinking something like "I use primitives and change their value all the time!" The fact is that with the ability to reassign variables with new values, primitives are harder to spot. Consider this code:

```
let a = "Hello";  
let b = a;  
  
b += ", World!";  
  
console.log(a);           1  
console.log(b);           2
```

- **1 Hello**
- **2 Hello, World!**

In this example it appears that you're changing the string contained in variable b, but in fact you're creating a new string and reassigning that new string to b. You can

confirm this because, before updating b, it contains the same value as a, but afterwards a remains unchanged. This is because a is still pointing to the same string but b was reassigned a new string. This is why you can't use the += operator with constants. See [figure 5.2](#).

Quick check 5.2

Q1:

What will happen when executing the following code?

```
const a = "Hello";  
  
const b = a.concat(", World!");
```

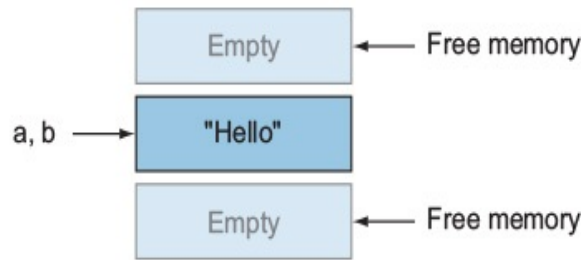
QC 5.2 answer

A1:

You may think that `a.concat` will throw an error when a is a constant, but remember the `concat` method on strings doesn't modify the existing string or reassign the variable containing it: it simply returns a new string. Because of this, the statement is valid and b becomes the string "Hello, World!"

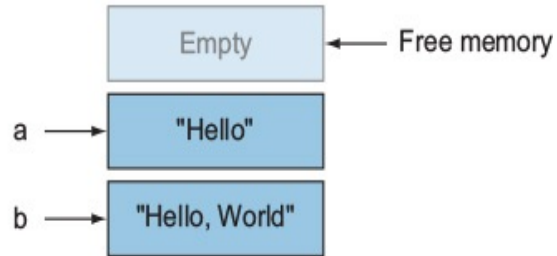
Figure 5.2. Demonstrating that primitives are immutable

```
var a = "Hello"; var b = a;
```



The original value hasn't changed but the variable points to a new value.

```
b += ", World";
```



5.2. WHEN TO USE CONSTANTS

The obvious place to use constants is when creating flags that are more about a unique identifier than the actual value they contain. Like this priming exercise, for example:

```
const ADD_ITEM = 'ADD_ITEM';
const DEL_ITEM = 'DEL_ITEM';
let items = [];

function actionHandler(action) {
  switch (action.type) {
    case ADD_ITEM:
      items.push(action.item);
      break;
    case DEL_ITEM:
      items.splice(items.indexOf(action.item), 1);
      break;
  };
}
```

This ensures that the action flags `ADD_ITEM` and `DEL_ITEM` can never be accidentally changed. You may stop there, but if you think about it, should the `items` array ever be reassigned? Probably not, so you can make that a `const` as well:

```
const items = [];
```

But what if you later decide that you need an action to empty the list? Your instinct may be to reassign `items` to a new empty array, but that can't be done using a constant:

```
case CLEAR_ALL_ITEMS:
  items = [];
  break;
```

- **1 Error**

You can still empty the array, though; you just need to figure out a way to modify the actual value without assigning a new value. In this case you can again use `splice` for such a task:

```
case CLEAR_ALL_ITEMS:
  items.splice(0, items.length);
  break;
```

OK, so your complete code now looks like this:

```
const ADD_ITEM = 'ADD_ITEM';
const DEL_ITEM = 'DEL_ITEM';
const CLEAR_ALL_ITEMS = 'CLEAR_ALL_ITEMS';
const items = [];

function actionHandler(action) {
  switch (action.type) {
    case ADD_ITEM:
      items.push(action.item);
      break;
    case DEL_ITEM:
      items.splice(items.indexOf(action.item), 1);
      break;
    case CLEAR_ALL_ITEMS:
      items.splice(0, items.length);
      break;
  };
}
```

Notice how you declared every single value with a `const`? This won't be an unusual circumstance.

Protecting a binding from reassignment isn't the only reason to use a constant. Because constants are never reassigned, certain optimizations can be made by the JavaScript engine to improve performance. Because of this, it makes sense to use `const` anytime a variable never needs to be reassigned, and fall back to `let` when they do.



Quick check 5.3

Q1:

What will happen when executing the following code?

```
function getValue() {  
  const val = 5;  
  return val;  
}  
  
let myVal = getValue();  
myVal += 1;
```

QC 5.3 answer

A1:

You may think that because the value is originally stored in a constant before being returned from the function, it would give errors when reassigned outside of the function. But remember that constants have to do with value bindings, not the values inside those bindings. The function merely returns the value, not the binding. So the new `let` binding is safe to reassign.

SUMMARY

In this lesson, you learned how to declare variables with `const` and how that differs from variables declared with `var` or `let`.

- Constants are variables that can't be reassigned.
- Constants are not immutable.
- Constants have the same scoping rules as `let`.

Let's see if you got this:

Q5.1

Modify your answer from the lesson 4 exercise to use `const` where possible.

Lesson 6. New string methods

After reading [lesson 6](#), you will

- Know how to use `String.prototype.startsWith`
- Know how to use `String.prototype.endsWith`
- Know how to use `String.prototype.includes`
- Know how to use `String.prototype.repeat`
- Know how to use `String.prototype.padStart`
- Know how to use `String.prototype.padEnd`

None of these methods would be extremely difficult to implement, but they are tasks that are used enough to warrant inclusion in the standard library.

Consider this

Let's say you're writing a function that tells the current time. Using an instance of the `Date` object, you can get the current hour and minutes with the `getHours` and `getMinutes` methods, respectively:

```
function getTime() {  
  const date = new Date();  
  return date.getHours() + ':' + date.getMinutes();  
}
```

But if the current time were 5:06 a.m., this function would return the string `5:6`. How can you fix the function so that the minutes side is always two digits?

6.1. SEARCHING STRINGS

Imagine you're loading products from a database. The product data comes from several different manufacturers and isn't normalized. Because of this, some of the prices are in the format 499.99 without a leading \$, whereas others are in the format \$37.95 with a leading \$. When you display the price on the web page, you can't simply prefix all prices with a \$ because that would make some have a double dollar sign like this: \$\$37.95.

You could easily tell if the first character is a \$ like so:

```
if( price[0] === '$' ) {  
  // price starts with $  
}
```

But what if you need to check the first three characters? Imagine you're showing a list of phone numbers. They all have the format XXX-XXX-XXXX and you need to determine which ones are in the current user's area code. You may do something like this:

```
if( phone.substr(0, 3) === user.areaCode ) {  
  // phone number is in users area code  
}
```

Both of these are doing the same thing, aren't they—checking if a given string starts with a given value? So why do they go about it in different ways? The answer is because, in the former scenario, you were checking against a single character, and in the latter, against a range of characters. Neither of these are doing a great job of making it easy to understand what you're checking; hence the need for a comment explaining what each does. With ES6, you can now solve both these problems

in the same *self documenting* way:

```
if( price.startsWith('$') ) {  
  // ...  
}  
if( phone.startsWith(user.areaCode) ) {  
  // ...  
}
```

Not only are these now more consistent, but isn't it much easier to understand exactly what they're doing at a glance?

With the addition of the `includes`, `startsWith`, and `endsWith` methods, searching for strings within strings has gotten much simpler. `startsWith` checks if a string starts with a specified value, and `endsWith` checks if the string ends with the value. `includes` checks the entire string to see if it contains the specified value:

```
let str = 'foo-bar';  
  
str.startsWith('foo');           1  
str.startsWith('bar');           2  
  
str.endsWith('foo');              3  
str.endsWith('bar');              4  
  
str.includes('foo');              5  
str.includes('bar');              5  
str.includes('o-b');              5
```

- **1 true**
- **2 false**
- **3 false**
- **4 true**
- **5 true**

All of these methods are case-sensitive. You can always lowercase the string before performing the search, though:

```
let location = 'Atlanta, Ga';
location.endsWith('GA');           1
location.endsWith('ga');           2
location.toLowerCase().endsWith('ga'); 3
```

- **1 false**
- **2 false**
- **3 true**

All of these methods also accept a second parameter to specify the position in the string to start searching:

```
let locationA = 'Atlanta, Ga';
let locationB = 'Galveston, Tx';

locationA.includes('Ga');           1
locationB.includes('Ga');           2

locationA.includes('Ga', locationA.indexOf(' ')); 3
locationB.includes('Ga', locationB.indexOf(' ')); 4
```

- **1 true**
- **2 true**
- **3 true**
- **4 false**

If your needs are more complex than what's achievable with these methods, you can still fall back to using regular expressions for more custom string searching.



Quick check 6.1

Q1:

Assume you have an array of objects representing weather data. Each object has an `icon` property that specifies the name of an icon that represents the weather condition. The icon name ends with `night` if the icon being used is the nighttime version of the icon. Write a filter that grabs all of the objects with

nighttime icons using one of the three methods we covered.

QC 6.1 answer

A1:

```
let nightIcons = weatherObjs.filter(function(weather) {  
  return weather.icon.endsWith('-night');  
});
```

6.2. PADDING STRINGS

Padding a string means specifying how long you want the string to be, and filling the string with a *pad* character until it is that length. For example, in the priming exercise, you wanted the minutes to always be two characters long. Sometimes it was (such as 36) but other times it wasn't (such as 5). By padding the string to a length of 2 with the pad character being 0, 36 would be unchanged because it is already two chars while 5 would become either 05 or 50 depending on whether you were padding left or right.

Let's say you needed a function that could take an IP address written in decimal (base 10) and convert it to binary (base 2). You may end up writing a function that looks like this:

```
function binaryIP(decimalIPStr) {  
  return decimalIPStr.split('.').map(function(octet) {  
    return Number(octet).toString(2);  
  }).join('.');  
}
```

But this function wouldn't work for any number less than 128 because its binary representation would be less than eight digits.

```
binaryIP('192.168.2.1');    1
```

- 1 “11000000.10101000.10.1”

To fix this, you need to pad each octet with zeros until each one is eight digits, as shown in the example at the top of the next page. To accomplish this, you can use a new feature introduced in ES2015, `String.prototype.repeat`. We could also use

`padStart`, but that's a bit more flexible as you'll see in a minute, and `repeat` is also useful to know.

```
function binaryIP(decimalIPStr) {  
  return decimalIPStr.split('.').map(function(octet) {  
    let bin = Number(octet).toString(2);  
    return '0'.repeat(8 - bin.length) + bin;  
  }).join('.');  
}  
  
binaryIP('192.168.2.1');    1
```

- 1 “11000000.10101000.00000010.00000001”

This works because the `repeat` function repeats the function it's invoked for the number of times specified in the argument:

```
'X'.repeat(4);    1
```

- 1 XXXX (X is repeated 4 times)

If the number specified isn't a whole number, it will first be floored (rounded down, not rounded up):

```
'X'.repeat(4.9);    1
```

- 1 XXXX

This works on strings of any length as well:

```
'foo'.repeat(3);    1
```

- 1 foofoofoo

Using `String.prototype.repeat` works for our binary example. But what if you wanted to pad with a string that's more than one character? That would be trickier to achieve with just the `repeat` function, but you can easily achieve it with `String.prototype.padStart` and

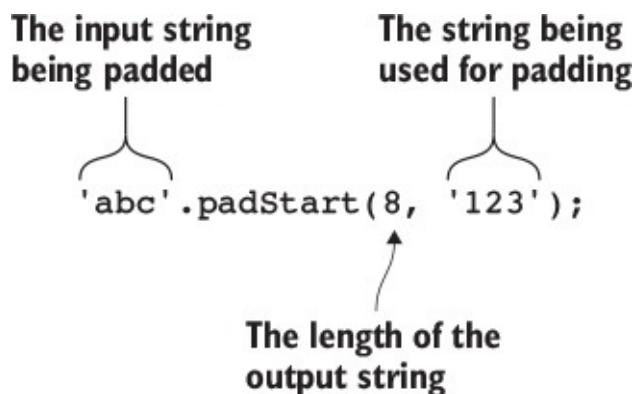
String.prototype.padEnd.

The padStart and padEnd methods take two arguments: the max length of the returned string, and the filler string, as shown in [figure 6.1](#). The filler string defaults to a space " " character:

'abc'.padStart(6);	1
'abc'.padEnd(6);	2
'abc'.padStart(6, 'x');	3
'abc'.padEnd(6, 'x');	4
'abc'.padStart(6, 'xyz');	5
'abc'.padEnd(6, 'xyz');	6

- **1** “abc”
- **2** “abc ”
- **3** “xxxabc”
- **4** “abcxxx”
- **5** “xyzabc”
- **6** “abcxyz”

Figure 6.1. Dissecting the padStart function



The max length property specifies the maximum length of the string after the filler has been repeated. If the filler is multiple chars and can’t evenly be added, it will be truncated:

'abc'.padStart(8, '123');	1
'abc'.padEnd(5, '123');	2

- **1** “12312abc”
- **2** “12abc”

If the max length is less than the original string’s length, the original string won’t be truncated but will be returned without any padding applied:

```
'abcdef'.padStart(4, '123');      1
```

- **1** “abcdef”

Using this, your `binaryIP` function could then be rewritten like so:

```
function binaryIP(decimalIPStr) {
  return decimalIPStr.split('.').map(function(octet) {
    return Number(octet).toString(2).padStart(8, '0')
  }).join('.');
}

binaryIP('192.168.2.1');          1
```

- **1** “11000000.10101000.00000010.00000001”

That’s a pretty concise implementation that’s easy to read and understand. Using pure ES5 code would have been much more verbose to achieve the same thing. Of course, you could have used a third-party string pad function or written your own, but `String.prototype.padStart` and `String.prototype.padEnd` eliminated that need.

Quick check 6.2

Q1:

Write a function using `repeat` that will repeat any string to exactly 50 chars, truncating any excess.

QC 6.2 answer

A1:

```
function repeat50(str) {  
  const len = 50;  
  return str.repeat(Math.ceil(len / str.length)).substr(0,  
len);  
}
```

SUMMARY

In this lesson, the object was to teach you the most useful new methods being added to strings.

- `String.prototype.startsWith` checks if a string starts with a value.
- `String.prototype.endsWith` checks if a string ends with a value.
- `String.prototype.includes` checks if a string includes a value.
- `String.prototype.repeat` repeats a string a given number of times.
- `String.prototype.padStart` pads the start of a string.
- `String.prototype.padEnd` pads the end of a string.

Let's see if you got this:

Q6.1

Write a function that will take an email address and mask all of the characters up to the `@`. For example, the email address `christina@example.com` would be masked to `*****@example.com`.

Lesson 7. Template literals

After reading [lesson 7](#), you will

- Know how to achieve string interpolation with template literals
- Understand how to use multiline strings with template literals
- Know how to make reusable templates with template literals
- Understand how tagged template literals can add custom processing to template literals

In JavaScript, one of my biggest annoyances has always been the lack of support for multiline strings. Having to break each line into a separate string and glue them all together is a tedious process. With the addition of template literals, that pain is gone, along with some others. *Template literals* introduce multiline strings to JavaScript, as well as interpolation and tagging. Don't worry if you're unsure what these terms mean: I'll explain each one.

Consider this

Study the following function, which takes a product object and returns the appropriate HTML to represent the product. Currently it's only displaying a photo and description of the product, and it's already difficult to deduce what's going on due to the many strings being glued together to make up for the lack of support for multi-line strings and interpolation. A real product would probably need to show a title, a price, and other details, making it even harder to deduce. As the HTML requirements for the product become more and more complex, what steps would you take to keep the code readable?

```
function getProductHTML(product) {  
  let html = '';  
  html += '<div class="product">';  
  html += '<div class="product-image">';  
  html += '';  
  html += '</div>';  
  html += '<div class="product-desc">' + product.desc +  
'</div>';  
  html += '</div>';  
  return html;  
}
```



7.1. WHAT ARE TEMPLATE LITERALS?

A template literal is a new literal syntax for creating strings. It uses back-ticks (`) for demarcation instead of quotes (' or ") like string literals do, and supports new features and functionality. But template literals evaluate to a string just like string literals do:

```
let str1 = `Hello, World`;
let str2 = "Hello, World";
let str3 = `Hello, World`;      1
console.log(str1 === str3);     2
console.log(str2 === str3);     2
}
```

- **1 Creating a string using a template literal**
- **2 true**

The difference is that template literals support three new features that regular string literals do not: *interpolation*, *multiline*, and *tagging*. Let's define each of those terms.

String interpolation—This is the concept of placing a dynamic value into the creation of a string. Many other languages support this, but up until now, JavaScript could only put dynamic values into larger strings by concatenation:

```
let interpolated = `You have ${cart.length} items in your cart`

let concatenated = 'You have ' + cart.length + ' items in your cart';
```

Multiline strings—This is not the concept of creating a string that is multiline. JavaScript has always been able to do that (for example, "Line One\nLine Two"). This is the concept of the literal itself spanning multiple lines. Regular strings in JavaScript must be defined in a

single line, meaning the opening quote and the ending quote must be on the same line. With template literals, this is not the case:

```
let multiline = `line one
                 lines two`
```

Tagging template literals—This is a much more advanced use. Remember how template literals can have values interpolated into them? A tagged template literal is a template literal that's *tagged* with a function. The function is given all the raw string parts as well as the interpolated values separately so it can do custom preprocessing of the string. It can return an entirely different string or a custom value that isn't even a string. This is especially useful if you want to do some sort of preprocessing on the interpolated values before merging them into the final value. For example, you could have a tagging function that converts the template literal into DOM nodes but escapes the interpolated values to protect from HTML injection. You could also prevent SQL injection by treating the string parts as safe and the interpolated values as potentially dangerous:

```
let html = domTag`<div class="first-name">${userInput}</div>`
```

So now we have an overview; let's look at each of these in more detail, starting with interpolation.

7.1.1. String interpolation with template literals

The syntax for interpolating value into template literals is to wrap the value to be interpolated with curly braces { and } and to prefix the opening curly brace with the dollar sign \$:

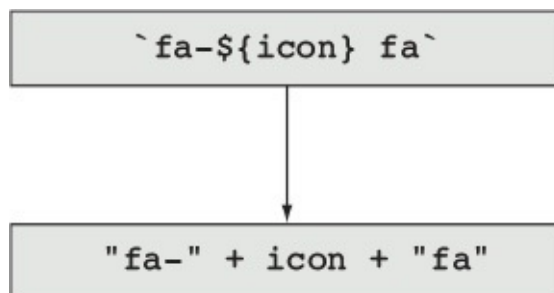
```
function fa(icon) {
  return `fa-${icon} fa`;
}

fa('check-square');          1
```

- 1 “fa-check-square fa”

This takes the value from the `icon` variable and injects it into the template string. You can visualize what’s happening in [figure 7.1](#).

Figure 7.1. The value of the `icon` variable is interpolated into the string



This function generates the Font Awesome (<http://fontawesome.io/icon/check-square/>) CSS class needed for a given icon. The more interpolation required, the more obvious it becomes how nice a feature this is, as shown in the code examples on the following page.

```
function greetUserA(user, cart) {
  const greet = `Welcome back, ${user.name}.`;
  const cart = `You have ${cart.length} items in your cart.`;
  return `${greet} ${cart}`;
}

function greetUserB(user, cart) {
  const greet = 'Welcome back, ' + user.name + '.';
  const cart = 'You have ' + cart.length + ' items in your cart.';
  return greet + ' ' + cart;
}

function madlibA(adjective, noun, verb) {
  return `The ${adjective} brown ${noun} ${verb}s`;
}

function madlibB(adjective, noun, verb) {
  return 'The ' + adjective + ' brown ' + noun + ' ' + verb + 's';
}
```

```
madlibA('quick', 'fox', 'jump');      1
madlibB('quick', 'fox', 'jump');      1
```

- **1 “The quick brown fox jumps”**

In each example, both these functions are equivalent in functionality. I don’t think anyone would make the argument in either case, though, that variant B is preferred.

Note that you can escape the interpolation using the normal escape character:

```
let color = 'hazel';

let str1 = `My eyes are ${color}`;      1
let str2 = `My eyes are \${color}`;     2
```

- **1 “My eyes are hazel”**
- **2 “My eyes are \${color}”**

But if you are just adding a dollar sign that isn’t followed by an opening brace then there’s no need to escape anything:

```
let price = `Only ${5.0.toFixed(2)}`;    1
```

- **1 “Only \$5.00”**

Any value you interpolate that isn’t already a string will be converted to a string, and its string representation will be used:

```
let obj = { foo: 'bar' };
let str = `foo: ${obj}`;

console.log(str);      1
```

- **1 “foo: [object Object]”**

Quick check 7.1

Q1:

What does the template literal assigned to the phrase variable evaluate to?

```
let fruit = 'banana';  
let color = 'yellow';  
  
let phrase = `the ${`big ${color}`} ${fruit}`;
```

QC 7.1 answer

A1:

If a template literal is interpolated into another template literal, the inner literal will be evaluated to a string and then its string value will be interpolated into the outer template literal. So the final value here would be the `big yellow banana`.

7.1.2. Multiline strings with template literals

Unlike string literals, an unterminated template literal will continue to the following line until terminated:

```
let multiline = `Hello,  
World`;  
  
console.log(multiline);      1
```

- **1** “Hello,\nWorld”

One thing to definitely keep in mind is that multiline template literals preserve all of their white space

characters:

```
function greetUser(user, cart) {
  return `Welcome back, ${user.name}.
        You have ${cart.length} items in your cart.`;
}

greetUser(currentUser, currentCart);
```

1

- 1 “Welcome back, JD.\n You have 0 items in your cart.”

You can now use a template literal to solve the priming exercise:

```
function getProductHTML(product) {
  return `
    <div class="product">
      <div class="product-image">
        
      </div>
      <div class="product-desc">${product.desc}</div>
    </div>
  `;
}
```

Quick check 7.2

Q1:

Are the following two functions equivalent?

```
// Using String Literals
function getProductHTML(product) {
  let html = '';
  html += '<div class="product">';
  html += '<div class="product-image">';
  html += '';
  html += '</div>';
  html += '<div class="product-desc">' + product.desc +
'</div>';
  html += '</div>';
  return html;
}
```

```
// Using Template Literals
function getProductHTML(product) {
  return `
    <div class="product">
```

```
    <div class="product-image">
      
    </div>
    <div class="product-desc">${product.desc}</div>
  </div>
  `;
}
```

QC 7.2 answer

A1:

No, they're close, but because the template literal preserves white space, it will include the new lines and spaces, which might slightly affect the layout when rendered to the page.

7.2. TEMPLATE LITERALS ARE NOT REUSABLE TEMPLATES

I think the name *template literal* is a bit misleading because it creates strings, not templates. When developers think of templates, they usually think of something that's reusable. Template literals aren't reusable templates, but rather one-time templates that get evaluated and turned into a string all at once, so there's no reusability to them. If you aren't familiar with using other JavaScript templates, like those from Underscore, Lodash, Handlebars, or several others, then you may not be too confused. If you are, on the other hand, coming from a template library like one of the ones I listed, let's explore the difference:

```
let name = 'Talan';

// Using underscore
let greetA = _.template('Hello, <%= name %>');

greetA(name);           1
greetA('Jonathon');     2

// Using template literals
let greetB = `Hello, ${name}`;
greetB;                 3
greetB('Jonathon');     4
```

- **1 Hello, Talan**
- **2 Hello, Jonathon**
- **3 Hello, Talan**
- **4 Uncaught TypeError: greetB is not a function**

When you create a template from a library like Underscore, you specify placeholders for values and you get back a function that you can invoke multiple times with different values to be interpolated. A template

literal, on the other hand, interpolates the values at the time it's created, so there's no re-evaluating it with new values.

You can, however, create a reusable template by wrapping the template literal in a function:

```
let name = 'Talan';

function greet(name) {
  return `Hello, ${name}`;
}
greet(name); // Hello, Talan
greet('Jonathon'); // Hello, Jonathon
```

Quick check 7.3

Q1:

In the following code, what would be logged to the console?

```
let name = 'JD';
let greetingTemplate = `Hello, ${name}`;
name = 'Jon';
console.log(greetingTemplate);
```

QC 7.3 answer

A1:

Hello, JD

7.3. CUSTOM PROCESSING WITH TAGGED TEMPLATE LITERALS

Imagine you're in charge of a growing ecommerce website. Originally all your users were English speakers but now you need to support multiple languages.

Wouldn't it be nice if there were a way to mark certain strings that needed to be internationalized? What if you could mark a string by prefixing it with `i18n`^[1] and have it automatically translated for a user's locale?

¹

`i18n` is a common shortening of *internationalization*, meaning `i` + 18 letters + `n`.

```
function greetUser(user, cart) {  
  return i18n`Welcome back, ${user.name}.  
           You have ${cart.length} items in your cart.`;  
}
```

A tagged template literal is a template literal that's tagged with a function by prefixing the template literal with the name of that function. The basic syntax is as follows:

```
myTag`my string`
```

By doing this, you're tagging `my string` with the function `myTag`. The `myTag` function will be invoked and given the template as arguments. If the template has any interpolated values, they're passed as separate arguments. The function is then responsible for performing some type of processing of the template and returning a new value.

There is currently only one built-in tagging function, `String.raw`, which processes a template without interpreting special characters:

```
const a = `my\tstring`;           1
const b = String.raw`my\tstring`; 2
```

- **1 “my string”**
- **2 “my\tstring”**

This may be the only built-in tagging function, but you can use ones by library authors, or write your own!

When the template literal is tagged with a function, it will be broken apart into the string parts and the interpolated values. The tagged function will be given these values as separate arguments. The first argument will be an array of all the string parts; then for every interpolated value, it will be given as an additional argument to the function:

```
function tagFunction(stringPartsArray, interpolatedValue1, ...,
    interpolatedValueN)
{
}
```

Fortunately, you don’t ever need to write these arguments out explicitly.

To get a better understanding of how these pieces are broken apart and passed, write a function that simply logs the parts:

```
function logParts() {
  let stringParts = arguments[0];
  let values = [].slice.call(arguments, 1);           1
  console.log( 'Strings:', stringParts );
  console.log( 'Values:', values );
}

logParts`1${2}3${4}${5}`;                             2
```

- **1 Gather all the arguments after the first as an array.**
- **2 Strings: [“1”, “3”, “”, “”] Values: [2, 4, 5]**

Here you use JavaScript’s `arguments` object to parse

out your string parts and interpolated values. You use `arguments[0]` to get the first parameter, which is an array of the string parts, and you use `[] .slice.call(arguments, 1)` to get an array of all the parameters after the first. This array will contain all the interpolated values.

You should have expected the strings 1 and 3, but why are there two additional empty strings? If values being interpolated are next to each other without any string parts in between, such as the 4 and the 5, there will be an implicit empty string between them. This is the same for the start and end of the string as well. You can visualize how this gets computed with the following graph:

Strings	"1"		"3"		"		"
Values		2		4		5	

Because of this, the first piece and last piece will always be strings and the values will be what glues those strings together. This also means you can easily process these values with a simple `reduce` function:

```
function noop() {  
  let stringParts = arguments[0];  
  let values = [].slice.call(arguments, 1);  
  return stringParts.reduce(function(memo, nextPart) {  
    return memo + String(values.shift()) + nextPart;  
  }, '');  
}  
  
noop`1${2}3${4}${5}`;  
2
```

- **1** The `.shift()` function removes and returns the first value of an array.
- **2** `"12345"`

In this function, you processed the template literal the

same as if it had not been tagged. But the point is to allow custom processing. This feature will most likely be implemented by library authors, but you can still use it to create some custom processing. Write a function that strips the whitespace between your markup:

```
function stripWS() {
  let stringParts = arguments[0];
  let values = [].slice.call(arguments, 1);
  let str = stringParts.reduce(function(memo, nextPart) {
    return memo + String(values.shift()) + nextPart;
  });
  return str.replace(/\n\s*/g, '');
}
function getProductHTML(product) {
  return stripWS`
    <div class="product">
      <div class="product-image">
        
      </div>
      <div class="product-desc">${product.desc}</div>
    </div>
  `;
}
```

We've only scratched the surface of what you can do with template literals. Because they allow for preprocessing of the interpolated values separately from the string parts of the literal and can return any data type, they're perfect for creating abstractions known as DSLs (domain-specific languages), which you'll be diving into next in your first capstone project.

Quick check 7.4

Q1:

In the following snippet there are three lines of code. Which is the correct syntax for using a function to tag a template literal?

```
myTag(`my templat`)
myTat()`my templat`
```



```
myTag`my templat`
```

QC 7.4 answer

A1:

The last one: myTag`my templat`.

SUMMARY

In this lesson, you learned the basics of template literals.

- Template literals are a new literal syntax for creating strings using the back tick ` character.
- Template literals can interpolate values with `\${}`.
- Template literals can span multiple lines.
- Template literals will open the door for domain-specific languages with tagged templates.

Let's see if you got this:

Q7.1

Write a function you can use to tag template literals that checks whether each value is an object, and if so, converts the object into a string of key/value pairs before interpolation.

For example if you call the function like so

```
const props = {  
  src: 'http://fillmurray.com/100/100',  
  alt: 'Bill Murray'  
};  
  
const img = withProps`<img ${props}>`;
```

you would get back

```

```

Lesson 8. Capstone: Building a domain-specific language

In this project, you will

- Learn what a domain-specific language (*DSL*) is
- Create a simple DSL for handling user input from an HTML formatter
- Create a simple DSL for repeating or looping templates around an array

Before you build a DSL, let's first define what one is. A *domain-specific language* uses a programming language's features and syntax in a clever way to make solving a particular task appear as if it has first-class support by the programming language itself. For example, a common type of DSL seen in JavaScript is used by unit test libraries:

```
describe('push', function() {  
  it('should add new values to the end of an array', function()  
  {  
    // perform test  
  });  
});
```

This is a DSL because it looks like you're using natural language to state that you are describing push, and then giving the actual description: *it should add new values to the end of an array*. In reality, the `describe` and `it` parts of the sentences are functions, and the rest of the sentences are parameters to those functions.

Other languages such as Ruby have a lot of DSLs. Ruby in particular is so malleable and has so much syntactic sugar that it makes creating DSLs extremely easy. JavaScript, on the other hand, has not historically lent itself to easy DSL creation. But with tagged templates,

that may change. For example, a clever programmer might be able to utilize tagged templates to change the preceding unit-testing DSL into this syntax:

```
describe`push: it should add new values to the end of an  
array`{  
  // perform test  
}
```

You aren't going to implement this DSL, but you are going to create a couple of your own. In the following sections you will

- Create some helper functions for string processing
- Create an HTML-escaping DSL
- Create a DSL for converting arrays into HTML

8.1. CREATING SOME HELPER FUNCTIONS

Before we begin, you'll create a couple helper functions. In the last lesson, when creating a template tagging function, you started the function with some code that resembled the following:

```
function stripWS() {  
  let stringParts = arguments[0];  
  let values = [].slice.call(arguments, 1);  
  // do something with stringParts and values  
}
```

These first two lines are for the purpose of getting the string values and the interpolated values of the template literal. They are a bit obscure, though, and detract from the rest of the logic. You'll soon discover much better ways of collecting these values, but for now you'll abstract this step away in a layer that hides this part from the rest of your logic. How? You'll create a separate function to do this and then pass the collected values to your tag function.

Listing 8.1. Abstracting the process of collecting interpolated values

```
function createTag(func) {  
  return function() {  
    const str = arguments[0];  
    const vals = [].slice.call(arguments, 1);  
    return func(str, vals);  
  }  
}
```

Now when you create a tagging function, you can do this, which is far neater:

```
const stripWS = createTag(function(str, vals){  
  // do something with str and vals  
});
```

Another redundant step was when you had to use

reduce to interlace the given strings and interpolated values. You can abstract that step away as well.

Listing 8.2. Abstracting of interlacing strings and interpolated values

```
function interlace(strs, vals) {  
  vals = vals.slice(0);  
  return strs.reduce(function(all, str) {  
    return all + String(vals.shift()) + str;  
  });  
}
```

- **1 This line is duplicating the array, a good practice before invoking methods that mutate the array in place like shift.**

You can now use the combination of `createTag` and `interlace` to process a template literal in the standard implementation, as shown in the next listing.

Listing 8.3. Processing a template literal

```
const processNormally = createTag(interlace);  
  
const text = 'Click Me';  
const link = processNormally`<a>${text}</a>`;
```

- **1 <a>Click Me**

Now that you have these parts abstracted away, you can write a tagging function that focuses on the business at hand—escaping an HTML string.

8.2. CREATE AN HTML-ESCAPING DSL

Let's consider the business logic behind getting a template literal and some values to HTML escape and interpolate. Your first tag is going to HTML escape your interpolated values. What you want to do is convert any occurrence of `<` and `>` with `<` and `>` respectively. You'll use this to HTML escape user input to prevent injecting unintended HTML into your document. Start by writing a function that will do this for a given string, as shown in the next listing.

Listing 8.4. A simple HTML-escaping function

```
function htmlEscape (str) {  
  str = str.replace(/</g, '&lt;');  
  str = str.replace(/>/g, '&gt;');  
  return str;  
}
```

This function uses regular expressions to replace the `<` and `>` characters with `<` and `>` respectively. Now that you have this, writing your tagging function is going to be super simple.

Listing 8.5. Your tagging function, `htmlSafe`

```
const htmlSafe = createTag(function(strs, vals){  
  return interlace(strs, vals.map(htmlEscape));  
  1  
});
```

- **1 Calls `htmlEscape` on only the values to be interpolated**

The reason why this is beneficial as a tagging function as opposed to just using the `htmlEscape` function directly is because it allows you to target only the interpolated values for escaping, as seen in [listing 8.6](#).

Listing 8.6. Basic HTML-escaping DSL in action

```
const userInput = 'I <3 ES6!';
```

```
const a = htmlEscape(`<strong>${userInput}</strong>`);  
1  
const b = htmlSafe`<strong>${userInput}</strong>`;   
2
```

- **1** `I <3 ES6!`
- **2** `I <3 ES6!`

Notice how the string **a** has all of the HTML pieces escaped but the string **b** correctly only escaped the `<3`.

8.3. CREATE A DSL FOR CONVERTING ARRAYS INTO HTML

Take a look at the following code. What would you expect the value `list` to be?

```
const fruits = ['apple', 'orange', 'banana'];

const list = expand`<li>${fruits}</li>`;
```

Wouldn't it be great to have a utility that could expand the template by repeating it around the array of values to produce the following HTML?

```
<li>apple</li>
<li>orange</li>
<li>banana</li>
```

Let's see how to build it! It's actually simple. You just need to

- Grab the first and last part of the template
- Grab the interpolated array
- Map over the items in the array, wrapping each item with the parts of the template

All of which translates to the following code:

```
const expand = function(parts, items){
  const start = parts[0];
  const end   = parts[1];
  const mapped = items.map(function(item){
    return start + item + end;
  });

  return mapped.join('');
};
```

Now let's see it in action.

Listing 8.7. DSL interpolating an array of values, not just one

```
const lessons = [
```

```
'Declaring Variables with let',  
'Declaring constants with const',  
'New String Methods',  
'Template Literals',  
]  
  
const html = `

${  
  expand`<li>${lessons}</li>`  
}</ul>`
```

The `html` variable now looks like so:

```
<ul>  
  <li>Declaring Variables with let</li>  
  <li>Declaring constants with const</li>  
  <li>New String Methods</li>  
  <li>Template Literals</li>  
</ul>
```

Currently the output looks like HTML but is still merely a string. You could take this even further by modifying `expand` or making another template tag that converts the string to actual DOM nodes.

SUMMARY

In this capstone project you made use of both `let` and `const` for defining your variables, and you used template literals and tagged template literals to build DSLs. This is just scratching the surface of what you can do with the new string capabilities of ES6. In the next unit we're going to explore the new additions to objects and arrays.

Unit 2. Objects and arrays

Objects and arrays have always been the workhorse of JavaScript, acting as the go-to data structures for organizing data. Even with the additions of maps and sets, which you'll learn about in [unit 5](#), objects and arrays aren't going anywhere and will still be used just as much as before.

Notice how I referred to objects and arrays as *data structures*. With literals, you can easily structure your data into complex structures that would be tedious to describe without literals. The inverse has always been missing. You probably never even noticed its absence, but once you see how awesome it is to be able to *destructure* your data as easily as you *structure* it, you may begin to feel handicapped anytime you have to go back to doing it the old way. Destructuring is one of my favorite new additions to JavaScript, and for good reason. You'll find yourself using it everyday to make your code easier to read and write. But before you jump to destructuring, we'll take a look at some useful new methods being added to objects and arrays. We'll also look at some welcome additions to object literals that make them even more powerful than before.

Finally, we'll look at a completely new primitive data type, symbols. Symbols are commonly used to define what I call “meta behavior(s)” — hooks to alter or define the behavior of existing functionality. They can also be used to avoid the sorts of naming collisions you might run into with strings.

You'll wrap up this unit by programming a lock and key construct using the uniqueness of symbols as keys to the

locks. You'll then use these locks and keys to create a *Choose the Door* game where players try to unlock doors using keys they were given.

Lesson 9. New array methods

After reading [lesson 9](#), you will

- Know how to construct arrays with `Array.from`
- Know how to construct arrays with `Array.of`
- Know how to construct arrays with `Array.prototype.fill`
- Know how to search in Arrays with `Array.prototype.includes`
- Know how to search in Arrays with `Array.prototype.find`

Arrays are probably the most common data structure used in JavaScript. We use them to hold all kinds of data, but sometimes getting the data we want into or out of the array isn't as easy as it should be. But those tasks just got a lot easier with some of the new array methods that we'll cover in this lesson.

Consider this

Consider this snippet of jQuery code that grabs all the DOM nodes with a specific CSS class and sets them to the color red. If you were going to implement this from scratch, what considerations would you have to make? For example, if you were to use `document.querySelectorAll`, which returns a `NodeList` (not an `Array`), how would you iterate each node to update its color?

```
$('.danger').css('color', 'red');
```

9.1. CONSTRUCTING ARRAYS WITH `ARRAY.FROM`

Let's say you're going to write a function for averaging numbers. The function should take any number of arguments and return the average of all of those numbers. You may at first define this function like in [listing 9.1](#). But this implementation won't work because it forgets the fact that the `arguments` object isn't actually an array. After trying to use this function and getting an error like *arguments.reduce is not a function*, you would probably realize that you need to convert the `arguments` object into an array.

Listing 9.1. avg version 1: producing an error, as `arguments` is not an array

```
function avg() {  
  const sum = arguments.reduce(function(a, b) {  
    return a + b;  
  });  
  return sum / arguments.length;  
}
```

Before ES6, the common way of converting an array-like object into an array was by using the `slice` method on `Array.prototype` and applying it to the array-like object. This works because calling `slice` on an array without any arguments simply creates a shallow copy of the array. By applying the same logic to an array-like object, you still get a shallow copy, but the copy is an actual array:^[1]

¹

Using `Array.prototype.apply` works just as well.

```
Array.prototype.slice.call(arrayLikeObject);  
  
// or a shorter version:  
[].slice.call(arrayLikeObject);
```

With that in mind, you can fix your `avg` function by converting the `arguments` object into an array using this trick, as shown in the following listing.

Listing 9.2. `avg` version 2: using `slice` to convert arguments into an array

```
function avg() {
  const args = [].slice.apply(arguments);
  const sum = args.reduce(function(a, b) {
    return a + b;
  });
  return sum / args.length;
}
```

This shortcut is no longer needed with `Array.from`. The purpose of `Array.from` is to take an array-like object and get an actual array from it. The *array-like* object is any object with a `length` property. The `length` property is used to determine the length of the new array; any integer properties that are less than the `length` property will be added to the newly created array at the appropriate index. For example, a string has a `length` property and numeric properties indicating the index of each character. So calling `Array.from` with a string will return an array of characters. The `arguments` object can be converted into an array using this technique as well, as shown in the following listing.

Listing 9.3. Updating the `avg` function to use `Array.from`

```
function avg() {
  const args = Array.from(arguments);
  const sum = args.reduce(function(a, b) {
    return a + b;
  });
  return sum / args.length;
}
```


<code>avg(1, 2, 3);</code>	1
<code>avg(100, 104);</code>	2
<code>avg(10 , 99, 5, 46);</code>	3

- **1 Returns 2**

- **2 Returns 102**
- **3 Returns 40**

Another common use case for needing `Array.from` is in conjunction with `document.querySelectorAll`. `document.querySelectorAll` returns a list of matching DOM nodes, but the object type is a `NodeList`, not an array.

You aren't limited to built-in objects for use with `Array.from`. Any object with a `length` property will work, even if all it has is a `length` property:

```
Array.from({ length: 50 })
```

This creates a new array exactly the same as `new Array(50)` would. But what if you just wanted to create an array with a single value of 50 in it, not a length of 50? See the next section for the answer.

Quick check 9.1

Q1:

Convert the following code to make use of `Array.from`:

```
let nodes = document.querySelectorAll('.accordian-panel');
let nodesArr = [].slice.call(nodes);
nodesArr.forEach(activatePanel);
```

QC 9.1 answer

A1:

```
let nodes = document.querySelectorAll('.accordian-panel');
```

```
let nodesArr = Array.from(nodes);  
nodesArr.forEach(activatePanel);
```



9.2. CONSTRUCTING ARRAYS WITH ARRAY.OF

Having a function that treats its arguments one way sometimes and a completely different way at other times is usually considered a bad design. For example, in the previous section, you created an `avg` function that returns the average of all of its arguments. Now imagine it only had one argument: you would expect to only get that number back, right? After all, the average of any single number is the same number? But what if, when only one number was supplied, it did something completely different and gave you back the square root of that number. You would probably tell me that's a terrible design, but that type of mixed behavior is exactly how the `Array` constructor works. Consider the three following arrays; one of them is not what it seems:

<code>let a</code>	<code>= new Array(1, 2, 3);</code>	1
<code>let b</code>	<code>= new Array(1, 2);</code>	2
<code>let c</code>	<code>= new Array(1);</code>	3

- **1 The `a` array contains three values: 1, 2, 3.**
- **2 The `b` array contains two values: 1 and 2.**
- **3 Finally the `c` array contains a single value: undefined.**

It's kind of a quirk that when you declare `new Array(1)`, you get an array with an undefined value in it.^[2] The reason for this is a special behavior in the `Array` constructor, if there is only one argument and that argument is an integer, it creates a sparse array of length n , where n is the number passed in as an argument. To avoid this quirk, you can use the `Array.of` factory function that works more predictably.

It doesn't even have an undefined value in it; it has a hole.

```
let a = Array.of(1, 2, 3);  
let b = Array.of(1, 2);  
let c = Array.of(1);
```

The arrays created with `Array.of` are the same except for array `c`, which more intuitively this time contains the single value `1`. At this point, you may be thinking, why not just use an array literal? For example:

```
let a = [1, 2, 3];  
let b = [1, 2];  
let c = [1];
```

In most situations, an array literal is actually the preferred way to create arrays. But there are some situations where an array literal will not work. One such situation is using subclasses of arrays.^[3] Imagine you are using a library that provides a subclass of `Array` called `AwesomeArray`. Because it's a subclass of `Array`, it has the same quirks. So you can't simply call `new AwesomeArray(1)` because you'll get an `AwesomeArray` with a single undefined value. But you can't use an array literal here, because that will give you an instance of `Array`, not `AwesomeArray`. You can, however, use `AwesomeArray.of(1)` and get an instance of `AwesomeArray` with a single value of `1`.

³

We'll cover classes and subclasses in later units.

So now you can construct an array with a single numeric value with `Array.of(50)`, but what if you did indeed want an array with 50 values? You could go back to `new Array(50)`, but that still has issues, as we'll see in the next section.

Quick check 9.2

Q1:

Which of the following returns an array with undefined values?

```
new Array()  
new Array(true)  
new Array(false)  
new Array(5)  
new Array("five")
```

QC 9.2 answer

A1:

Which of the following returns an array with undefined values?

```
new Array(5)
```

9.3. CONSTRUCTING ARRAYS WITH `ARRAY.PROTOTYPE.FILL`

Imagine you're creating a tic-tac-toe game. Maybe you're building this for a client or maybe as a side project. Either way, tic-tac-toe is fun and you get to build it! To start, you need to decide how you're going to implement the board. A tic-tac-toe board is a 3×3 grid of 9 slots. You decide you'll just use an array for this. You can use an array of length 9 to represent the 9 grid slots and the possible values will be either "x", "o", or an empty space " ". You'll call this array *board* but you need to initialize with 9 spaces " ". Maybe you'll do something like:

```
const board = new Array(9).map(function(i) {  
  return ' '  
})
```

The thought process here is that you initialize the array with 9 values, all of them `undefined`. Then you use `map` to convert each `undefined` value into a space. But this won't work. When you create an array like `new Array(9)`, it doesn't actually add nine `undefined` values to the new array. It merely sets the newly created array's `length` property to 9.

If you're confused by that, let's first talk about how arrays work in JavaScript. An array isn't as special as many people think it is. Other than having a literal syntax like `[. . .]`, it's no different than any other object. When you create an array like `['a', 'b', 'c']`, internally that array looks like this:

```
{  
  length: 3,
```

```
0: 'a',  
1: 'b',  
2: 'c'  
}
```

Of course, it will also inherit several methods such as `push`, `pop`, `map`, and so on from `Array.prototype`. When performing iterative operations like `map` and `forEach`, the array will internally look at its `length` and then check itself for any properties within the range starting at zero and ending at `length`.

When you create an array via `new Array(9)`, many developers believe that internally the array will look like this:

```
{  
  length: 9,  
  0: undefined,  
  1: undefined,  
  2: undefined,  
  3: undefined,  
  4: undefined,  
  5: undefined,  
  6: undefined,  
  7: undefined,  
  8: undefined  
}
```

But in fact it will internally look like this:

```
{  
  length: 9  
}
```

It will have a `length` of 9 but it won't have the actual nine values. These missing values are called *holes*. Methods like `map` don't work on holes, so that's why your attempt at creating an array with nine spaces didn't work. A new method called `fill` fills an array with a specified value. When filling the array, it doesn't care if at a given index there's a value or a hole, so it will work

perfectly for your use:

```
const board = new Array(9).fill(' ')
```

We've covered several ways to construct arrays containing the values you want. Now let's look at some new methods of searching for those values once the arrays are constructed.

Quick check 9.3

Q1:

What's the difference between `arrayA` and `arrayB`?

```
let arrayA = new Array(9).map(function() { return 1 });  
let arrayB = new Array(9).fill(1);
```

QC 9.3 answer

A1:

The variable `arrayB` accurately creates an array with nine 1s. The variable `arrayA` does not because it does not actually contain any values to be mapped.

9.4. SEARCHING IN ARRAYS WITH `Array.prototype.includes`

You learned in [lesson 6](#) that strings have a new method on their prototype called `includes` for determining whether a string contains a value. Arrays have also been given this method, and it works similarly as seen in the next listing.

Listing 9.4. Using `includes` to check whether an array contains a value

```
const GRID = 'grid';
const LIST = 'list';
const availableOptions = [GRID, LIST];

let optionA = 'list';
let optionB = 'table';

availableOptions.includes(optionA);      1
availableOptions.includes(optionB);      2
```

- **1 true**
- **2 false**

With the `String.prototype.includes` method, you're checking if a string contains a substring. `Array.prototype.includes` works similarly, but you're checking whether any of the values at any of the array's indices are the value you're checking against.

Previously `indexOf` was used for determining if a value was in an array. For example:

```
availableOptions.indexOf('list');      1
```

- **1 1**

This works fine, but often led to bugs if the developer forgot that they needed to compare the result to `-1`, not truthiness. If the given value were at the `0` index, they

would get back 0, a falsy value, and vice versa: if the value wasn't found they would get back -1, a truthy value. But this gotcha can be avoided now by using `includes`, which returns a Boolean instead of an index.

Truthiness and falsiness

As a refresher, the idea of *falsy* values are any values that evaluate to false: `false`, `undefined`, `null`, `NaN`, `0`, and the empty string `""`. Truthy values are values that evaluate to true. Any value not previously listed as falsy would be truthy, including negative numbers.

9.5. SEARCHING IN ARRAYS WITH `ARRAY.PROTOTYPE.FIND`

Imagine you have an array of records cached from a database. When a request is made for a record (by its ID), you want to check the cache first to see if it has the record and return it before hitting the database again. You may end up writing code that resembles the following listing.

Listing 9.5. `filter` method returns all matches even if you only want one

```
function findFromCache(id) {  
  let matches = cache.filter(function(record) {  
    return record.id === id;  
  });  
  if (matches.length) {  
    return matches[0];  
  }  
}
```

The `findFromCache` function from [listing 9.5](#) would certainly work, but what happens when the cache has 10,000 records and it finds the one it's looking for at only the 100th try? In its current implementation, it would still check the remaining 9,900 records before returning the one it found. This is because the purpose of the `filter` function is to return all the records that match. You're only expecting one record to match, and once found you just need that record back. That's exactly what `Array.prototype.find` does: it searches through an array much like `Array.prototype.filter`, but as soon as it finds a match, it immediately returns that match and stops searching the array.

You can rewrite your `findFromCache` function to make use of the `find` function like so:

```
function findFromCache(id) {  
  return cache.find(function(record) {  
    return record.id === id;  
  });  
}
```

Another nice thing about `find` is that because it returns the item that matched instead of an array of matches, you don't have to pull it out of the array afterward; in fact, you can return the result of `find` directly.

Quick check 9.4

Q1:

In the following snippet what will the variable `result` be set to?

```
let result = [1, 2, 3, 4].find(function(n) {  
  return n > 2;  
});
```

QC 9.4 answer

A1:

The variable `result` will be set to 3, not `[3, 4]` because 3 is the first value that meets the criteria.

SUMMARY

In this lesson, the object was to teach you the most useful new methods being added to arrays.

- `Array.from` creates an array containing all the values from an array-like object.
- `Array.of` creates an array with supplied values and is safer than `new Array`.
- `Array.prototype.includes` checks whether an array contains a value at any of its indices.
- `Array.prototype.find` searches an array based on a criteria function and returns the first value found.
- `Array.prototype.fill` fills an array with a specified value.

Let's see if you got this:

Q9.1

Implement the function from the priming exercise. Don't worry about recreating all of jQuery. Just write your own implementation so that the following snippet works:

```
$('#a').css('background', 'yellow').css('font-weight', 'bold');
```

Lesson 10. Object.assign

After reading lesson 10, you will

- Know how to set default values with `Object.assign`
- Know how to extend objects with `Object.assign`
- Know how to prevent mutations when using `Object.assign`
- Understand how `Object.assign` assigns values

Libraries like Underscore.js and Lodash.js have become like the utility belts of JavaScript. They add solutions to common tasks so that the average developer doesn't need to reinvent the wheel for every project. Some of these tasks become so ubiquitous and well-defined that it makes sense to actually include them in the language. `Object.assign` is one such method. With it you can easily set default values or extend objects in place or as copies, all without having to write a lot of boilerplate code to do so.

Consider this

JavaScript objects can inherit properties from another object via the prototype chain. But any given object can only have one prototype. How would you go about devising a way to allow objects to inherit from multiple source objects?

10.1. SETTING DEFAULT VALUES WITH `Object.assign`

Let's pretend your favorite local pizza shop contacts you to build a pizza-tracking website. The idea is to use GPS to allow customers to see exactly where the pizza they ordered is as it's being delivered. You explain that you love their pizza and with your help you can put them on the map! As you set out to build this, you decide you need a function for generating the map from the pizza shop to the delivery address. The function would take an `options` argument for specifying the width, height, and coordinates of the map. The problem is that if any of these values aren't set, you need to use sane default values. One approach is to do something like the following listing.

Listing 10.1. Basic pizza-tracking map

```
function createMap(options) {  
  
  const defaultOptions = {  
    width: 900,  
    height: 500,  
    coordinates: [33.762909, -84.422675]  
  }  
  
  Object.keys(defaultOptions).forEach(function(key) {  
    if ( !(key in options) ) {  
      options[key] = defaultOptions[key];  
    }  
  });  
  
  // ...  
}
```

This implementation has a lot going on just assigning the default values before it can even get to the actual problem it's solving. Reimplement the default values using `Object.assign`:

```
function createMap(options) {  
  
  options = Object.assign({  
    width: 900,  
    height: 500,  
    coordinates: [33.762909, -84.422675]  
  }, options);  
  
  // ...  
}
```

Using `Object.assign` is much more terse than your initial approach. `Object.assign` works by taking any number of objects as arguments and assigning all the values from subsequent objects onto the first (see [figure 10.1](#)). If any objects contain the same keys, the object to the right will override the object to the left in the list, like so:

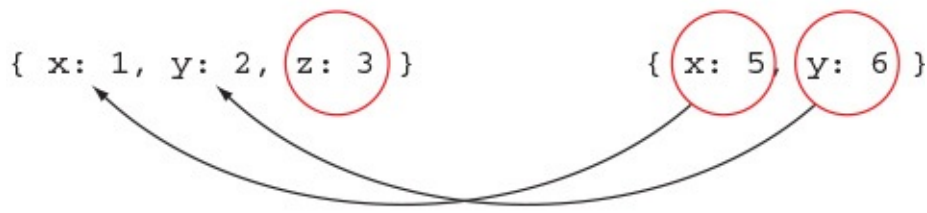
```
let a = { x: 1, y: 2, z: 3 };  
let b = { x: 5, y: 6 };  
let c = { x: 12 };  
  
Object.assign(a, b, c);  
  
console.log(a);
```

1

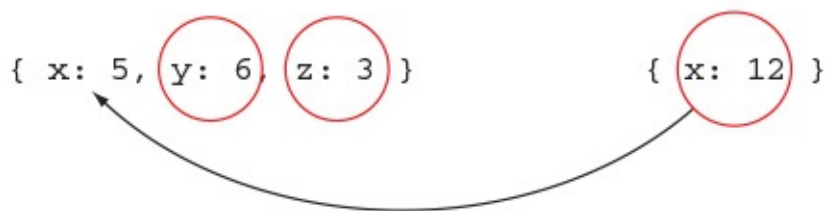
- **1 { x: 12, y: 6, z: 3 }**

Figure 10.1. Assigning values with `Object.assign`

First b gets assigned to a, overriding any existing properties:



Then c gets assigned to a, overriding any existing properties:



The final result is that a becomes:

```
{ x: 12, y: 6, z: 3 }
```

b and c are left unchanged.

When setting defaults like this with `Object.assign`, you're basically taking objects that are incomplete and filling in the missing values. Another common task is taking objects that are already complete and extending them into more customized variations.



Quick check 10.1

Q1:

In the following use of `Object.assign`, what are a and b set to afterward?

```
let a = { city: 'Dallas', state: 'GA' };  
let b = { state: 'TX' };
```

```
Object.assign(a, b);
```

QC 10.1 answer

A1:

a is set to { city: 'Dallas', state: 'TX' }
and b is unchanged.

10.2. EXTENDING OBJECTS WITH OBJECT.ASSIGN

Another common use case for `Object.assign` is to add some additional properties to an existing object. Let's say you're building a spaceship game. The game has a base spaceship and several other specialized spaceships. The specialized spaceships all have the same basic features as the base spaceship, but with an extra or enhanced skill. You start with a function that creates a base spaceship:

```
function createBaseSpaceShip() {
  return {
    fly: function() {
      // ... fly function implementation
    },
    shoot: function() {
      // ... shoot function implementation
    },
    destroy: function() {
      // ... function for when the ship is destroyed
    }
  }
}
```

Then to create a specialized spaceship, you create a copy of a base spaceship and extend it with the specialized features:

Listing 10.2. Adding a bomb property to the spaceship

```
function createBomberSpaceShip() {
  let spaceship = createBaseSpaceShip();

  Object.assign(spaceship, {
    bomb: function() {
      // ... make the ship drop a bomb
    }
  });

  return spaceship;
}

let bomber = createBomberSpaceShip();
bomber.shoot();
bomber.bomb();
```

First you create a base spaceship object. Then you use `Object.assign` to add an additional bomb property. You could create several enhanced spaceships using this technique. This function does have a lot of boilerplate, though, so if you were going to reuse this technique, you could create a helper function for enhancing the base spaceship.

Listing 10.3. Helper function for enhancing spaceships

```
function enhancedSpaceShip(enhancements) {
  let spaceship = createBaseSpaceShip();

  Object.assign(spaceship, enhancements);

  return spaceship;
}

function createBomberSpaceShip() {
  return enhancedSpaceShip({
    bomb: function() {
      // ... make the ship drop a bomb
    }
  });
}

function createStealthSpaceShip() {
  return enhancedSpaceShip({
    stealth: function() {
      // ... make the ship invisible
    }
  });
}

let bomber = createBomberSpaceShip();bomber.shoot();
bomber.bomb();

let stealthship = createStealthSpaceShip();
stealthship.shoot();
stealthship.stealth();
```

Now you've created an `enhancedSpaceShip` function that takes an object of enhancements as an argument. These enhancements could be new features or could override existing features. Notice how in every example so far, you've created an object and then mutated it? Sometimes that's desired, but often you just want a copy.

Quick check 10.2

Q1:

Use `Object.assign` to create a spaceship that extends the base with a `warp` function.

QC 10.2 answer

A1:

```
function createWarpSpaceShip() {  
  return enhancedSpaceShip({  
    warp: function() {  
      // ... make the ship go warp speed  
    }  
  });  
}
```

10.3. PREVENTING MUTATIONS WHEN USING `Object.assign`

Oftentimes when making modifications to objects, programmers like the operation to not mutate the existing object but instead return a copy with the desired changes. One reason for this is to prevent anything else that currently holds a reference to the object from getting its reference changed out from under it. For example, what if in your spaceship example you didn't have a function for creating a new copy of the base spaceship object? What if you just had a single base spaceship object? If that were the case, every time you called `Object.assign` to enhance it, you would be enhancing the base, which you don't want. Instead you would want to make a copy and leave the base unchanged. You may think that `const` can help you here, but it can't.

Remember, `const` only prevents overriding a variable with a new object; it doesn't prevent modifying/mutating the existing object.

Because `Object.assign` mutates the first object in the parameter list, and leaves the rest of the objects unchanged, a common practice is to make the first object an empty object literal:

```
let newObject = {};  
Object.assign(newObject, {foo: 1}, {bar: 2});  
console.log(newObject);
```

1

- **1 {foo: 1, bar: 2}**

When doing this, it's inconvenient to first create the empty object, then pass it to `Object.assign`.

Conveniently, `Object.assign` also returns the mutated

object, so you can shorten it like so:

```
let newObject = Object.assign({}, {foo: 1}, {bar: 2});  
console.log(newObject);
```

1

- **1 {foo: 1, bar: 2}**

Rewrite your spaceship example to use a single base object and your enhance function to make an enhanced copy, as shown in the next listing.

Listing 10.4. Copying the spaceship

```
const baseSpaceShip = {  
  fly: function() {  
    // ... fly function implementation  
  },  
  shoot: function() {  
    // ... shoot function implementation  
  },  
  destroy: function() {  
    // ... function for when the ship is destroyed  
  }  
}  
  
function enhancedSpaceShip(enhancements) {  
  return Object.assign({}, baseSpaceShip, enhancements);  
}  
  
function createBomberSpaceShip() {  
  return enhancedSpaceShip({  
    bomb: function() {  
      // ... make the ship drop a bomb  
    }  
  });  
}  
  
function createStealthSpaceShip() {  
  return enhancedSpaceShip({  
    stealth: function() {  
      // ... make the ship invisible  
    }  
  });  
}
```

- **1 By passing {} as the first argument, Object.assign makes a copy.**

Notice that, because you abstracted away your

enhancement technique into its own function, when you modified how it works, you didn't have to make any changes to the `createBomberSpaceShip` or `createStealthSpaceShip` functions.

At this point you may be thinking that `Object.assign` basically merges objects from right to left. Well, not exactly; it only assigns values from the left to the right. This minor distinction is what we'll discuss next.

Quick check 10.3

Q1:

Write a function called `createStealthBomber` that creates a spaceship that's a combination of the bomber and the stealth spaceship using `Object.assign`.

QC 10.3 answer

A1:

```
function createStealthBomber() {  
  const stealth = createStealthSpaceShip();  
  const bomber = createBomberSpaceShip();  
  return Object.assign({}, stealth, bomber);  
}
```

10.4. HOW OBJECT.ASSIGN ASSIGNS VALUES

There's a subtle difference between definition and assignment. Assigning a value to an existing property doesn't define how a property behaves, the way `Object.defineProperty` does. Assignment only falls back to defining a property if an assignment is made to a property that doesn't exist. This is important because `Object.assign` doesn't define (or redefine) properties; it assigns to them. Most of the time this distinction makes no difference, but sometimes it does. Let's explore the difference:

```
let person = {
  name: 'JD'
}

Object.assign(person, { name: 'JD Isaacks' })
```

The preceding function works as expected. But what if the name property wasn't a mere string but a getter and setter that ensured that the name contain no spaces?

```
let person = {
  __name: 'JD',
  get name() {
    return this.__name
  },
  set name(newName) {
    if (newName.includes(' ')) {
      throw new Error('No spaces allowed!')
    } else {
      this.__name = newName
    }
  }
}

person.name = 'JD Isaacks' 1
```

- **1 Error: No spaces allowed!**

As you can see, if you assign a string to name, it gets

processed by the setter function and if the string contains a space, it throws an error. The same thing happens if you use `Object.assign` because it merely makes the assignments; it doesn't redefine the property:

```
Object.assign(person, { name: 'JD Isaacks' })    1
```

- **1 Error: No spaces allowed!**

Additionally, `Object.assign` only assigns an object's own enumerable properties, meaning it won't assign properties that are on an object's prototype chain or properties set to non-enumerable. This is usually a good thing. Here's an example illustrating what I mean:

```
let numbers = [1, 2, 3]
let summableNumbers = Object.assign({}, numbers, {
  sum: function() {
    return this.reduce(function(a, b) {
      return a + b
    })
  }
})

summableNumbers[0]    1
summableNumbers.sum() 2
```

- **1 1**
- **2 TypeError: this.reduce is not a function.**

Why did you get an error? Because the `reduce` method is on the `numbers` array's prototype chain, not directly on the `numbers` array itself. The only properties directly on the `numbers` array are `0`, `1`, and `2` for the three values it contains. Because of this, those three values got assigned to the new object, but all the methods on the array prototype such as `pop`, `push`, and `reduce` did not. You could make this work, however, by assigning to an empty array which already has all the methods it needs:

```
let summableNumbers = Object.assign([], numbers, {  
  1  
  // ...  
})
```

- **1 Assigning to an empty array rather than an empty object**

Most of the time these issues won't be a concern, because more often than not you'll be using `Object.assign` on what many developers like to call a *POJO* (plain old JavaScript object), an object without things like getters, setters, or a prototype. This is the type of object you create with the object literal `{ ... }` syntax.

Quick check 10.4

Q1:

What will be logged to the console?

```
let doubleTheFun = {  
  __value: 1,  
  get value() {  
    return this.__value  
  },  
  set value(newVal) {  
    this.__value = newVal * 2  
  }  
}  
console.log( Object.assign(doubleTheFun, { value: 2  
}).value );
```

QC 10.4 answer

A1:

4

SUMMARY

In this lesson you learned the basics of how to use `Object.assign` and why you would do so.

- `Object.assign` assigns values properties from several objects onto a base object.
- `Object.assign` can be used to fill in default values.
- `Object.assign` can be used to extend objects with new properties.
- `Object.assign` returns the mutated object so that using `{}` as the first argument will create a copy.
- `Object.assign` only assigns to properties; it doesn't redefine them.

Let's see if you got this:

Q10.1

In this lesson you created a base spaceship and made several other spaceships inherit from it. You even made a stealth bomber that inherited from multiple other spaceships. Come up with another series of objects that inherit from one another, such as passenger vehicles, the animal kingdom, video game characters, or anything else you can think of.

Lesson 11. Destructuring

After reading lesson 11, you will

- Know how to destructure objects
- Know how to destructure arrays
- Know how to combine array and object destructuring
- Understand what types can be destructured

JavaScript, like other programming languages, has data structures like objects and arrays that allow you to structure data into logical groups and treat them as a single piece of data. JavaScript has always supported the concept of *structuring*, which is a concise syntax for taking several pieces of data and arranging them into a data structure. But that's only half of the story: JavaScript has always been missing the reverse, a syntax for taking an existing data structure and *destructuring* it back into the parts that make it up.

Consider this

In the following code, there are two sections. The first section builds a data structure and the second section breaks it down. You can probably already think of a way to clean up the creation of the data structure using an object literal, but how would you clean up the teardown?

```
// Creating data structures without structuring
let potus = new Object();
potus.name = new Object();
potus.name.first = 'Barack';
potus.name.last = 'Obama';
potus.address = new Object();
potus.address.street = '1600 Pennsylvania Ave NW';
potus.address.region = 'Washington, DC';
potus.address.zipcode = '20500';

// Breaking down data structures without destructuring
```

```
let firstName = potus.name.first;  
let lastName = potus.name.last;  
let street = potus.address.street;  
let region = potus.address.region;  
let zipcode = potus.address.zipcode;
```



11.1. DESTRUCTURING OBJECTS

Imagine how tedious it would be to write an application without ever using an object literal. You may think you'd quit writing JavaScript before doing something like that. But you've been doing the same tedious process when breaking down objects, and once you see how nice it is to destructure objects the same way you structure them, you'll never want to go back. Let's start by taking a look at a simple example:

```
let person = {                                1
  name: 'Christina'
}

let { name } = person;                        2

console.log(name);                            3
```

- **1 Creating a data structure**
- **2 Destructuring a data structure**
- **3 Christina**

In this example you created an object in a structured manner using an object literal. By doing so, you were able to specify the properties you were setting on the structure in a concise way. Later you destructured that data structure with a destructuring statement. You specified that you wanted to extract the name property into a variable.

In this simple example, the benefits aren't that great, but as usual, the more complex the use case, the more obvious it becomes how useful destructuring is:

```
let person = {
  name: 'Christina',
  age: 25
}
```

```
let { name, age } = person;  
  
console.log(name, age);
```

1

- 1 “Christina” 25

When you destructure an object like this, you’re specifying the names of the fields in the object that you’re extracting. Variables by the same names are then assigned the values. The ES5 equivalent would look like this:

```
let name = person.name;  
let age = person.age;
```

But what if you wanted to use different names?

```
let firstName = person.name;  
let yearsOld = person.age;
```

When destructuring, you can specify different names to use like so:

```
let { name: firstName, age: yearsOld } = person;
```

This creates the variables `firstName` and `yearsOld`, and assigns them the values from `person.name` and `person.age` respectively. This is nice because it’s exactly the reverse of how you would build the data structure:

```
let person = { name: firstName, age: yearsOld };
```

You can do nested destructuring to drill down into the data structure to extract data at different levels of the data structure:

```
let geolocation = {  
  "location": {  
    "lat": 51.0,
```



```
    "lng": -0.1
  },
  "accuracy": 1200.4
}

let { location: {lat, lng}, accuracy } = geolocation;

console.log(lat); // 51.0
console.log(lng); // -0.1
console.log(accuracy); // 1200.4
console.log(location); // undefined
```

With the syntax key: {otherKeys} you are specifying that you are drilling down into the specified key to extract the otherKeys from within it. Here you specified that you wanted to extract the lat and lng properties from the location property. Also notice how there was no creation of a location variable: you merely drilled into it.

Quick check 11.1

Q1:

How would you rewrite the previous destructuring statement to assign the lng property into a variable called lon?

QC 11.1 answer

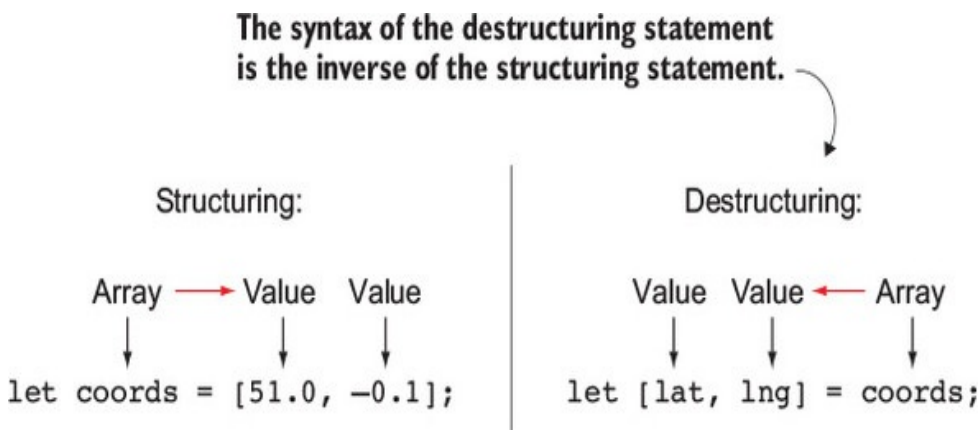
A1:

```
let {location: {lat, lng: lon}, accuracy} = geolocation;
```

11.2. DESTRUCTURING ARRAYS

In the previous section you learned that the syntax for object destructuring is the inverse of the syntax for object structuring. Similarly, the syntax for array destructuring is the inverse of the syntax for structuring arrays. See [figure 11.1](#).

Figure 11.1. Structuring and destructuring syntax



```
let coords = [51.0, -0.1];
```

```
let [lat, lng] = coords;
```

```
console.log(lat);
```

1

```
console.log(lng);
```

2

- 1 51.0

- 2 -0.1

Because objects track their values by keys or names, they must be extracted by those same keys/names. And since arrays track their values by what index they're positioned in, it makes sense that when destructuring an array the position of the variable is how you specify what you're destructuring.

Just like with destructuring objects, you can do nested destructuring with arrays as well:

```
let location = ['Atlanta', [33.7490, 84.3880]];

let [ place, [lat, lng] ] = location;

console.log(place); // "Atlanta"
console.log(lat); // 33.7490
console.log(lng); // 84.3880
```

One idiom I've grown fond of is using array destructuring to grab the first value from an array:

```
let [firstValue] = myArray;

// vs

let firstValue = myArray[0];
```

In the lesson on `let`, you reversed the values between two variables, and in doing so, you needed to create a third temporary variable to prevent the value from being lost before reversal:

```
if (stop < start) {
  // reverse values
  let tmp = start;
  start = stop;
  stop = tmp;
}
```

You can achieve this same thing without the need for a temporary variable by using destructuring:

```
if (stop < start) {
  // reverse values
  [start, stop] = [stop, start];
}
```

Quick check 11.2

Q1:

What if the array in the previous example stored the `lat`/`lng` values in reverse order? How could you

extract them to the correct variables?

QC 11.2 answer

A1:

Since array destructuring is tied to indices, not names, you can simply reverse the names order in the destructuring statement:

```
let location = ['Atlanta', [84.3880, 33.7490]];
let [ place, [lng, lat] ] = location;
```

11.3. COMBINING ARRAY AND OBJECT DESTRUCTURING

You can combine the destructuring statements for objects and arrays just as you can combine the use of object and array literals when creating data structures:

```
let geoResults = {  
  coords: [51.0, -0.1],  
  ...  
}  
  
let { coords: [ latitude, longitude ] } = geoResults;  
  
console.log(latitude, longitude)      1
```

• 1 51.0 -0.1

This is also a form of nested destructuring, because you're using object destructuring to drill down into the array destructuring.

Also note that in this example, you used the name `coords` in your destructuring assignment but you didn't create a variable named `coords`. You only created the `latitude` and `longitude` variables. The use of the name `coords` was only to drill down and specify the array you were destructuring. You can think of `coords` as a branch and `latitude` and `longitude` as leaves in the destructuring assignment. The branches won't create variables; only the leaves will. Consider the following code:

```
let product = {  
  name: 'Whiskey Glass',  
  details: {  
    price: 18.99,  
    description: 'Enjoy your whiskey in this glass'  
  },  
  images: {  
    primary: '/images/main.jpg',  
  }  
}
```

```

    others: [
      '/images/1.jpg',
      '/images/2.jpg'
    ]
  }
}

let {
  name,
  details: { price, description},
  images: {
    primary,
    others: [secondary, tertiary]
  }
} = product;

console.log(name);           1
console.log(price);         2
console.log(description);    3
console.log(primary);        4
console.log(secondary);      5
console.log(tertiary);       6

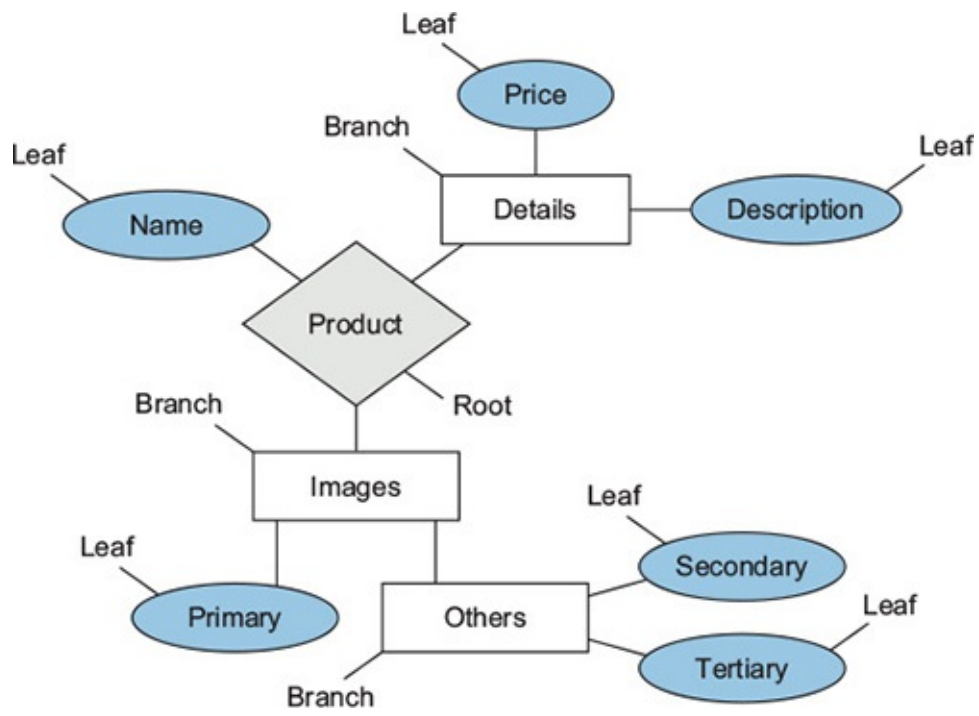
console.log(details);        7
console.log(images);         7
console.log(others);         7

```

- **1 Whiskey Glass**
- **2 18.99**
- **3 Enjoy your whiskey in this glass.**
- **4 /images/main.jpg**
- **5 /images/1.jpg**
- **6 /images/2.jpg**
- **7 undefined**

In the preceding destructuring assignment, if you mark which values are used as branches and which are leaves, you'll know which variables got created. See [figure 11.2](#).

Figure 11.2. Destructuring a product array



11.4. WHAT TYPES CAN BE DESTRUCTURED

You can use object destructuring on any object, even arrays. But since numbers aren't valid variable names, you would need to rename them in the destructuring assignment:

```
const { 0:a, 1:b, length } = ['foo', 'bar']
```

```
console.log(a)           1  
console.log(b)           2  
console.log(length)      3
```

- **1 foo**
- **2 bar**
- **3 2**

Using array destructuring is more restrictive, but can be used on much more than just arrays. Any object that implements the iterable protocol (which we'll cover in a later unit) can be destructured using array destructuring. One example of an object that's iterable is a `Set`, which we'll cover later in this book. Another example is a `String`:

```
const [first, second, last] = 'abc'
```

```
console.log(first)        1  
console.log(second)       2  
console.log(last)         3
```

- **1 a**
- **2 b**
- **3 c**

Later in this book when you make our own iterables, you'll be able to use array destructuring on them as well!

SUMMARY

In this lesson you learned the mechanics of destructuring and why it's such a useful technique.

- Destructuring is a syntactic sugar for retrieving values from data structures.
- Destructuring is the inverse complement to how object/array literals provide a syntax for structuring.
- With object destructuring, you specify values by property name.
- With array destructuring, you specify values by their index.
- Destructuring can be combined and nested just like data structures.
- When nesting destructuring, only the leaves (not branches) are retrieved.

Let's see if you got this:

Q11.1

You should now be able to convert the code from the priming exercise into a single destructuring statement:

```
let firstName = potus.name.first;  
let lastName = potus.name.last;  
let street = potus.address.street;  
let region = potus.address.region;  
let zipcode = potus.address.zipcode;
```

Additionally, try to convert this code into a single destructuring statement:

```
let firstProductName = products[0].name  
let firstProductPrice = products[0].price  
let firstProductFirstImage = products[0].images[0]  
  
let secondProductName = products[1].name  
let secondProductPrice = products[1].price  
let secondProductFirstImage = products[1].images[0]
```

Lesson 12. New object literal syntax

After reading [lesson 12](#), you will

- Know how to use shorthand property names
- Know how to use shorthand method names
- Know how to use computed property names

I don't think there's anything as ubiquitous in JavaScript as the object literal. They're everywhere. With a tool that's used so frequently, any conveniences can have a huge net positive on productivity. Three syntactic additions to object literals introduced in ES2015 make them much more of a pleasure to read and write. You don't gain new functionality, but having code that's easy to read and write, especially when it comes to maintenance, is just as important a feature.

Consider this

Look at the following object literal. What parts seem redundant? If you were writing a JavaScript-aware string compression engine, what parts do you think you could safely remove while still being able to reconstruct the original object?

```
const redundant = {  
  name: name,  
  address: address,  
  getStreet: function() { /* ... */ },  
  getZip:    function() { /* ... */ },  
  getCity:   function() { /* ... */ },  
  getState: function() { /* ... */ },  
  getName:   function() { /* ... */ },  
}
```

12.1. SHORTHAND PROPERTY NAMES

Before ES6 there were countless times when I would inevitably create an object literal that had a key or property whose name was the same as the variable I was assigning to it:

```
const message = { text: text }           1
```

- **1 Assigning the property text to a variable also named text**

I always thought this looked awkward, and thanks to ES6 shorthand property names, that's a thing of the past. The previous object literal can now be concisely written as

```
const message = { text }                 1
```

- **1 Assigning the property text to a variable also named text**

This also works fantastically when coupled with destructuring:

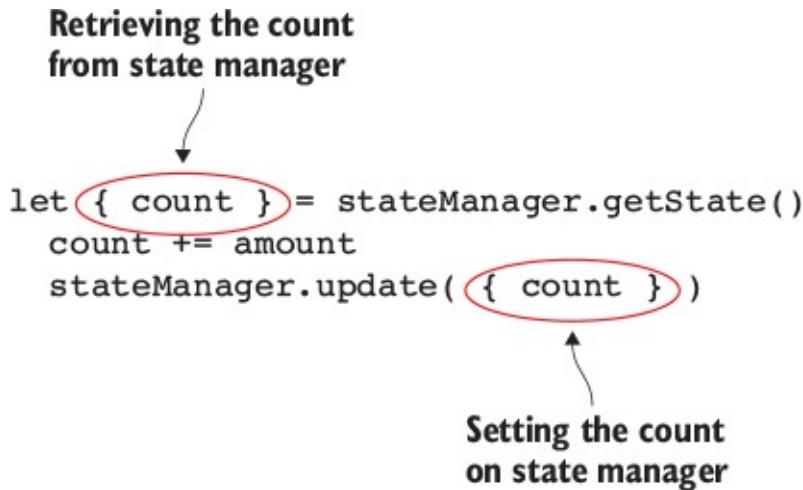
```
function incrementCount(amount) {  
  let { count } = stateManager.getState()    1  
  count += amount                           2  
  stateManager.update({ count })            3  
}
```

- **1 Extracting count from the object returned from getState**
- **2 Updating the values of count**
- **3 Passing an object literal with a property named count set to the new count**

With shorthand property names, if the name of the property is the same as the name of the variable you're assigning to it, you can simply list the name once. That means that an object literal that would have been written as `{ count: count }` can now be written as `{ count`

} and will evaluate to the same thing (see [figure 12.1](#)).

Figure 12.1. Using shorthand property names



Notice that you now have symmetry between where you're grabbing your count with { count } and then passing it back with { count }. I personally find this symmetry extremely elegant.

Shorthand properties can be mixed and matched with any other property or method in an object literal. In the next listing the shorthand and longhand object literals are equivalent.

Listing 12.1. Mixing shorthand and longhand object literals

```
const lat = 33.762909;  
const lng = -84.422675;  
const accuracy = 1200.4;  
  
const shorthand = {  
  name: 'Atlanta',  
  accuracy,  
  location: {  
    lat,  
    lng  
  }  
};  
  
const longhand = {  
  name: 'Atlanta',  
  accuracy: accuracy,  
  location: {  
    lat: lat,  
    lng: lng  
  }  
}
```

```
}
```

Concise property names take the redundancy out of object literals with matching key/value names. You know what else is redundant in object literals? Methods.

Quick check 12.1

Q1:

Rewrite the following object literal to make use of shorthand property names.

```
let foo, bar, bizz, bax = {  
  foo: foo, bar: bar,  
  biz: bizz  
}
```

QC 12.1 answer

A1:

```
let foo, bar, bizz, bax = {  
  foo, bar, biz: bizz  
}
```

12.2. SHORTHAND METHOD NAMES

Let's assume your team is building a game that teaches math to elementary students. The game must track several pieces of state (such as the name of the player, current level, number of correct/incorrect answers, which questions have been asked, and so forth) across the game session. You may start with a simple state manager like the following:

```
function createStateManager() {  
  const state = {};  
  return {  
    update: function(changes) {  
      Object.assign(state, changes);  
    },  
    getState: function() {  
      return Object.assign({}, state);  
    }  
  }  
}
```

The use of the word `function` is redundant here. Since functions always have a parameter list `()`, and since parentheses aren't valid in variable or property names, their presence should be enough to signify that it's a method. Shorthand method names allow for doing just that.

This is just like the way shorthand property names remove the need to type `: redundant name`, and `{ count: count }` becomes `{ count }`. Shorthand method names remove the need to type `: redundant function`, and `{ update: function(){} }` is shortened to `{ update(){} }`. See [figure 12.2](#).

Figure 12.2. Shorthand method names remove redundant syntax.

```
const verbose = {
  method: function() {
    //....
  },
  name: name
}

const concise = {
  method() {
    //...
  },
  name
}
```

Concise methods
remove : function

Concise properties
remove : name

With that in mind, you can update your `createStateManager` to use shorthand method names:

```
function createStateManager() {
  const state = {};
  return {
    update(changes) {
      Object.assign(state, changes);
    },
    getState() {
      return Object.assign({}, state);
    }
  }
}
```

It's important to understand that shorthand methods evaluate to *anonymous* functions, not *named* functions. This means that you can't refer to the function by name within itself:

```
const fibonacci = {
  at(n) {
    if (n <= 2) return 1;
    return at(n - 1) + at(n - 2);
  }
}

fibonacci.at(7)      1
```

- **1 ReferenceError: at is not defined**

In the next listing, the `withShorthandFunction` object literal evaluates (or is *desugared*) similar to the `withAnonymousFunction` object, and not like the `withNamedFunction` object.

Listing 12.2. Shorthand methods are anonymous functions

```
const withShorthandFunction = {
  fib() {
    // ...
  }
}
const withNamedFunction = {
  fib: function fib() {
    // ...
  }
}
const withAnonymousFunction = {
  fib: function() {
    // ...
  }
}
```

This really only matters when the function is self-referencing, meaning the function is making a reference to itself, as with *recursion*. Because of this, defining the `at` method using shorthand syntax wouldn't work. You may be tempted to solve this problem using `this.at` as shown in [listing 12.3](#), which would work as long as the function is always invoked in a context in which `this.at` will resolve back to the function. If the function becomes detached or attached under a different name, it will no longer work.

Listing 12.3. Recursive function with `this.at`

```
const fibonacci = {
  at(n) {
    if (n <= 2) return 1;
    return this.at(n - 1) + this.at(n - 2);
  }
}
const { at } = fibonacci;
const fib = { at };
const nacci = { find: at };
```


fibonacci.at(7);	1
at(7);	2
fib.at(7);	3
nacci.find(7);	4

- **1 13**
- **2 Throws ReferenceError**
- **3 13**
- **4 Throws ReferenceError**

At the end of the day, if you need to self-reference, don't use shorthand method names.

Concise properties and methods make object literals less redundant. In the next section you'll remove even more cruft with computed property names.

Quick check 12.2

Q1:

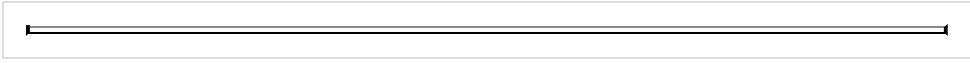
Rewrite the following object literal to make use of shorthand method names:

```
const dtslogger = {  
  log: function(msg) {  
    console.log(new Date(), msg);  
  }  
}
```

QC 12.2 answer

A1:

```
const dtslogger = {  
  log(msg) {  
    console.log(new Date(), msg);  
  }  
}
```



12.3. COMPUTED PROPERTY NAMES

Suppose you're using your state manager and need to make it work with another library that's going to give you values you need to store in state. The problem is that this library doesn't give you the key/value pairs in the form of an object that you can pass to the state manager's `update` function. Instead, this other library gives you the key/value pair in the form of an array where the first index is the key and the second index is the value. To make these two libraries play nicely together, you write an intermediate function called `setValue` that takes a name and a value as parameters and internally turns them into an object that it passes to `stateManager.update`:

```
function setValue(name, value) {  
  const changes = {};  
  changes[name] = value;  
  stateManager.update(changes);  
}  
  
const [name, value] = otherLibrary.operate();  
setValue(name, value);
```

Notice how in the `setValue` function you must first create an empty object before defining the property. This is because the name of the property is dynamic: if we did something like `{ name: value }` then the property name would actually be the value `"name"`, instead of the value the `name` variable contains. This would be similar to `changes.name = value`, which again would set the property name to `name`. Instead you use `changes[name]` to use the actual value contained in `name`.

The same syntax that you use to add a dynamically named property to an object after it has been created can now be used to compute the property name in an object literal. This syntax is concise and seems to tell the story of what it's doing much better than your original code:

```
function setValue(name, value) {
  stateManager.update({
    [name]: value
  });
}

const [name, value] = otherLibrary.operate();
setValue(name, value);
```

This works exactly the same as using brackets on objects after they're created, and allows you to use the same syntax with object literals.

Quick check 12.3

Q1:

What will be logged to the console?

```
const property = 'color';
const eyes = { [property]: 'green' };
console.log(eyes.property, eyes.color);
```

QC 12.3 answer

A1:

undefined “green”

SUMMARY

In this lesson you learned how to take advantage of the additions to the object literal syntax.

- Concise property names remove redundancy when the key and value have the same name.
- Concise methods give a simpler syntax for defining functions on an object literal.
- Concise methods are anonymous and shouldn't be used for recursion.
- Computed properties allow you to use dynamic property names on object literals.

Let's see if you got this:

Q12.1

Use shorthand method names to extend your state manager to support subscribing and unsubscribing to state changes.

Lesson 13. Symbol—a new primitive

After reading lesson 13, you will

- Know how to use symbols as constants
- Know how to use symbols as object keys
- Know how to create behavior hooks with global symbols
- Know how to modify object behavior with well-known symbols

In JavaScript there are *objects* and there are *primitives*. The primitives of JavaScript are strings, numbers, Booleans (true or false), null, and undefined. *Symbol* is a new primitive added in ES2015 and is the first primitive added to JavaScript since its creation. A symbol is a unique value that's used for hooking into the behavior of built-in JavaScript objects. Symbols can be broken into three categories:

1. Unique symbols
2. Global symbols
3. Well-known symbols

Consider this

Imagine you were programming a chess game. You program the base chess piece with a `moves()` function that determines the available destinations, then checks if each is legal by checking if the destination is occupied by a teammate piece. The code for each type of chess piece needs to inherit from the base chess piece code. How would you make each chess piece override how the `moves()` function determines the possible destinations without overriding how it determines if the move is legal?

13.1. USING SYMBOLS AS CONSTANTS

Developers often create constants to represent a *flag*. A flag is a special variable with only one purpose: identification. The flag's value isn't important, only that the flag is identified.

Many libraries use flags; for example, Google Maps uses the flags HYBRID, ROADMAP, SATELLITE, and TERRAIN to identify what type of map to draw. Each flag's value is the lowercase string version of its name (for example `HYBRID === "hybrid"`), but this doesn't matter as long as the flag can be identified.

In [listing 13.1](#) you're using constants as flags to identify where to place a tooltip. But the names are generic and could easily conflict with other values. Since the only purpose of a flag is to be identified, it makes sense to protect the flag from misidentification. Imagine another part of the program has flags for CORRECT and INCORRECT that contain the values `right` and `wrong` respectively. This means the CORRECT flag could be misidentified as the RIGHT flag since they both contain the same internal value.

Some libraries like Redux send actions through multiple handlers. Each handler identifies what actions it's responsible for by the use of flags and identifiers. Different plugins written by different authors can specify handlers for actions. This increases the possibility of naming collisions.

Listing 13.1. Using constants as flags

```
const answer = {  
  CORRECT : 'right',  
  INCORRECT : 'wrong'
```

```

}

const positions = {
  TOP    : 'top',
  BOTTOM : 'bottom',
  LEFT   : 'left',
  RIGHT  : 'right'
}

function addToolTip(content, position) {
  switch(position) {
    case positions.TOP:
      // add content above
      break;
    case positions.BOTTOM:
      // add content below
      break;
    case positions.RIGHT:
1      // add content to right
      break;
    case positions.LEFT:
      // add content to left
      break;
    default:
      throw new Error(`${position} is not a valid position`)
      break;
  }
}

addToolTip('You are not logged in', answer.CORRECT);
2

```

- **1 This will also match answer.CORRECT.**
- **2 This will position the tooltip to the right.**

Notice how the `addToolTip` function expects a flag from `positions` but you passed a flag from `answer`. This ideally should throw an error notifying you of your mistake, but it doesn't. It just positions the tooltip to the right because your flags aren't unique.

You can solve these naming collisions using unique symbols. Create a unique symbol by calling the `Symbol()` function with an optional string description as a parameter. Each unique symbol is always unique as demonstrated here:

<code>Symbol() === Symbol()</code>	<code>1</code>
<code>Symbol('right') === Symbol('right')</code>	<code>1</code>

- `1 false`

When you create a symbol using the `Symbol` function, the symbol created is always unique. You can call the `Symbol` function over and over again with the same description and every time you will get a unique symbol back. This is because when you call the `Symbol` function, you are creating a new value in memory every time that will never equate to another symbol in memory somewhere else.

The same string such as `"foo"` will only ever be stored in memory once. No matter how many times you create a new string `"foo"`, the primitive value is always pointing to the same place in memory. This is why `"foo" == "foo"` is always true. This is the same for all primitives, and is why primitives can always be compared to each other no matter where they came from. Objects on the other hand always create their own identify in memory, and thus two objects that look the same will not equate as the same `{ foo: 1 } != { foo: 1 }`.

Every time you invoke `Symbol()` you're creating a brand new value stored in memory so there's no chance of an accidental collision. Now that you know symbols created from the `Symbol()` function will never collide, you could rewrite your previous tool tip example to make use of symbols and capture the previous bug:

```
const positions = {  
  TOP    : Symbol('top'),  
  BOTTOM : Symbol('bottom'),  
  LEFT   : Symbol('left'),  
  RIGHT  : Symbol('right')  
}
```

This also has the benefit of making sure that the constant and not the value is used when invoking the function, because passing `Symbol(' top ')` as the position argument won't actually match any flags! This would make it easier to refactor your constants knowing that the values can't be directly used as strings can.

Quick check 13.1

Q1:

In the following snippet, of the values a through e, which are `true`?

```
const x = Symbol('x')
const y = Symbol('y')

const a = x === x;
const b = x === y;
const c = x === Symbol('x');
const d = y === y;
const e = y === Symbol('y');
```

QC 13.1 answer

A1:

Only a and d.

13.2. USING SYMBOLS AS OBJECT KEYS

Sometimes developers add properties to objects with a leading underscore to signify that they should be treated as pseudo-private, such as `Store._internals`. The problem with this is that (unless specified with `Object.defineProperty()`) these values will still be enumerable, meaning they'll still be included in things like `for...in` statements and `JSON.stringify()`, which may not be desired. You can achieve the same pseudo-private properties using symbol as the property name:

```
const Store = {  
  [Symbol('_internals')]: { /* ... */ }  
}
```

- **1 To use a symbol as a property name, it must be a computed property name**

You must use a *computed property name*, though, to add a symbol as a property name. When using a symbol as a property name, the property isn't enumerable, meaning it won't be included in a `for...in` statement or in `JSON.stringify()`, and it won't even be returned in `Object.getOwnPropertyNames()`. Also, since you don't have any references to the symbol, and since calling `Symbol('hidden')` again will return a different symbol, the property is essentially private.

It isn't really private, though, because there's a new function in ES6 on `Object` called `Object.getOwnPropertySymbols()` which returns an array of all the own properties that are symbols. With this you can get a reference to the symbol and access the

property.

If symbols as property names aren't truly private, and since they can be private with closures, what's the point? Setting a symbol property isn't meant to be private from a security point of view, but more of an available but hidden API. This not-quite-private but not-quite-public API is a great place to define hooks that modify the object, or *meta* properties.

Quick check 13.2

Q1:

What's wrong with this code?

```
const DataLayer = {  
  Symbol.for('fetch'): function() {  
    // fetch from database.  
  }  
}
```

QC 13.2 answer

A1:

The symbol property needs to be a computed property:

```
const DataLayer = {  
  [Symbol.for('fetch')]: function() {  
    // fetch from database.  
  }  
}
```

13.3. CREATING BEHAVIOR HOOKS WITH GLOBAL SYMBOLS

Global symbols are symbols that are added to a registry and can be globally accessed from anywhere. You access a global symbol by invoking `Symbol.for`:

```
const sym = Symbol.for('my symbol');
```

When you invoke `Symbol.for`, it first checks the registry to see if a symbol exists for the given name; if so, it's returned. If a symbol doesn't yet exist, it's created, added to the registry, and then returned. Since global symbols are global, it goes without saying that the symbols you get from `Symbol.for()` aren't unique like the symbols you get back from `Symbol()`. Global symbols are great for setting predefined hooks into an object's behavior.

In [lesson 10](#) you created a factory function for creating base spaceships. You then created other factory functions for creating more specialized spaceships. In that lesson you simply added new methods to the base like `bomb`, as shown in the next listing.

Listing 13.2. The `createBomberSpaceShip` function from [lesson 10](#)

```
function createBomberSpaceShip() {  
  return enhancedSpaceShip({  
    bomb: function() {  
      // ... make the ship drop a bomb  
    }  
  });  
}
```

But in a real game you most likely have a way for the player to fire the weapon, such as pressing the spacebar. It would make more sense to provide a method like

`fire()` that lets the enhanced spaceships hook into and alter what happens when the ship fires. This way when the user presses the spacebar, in response the code can invoke `fire()` without having to worry about what kind of spaceship it is and whether or not it should invoke `shoot()` or `bomb()` or some other method. Such an approach may look like this:

```
const baseSpaceShip = {
  [Symbol.for('Spaceship.weapon')]: {
    fire() {
      // the default shooting implementation
    }
  },
  fire: function() {
    if (this.hasAmmo()) {
      const weapon = this[Symbol.for('Spaceship.weapon')];
      weapon.fire();
      this.decrementAmmo();
    }
  },

  // other methods omitted
}
```

And the enhanced bomber spaceship could be implemented like so:

```
const bomberSpaceShip = Object.assign({}, baseSpaceShip, {
  [Symbol.for('Spaceship.weapon')]: {
    fire() {
      // drop a bomb
    }
  }
});
```

You could use a method like `getWeapon()` and this approach wouldn't be terrible, but it would add an additional public method to the spaceship that isn't meant to be used (just overridden). If there are a lot of hooks like this, it could make the public spaceship API bloated and harder to maintain.

If you use symbols to define these hooks, they're much

less accessible than the `getWeapon()` approach, yet not quite private so they can still be overridden. It's the perfect balance.

So far you've been hooking into the behavior of objects that you defined yourself, but what if you wanted to hook into the behavior of built-in objects?

Quick check 13.3

Q1:

What's wrong with the following code?

```
const laser = Symbol.for('Spaceship.weapon')
const laserSpaceShip = Object.assign({}, baseSpaceShip, {
  laser: {
    fire() {
      // shoot a laser
    }
  }
});
```

QC 13.3 answer

A1:

The symbol property must be a computed property, like so:

```
const laser = Symbol.for('Spaceship.weapon')
const laserSpaceShip = Object.assign({}, baseSpaceShip, {
  [laser]: {
    fire() {
      // shoot a laser
    }
  }
});
```

13.4. MODIFYING OBJECT BEHAVIOR WITH WELL-KNOWN SYMBOLS

Just as you used symbols to hook into the behavior of your spaceship in the previous section, built-in JavaScript objects also provide several hooks into their functionality with *well-known symbols*. A well-known symbol is a built-in symbol that's attached directly to `Symbol` such as `Symbol.toPrimitive`.

The `Symbol.toPrimitive` allows hooking into an object and controls how it gets coerced into a primitive value. A simple implementation could be

```
const age = {
  number: 32,
  string: 'thirty-two',
  [Symbol.toPrimitive]: function() {
    return this.string;
  }
};
console.log(`${myObject}`)      1
```

- 1 thirty-two

Since the value is a function, you could use the shorthand method syntax to tidy this up:

```
const age = {
  number: 32,
  string: 'thirty-two',
  [Symbol.toPrimitive]() {
    return this.string;
  }
};
console.log(`${myObject}`)      1
```

- 1 thirty-two

In this particular example, you're coercing the object into a string, but you may want to do something different when the object is coerced into a number. The

`Symbol.toPrimitive` function gets passed an argument hinting at what type of primitive it's being coerced into for such a case. The possible values are `string`, `number`, and `default`. We can update our example to make use of this, as in the next listing.

Listing 13.3. Working with well-known symbols

```
const age = {
  number: 32,
  string: 'thirty-two',
  [Symbol.toPrimitive](hint) {
    switch(hint) {
      case 'string':
        return this.string;
      break;
      case 'number':
        return this.number;
      break;
      default:
        return `${this.string}(${this.number})`;
      break;
    }
  }
};

console.log(`
  I am ${age} years old, but by the time you
  read this I will be at least ${+age + 1}
`);
console.log( age + ' ');
console.log( age.toString() );
```

- **1 I am 32 years old, but by the time you read this I will be at least 33.**
- **2 thirty-two(32)**
- **3 [object Object]**

In [listing 13.3](#) the `age` object will default to `thirty-two(32)` when coerced into a primitive, but will return `thirty-two` and `32` when coerced into a string or number, respectively. But calling `toString()` isn't a coercion and thus isn't affected.

There are several well-known symbols. Covering all of them is beyond the scope of this book. We'll cover

`Symbol.iterator` in the iteration unit, and you can get a list of all the well-known symbols here: <http://mng.bz/5838>.

Quick check 13.4

Q1:

Create an object with a `born` property which is a date. Use `Symbol.toPrimitive` to calculate the age based on the born date when coerced into a number.

QC 13.4 answer

A1:

```
const Person = {
  name: 'JD',
  born: new Date(1984, 1, 11),
  [Symbol.toPrimitive](hint) {
    const age = new Date().getFullYear() -
    this.born.getFullYear();
    switch(hint) {
      case 'number':
        return age;
      break;
      default:
        return `${this.name}, ${age}`;
      break;
    }
  }
};
```

13.5. SYMBOL GOTCHAS

Symbols are *primitives*, but they're the only primitives that don't have a literal form. So they must be created by calling the `Symbol()` function. Other primitives like numbers, strings, and Booleans (but not null or undefined) can be created the same way; `Number(1)`, `String('foo')`, or `Boolean(true)` for example. These same primitive functions can also be called with `new`, for example `new Number(1)`, but this doesn't return a primitive value; it returns a primitive *wrapper*, an object that wraps a primitive value. A primitive wrapper around a symbol is usually not desired, so to protect from accidentally creating one, if `Symbol()` is invoked with `new` like `new Symbol()`, an error will be thrown.

There are protections to keep you from accidentally coercing symbols to strings or numbers for the same reason. Doing so would throw an error as demonstrated in the next listing.

Listing 13.4. Every line here throws an error

```
new Symbol()  
`${Symbol()}`  
Symbol() + ''  
+Symbol()  
+Symbol.iterator  
Symbol.for('foo') + 'bar'
```

In this lesson you learned how to create unique constants that are safe from naming collisions. You then learned how to use symbols to create custom hooks into object behavior with global symbols. Finally you learned how to use well-known symbols for built-in-behavior hooks.

SUMMARY

In this lesson you learned the basics of symbols.

- Unique symbols are guaranteed to be unique because they all have unique identities in memory.
- Unique symbols are a great way to prevent naming collisions.
- Global symbols are automatically stored and retrieved from a global registry.
- Global symbols can be used to provide hooks into custom objects.
- Well-known symbols exist to provide hooks into built-in objects.
- The `new` operator shouldn't be used to create symbols.
- Symbols shouldn't be coerced into other primitives.

Let's see if you got this:

Q13.1

Use `Symbol.prototype.toPrimitive` to create an object that, when coerced into a string, serializes into a URI query string.

Lesson 14. Capstone: Simulating a lock and key

In this capstone exercise you're going to build a lock and key system. Each lock will have its own key that's unique to it. The lock will hold some secret data and only return the data if accessed with the correct key. You'll then build a game where you generate three locks and give the player only one key and one chance to unlock a prize!

14.1. CREATING THE LOCK AND KEY SYSTEM

You'll start by designing the lock API. You want to have a function called `lock` which accepts a single parameter that's the data you're locking up. It should then return a unique key (key as in *key to a lock*) and an `unlock` function that reveals the secret if invoked with the correct key. So the high-level API is basically `{ key, unlock } = lock(secret)`.

The `secret` can be any single datum. Both `lock` and `unlock` are functions, but you need to figure out what data type `key` is going to be.

A good solution would be a randomly generated string. But strings aren't exactly unique; it's possible they may overlap or even be guessable. You can use a complex string that's hard to guess or rarely overlaps like a SHA-1, but that would take a lot of effort or require installing a library. You already have a data type that's easy to generate and guaranteed to be unique, the `Symbol`. So you'll use that:

```
function lock(secret) {
  const key = Symbol('key')

  return {
    key, unlock(keyUsed) {
      if (keyUsed === key) {
        return secret
      } else {
        // do something else
      }
    }
  }
}
```

Here you have a `lock` function. When invoked it's given a `secret` and generates a new key which is a unique

symbol. Remember, every symbol created this way will always be unique and never overlap. The lock function then returns an object with the generated key and an unlock function. The unlock function accepts a keyUsed parameter and compares the key used to unlock the secret against the correct key. If they're the same, it returns the secret:

```
const { key, unlock } = lock('JavaScript is Awesome!')
unlock(key) 1
```

- **1 JavaScript is Awesome!**

You still need to figure out what to do if an incorrect key is used. In a real-world app you would probably throw an error if the wrong key is used. But for the sake of this exercise, just make it mask the value. You can use `String.prototype.repeat` which you learned in [lesson 6](#) to return a masked copy of the string. Something like

```
'*'.repeat( secret.length || secret.toString().length )
```

Here's the updated function.

Listing 14.1. Detecting incorrect keys

```
function lock(secret) {
  const key = Symbol('key')

  return {
    key, unlock(keyUsed) {
      if (keyUsed === key) {
        return secret
      } else {
        return '*'
          .repeat( secret.length ||
secret.toString().length )
      }
    }
  }
}

const { key, unlock } = lock(42)
```


<code>unlock()</code>	1
<code>unlock(Symbol('key'))</code>	1
<code>unlock('key')</code>	1
<code>unlock(key)</code>	2

- **1****

- **2 42**

Awesome! You've completed your lock and key system and it works great! Best of all, you did it with little code and no need for an external library. Now you can create several locks with several keys, and each lock will only be accessible by its associated key. What if you created several locks and keys and mixed them up?

14.2. CREATING A CHOOSE THE DOOR GAME

If you created three locks and presented them to a player as Door #1, Door #2, and Door #3, and gave the player a single key, you could ask the player to guess which door their key goes to. If they guess right, they win the prize behind the door. Hopefully that sounds like a fun game because that's exactly what you're going to build. Since the locks in this game are going to be doors, the secret can be *what's behind the door*. This may seem like a familiar game, but the twist is that the player may choose any door, but only if their key opens that door do they find out what's behind it and win the prize. Ready? Let's see how to build it.

Start by building the main user interface to your game, as shown in [listing 14.2](#). You need a way to present some options to a player and for that player to select an option. To keep it simple use prompt for this task.

Listing 14.2. Main interface for the game

```
function choose(message, options, secondAttempt) {
  const opts = options.map(function(option, index) {
1    return `Type ${index+1} for: ${option}`
  })

  const resp = Number(
    prompt(`${message}\n\n${opts.join('\n')}`) ) - 1 2

  if ( options[resp] ) {
    return resp
3
  } else if (!secondAttempt) {
    return choose(`Last try!\n${message}`, options, true)
4
  } else {
    throw Error('No selection')
5
  }
}
```

- **1 Build the multiple choice options that you display in the prompt. Add 1 to each index so the options start at 1 instead of 0.**
- **2 Then get back the number that the user typed into the prompt. You need to subtract that 1 that you added earlier.**
- **3 If you got an appropriate response from the prompt, return the value.**
- **4 If you didn't get a valid value, make a second attempt.**
- **5 If you still didn't get a valid value after a second attempt, throw an error.**

This choose function accepts a single message and an array of options. Then it handles wiring everything up so that the player is presented the message and the options, and told to type a number correlating to each option to make a selection. If an incorrect input is received, the player is asked one last time, and if a valid option is still not selected by the player, an error is thrown. Once a valid option is selected, the index of that option is returned from the choose function. By handling all of those details in this choose function, you can focus on what the message and options are elsewhere in your game. You can use the choose function as shown in the next listing and [figure 14.1](#).

Figure 14.1. Basic multiple choice question

example.com says:

Who is the greatest superhero of all time?

Type 1 for: Superman
Type 2 for: Batman
Type 3 for: Iron Man
Type 4 for: Captain America

☐ Prevent this page from creating additional dialogs.

Cancel OK

Listing 14.3. Building a multiple choice question

```
const message = 'Who is the greatest superhero of all time?'
const options = ['Superman', 'Batman', 'Iron Man', 'Captain America']

const hero = choose(message, options)
```

Now that you have a way to ask the user to choose a door, you need to generate three doors (locks) and assign the player a random key. First generate the doors with prizes. How do you think you should do that? You may try something like this:

```
const { key1, door1 } = lock('A new car')
const { key2, door2 } = lock('A trip to Hawaii')
const { key3, door3 } = lock('$100 Dollars')
```

This, however, won't work. Remember, you have to destructure objects using the correct property names that are on that object. The objects returned from `lock` only have the properties `key` and `unlock` and must be destructured as such. That may lead you to trying this:

```
const { key, unlock } = lock('A new car')
const { key, unlock } = lock('A trip to Hawaii')
const { key, unlock } = lock('$100 Dollars')
```

This, again, won't work. After the first destructure, the constants `key` and `unlock` are already taken and can't be declared again, which is attempted on lines 2 and 3. All is not lost: remember you learned when destructuring property names that there's a special syntax to assign them to different variable names. This is exactly what you need to do now:

```
const { key:key1, unlock:door1 } = lock('A new car')
const { key:key2, unlock:door2 } = lock('A trip to Hawaii')
const { key:key3, unlock:door3 } = lock('$100 Dollars')
```

Now you have three keys and three doors. So put them each in an array and grab a random key which you will give to the user:

```
const keys = [key1, key2, key3]
const doors = [door1, door2, door3]

const key = keys[Math.floor(Math.random() * 3)]
```

Easy enough. Now create the message and options that you'll present to the player:

```
const message = 'You have been given a \u{1F511} please choose a door.' 1

const options = doors.map(function(door, index) {
  return `Door #${index+1}: ${door()}`
})
```

- `1 \u{1F511}` is the Unicode key character.

Notice how you used `\u{1F511}` to escape your Unicode character. You could have just put the actual character into the message like so:

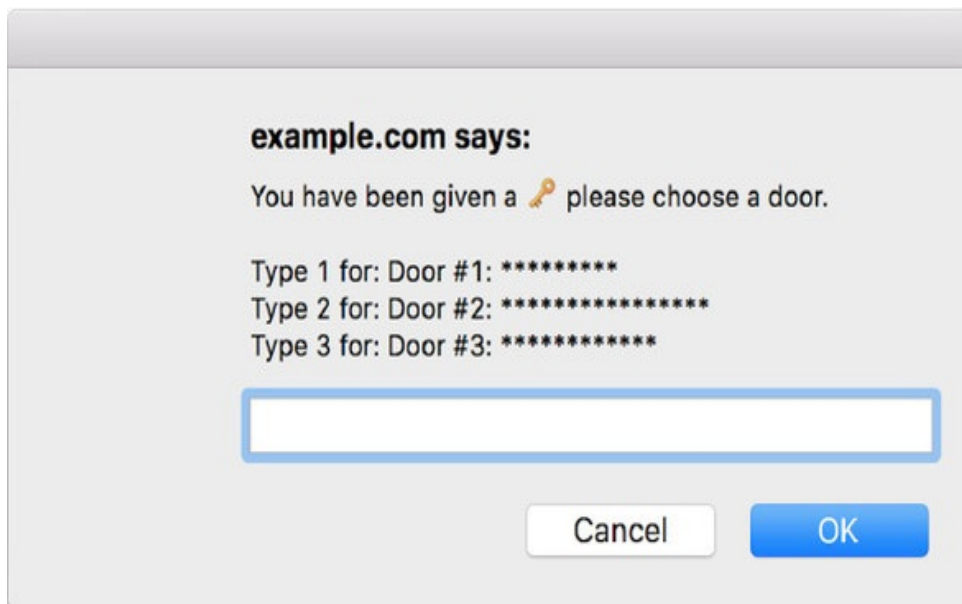
```
const message = 'You have been given a 🗝️ please choose a door.'
```

But that would be a lot harder to expect the reader of this book to type. Incidentally it brings up another new feature. Prior to ES2015 the syntax for Unicode escaping was `\uXXXX`, where `XXXX` was the Unicode hexadecimal code. But it only allowed up to four hex characters (16 bits), but notice how your character used five: `1F511`. To solve this prior to ES2015, you had to use what is called a *surrogate pair*, which uses two smaller Unicode values to generate a larger one. To get your key you could use `\uD83D\uDD11`. The concept of surrogate pairs or how to generate them is beyond the scope of this book, but ES2015 introduced a much easier way with `\u{XXXXXX}` which allows you to escape any size Unicode character.

Now that you have your message and options, you can pass them to your `choose` function, which will ask the user to choose a door. See [figure 14.2](#). Once they choose a door, they can attempt to unlock it with their key. If the key is correct they'll get the secret behind the door; otherwise they'll get the masked text. Either way you'll alert the response back to the user:

```
const door = doors[ choose(message, options) ]
alert( door(key) )
```

Figure 14.2. Choose the Door game in action



Put all this together in a function call `init`;, as shown in the next listing.

Listing 14.4. Combined Choose the Door function

```
function init() {  
  const { key:key1, unlock:door1 } = lock('A new car')  
  const { key:key2, unlock:door2 } = lock('A trip to Hawaii')  
  const { key:key3, unlock:door3 } = lock('$100 Dollars')  
  
  const keys = [key1, key2, key3]  
  const doors = [door1, door2, door3]  
  
  const key = keys[Math.floor(Math.random() * 3)]  
  
  const message = 'You have been given a \u{1F511} please  
choose a door.'  
  
  const options = doors.map(function(door, index) {  
    return `Door #${index+1}: ${door()}`  
  })  
  
  const door = doors[ choose(message, options) ]  
  
  alert( door(key) )  
}
```

Now if the user selects the correct door, they will see what's behind it, winning the prize. What other uses can you think of for this lock and key system? Maybe a multiplayer game with hidden treasure chests and their keys spread throughout the game. What if you took the

key from one lock and locked it up in another lock? That could get interesting.

SUMMARY

In this capstone exercise you simulated a lock and a key using symbols. When you created a lock, you got back a unique symbol as a key. Because symbols are always unique and don't risk collision, the lock cannot be opened without the correct key (symbol). You then used this lock and key to create a Choose the Door game.

Unit 3. Functions

Functions are a pretty fundamental construct for authoring applications. This is especially true in a language like JavaScript, which treats functions as first-class citizens. With ES2015 and later, many awesome features have been added to functions, including several completely new types of functions.

We'll start the unit by taking a look at default parameters and rest. I think most programmers, at some point, have had the need for default parameters and likely worked around it by checking for a value and assigning one if undefined at the start of the function. The rest param is even more useful. Anyone who has used the `arguments` object before will be happy to use *rest*. Not only does rest render the `arguments` object obsolete, it allows you to combine it with other arguments, rather than always acting like a catchall as the `arguments` object does.

We'll then take a dive into destructuring function arguments, which can also be combined with default parameters. When you put these all together, you can make some pretty powerful function declarations. You can even simulate *named* parameters. So, no more passing `null` values to a function like `myFunc(null, null, 5)`, because you only need to specify the third argument (or fourth, fifth, and so on).

After looking at all the new things you can do with existing functions, we'll look at two new types of functions: *arrow functions* and *generator functions*. Arrow functions are extremely useful in JavaScript, and you will find yourself using them daily once they're in

your repertoire. Generator functions, on the other hand, may not be something you need to reach for that often, but they're a powerful new tool when needed.

You'll wrap up the unit by creating a simulation runner that pits prisoner simulations against each other in a Prisoner's Dilemma scenario.

Lesson 15. Default parameters and rest

After reading lesson 15, you will

- Know how to use default parameters
- Know how to gather parameters with the rest operator
- Know how to use rest to pass arguments between functions

Sometimes new language features provide ways to achieve things that were impossible or nearly impossible to do before. Other times they simply add a nicer way of achieving something that was already easily implemented. But just because something is easy to implement or requires few lines of code to do, doesn't necessarily make it *readable*. That's exactly what *default function parameters* and *rest parameters* do: they provide a more concise and much more readable way of achieving something.

Consider this

By looking at this implementation of the `pluck` function, can you quickly tell what it's doing? That first line—what is it doing? That line requires too much thought. In its current state it probably needs a comment explaining what it does.

```
function pluck(object) {  
  const props = Array.from(arguments).slice(1);  
  return props.map(function(property) {  
    return object[property];  
  });  
}  
const [ name, desc, price ] =  
  ➡ pluck(product, 'name', 'description', 'price');
```

15.1. DEFAULT PARAMETERS

Let's imagine you're building a website that shows the current president's approval ratings over the course of their time in office. You want to build a chart function that will show the information in a line chart. You decide that 800 by 400 is a good size for the chart, but you want to allow for setting custom sizes. For example, the 800 by 400 chart might be at the top with the national approval ratings, and then be broken down by state using charts that are only 400 by 200. You may end up writing a function like so:

```
function approvalsChart(ratings, width, height) {  
  if (!width) {  
    width = 800  
  }  
  if (!height) {  
    height = width / 2  
  }  
  // build the chart with ratings  
}  
  
const nationalChart = approvalsChart(nationalRatings)  
1  
const georgiaChart = approvalsChart(georgiaRatings, 400)  
2
```

- **1 Chart is 800x400**
- **2 Chart is 400x200**

You can now achieve the same thing with default function parameters like so:

```
function approvalsChart(ratings, width = 800, height = width /  
2) {  
  // build the chart with ratings  
}  
  
const nationalChart = approvalsChart(nationalRatings)  
1  
const georgiaChart = approvalsChart(georgiaRatings, 400)  
2
```

- **1 Chart is 800x400**
- **2 Chart is 400x200**

Notice how you used the expression `height = width / 2` right in your arguments list. You can actually use any expression in the assignment of a default parameter, and that expression will be evaluated at the time the function is evaluated. You can use any variables in scope, including preceding parameters. The context of the expression will always be the same context the function is invoked with, meaning any use of `this` will evaluate the same as it would inside the function body.

Listing 15.1. Accessing `this` in default parameters

```
const ajax = {
  host: 'localhost',
  load(url = this.host + '/data.json') {
    // load data from url.
  }
}

ajax.load();                                1
ajax.host = 'www.example.com';
ajax.load();                                2
ajax.load('localhost/moredata.json');       3
```

- **1 url is localhost/data.json**
- **2 url is www.example.com/data.json**
- **3 url is localhost/moredata.json**

When you first invoked `ajax.load()` without a parameter, the default checked `this.host` which was `localhost`, and the `url` parameter became `localhost/data.json`. You then changed the `host` property, so when you invoked the same function again, the default again checked `this.host` which now was `www.example.com`, so the `url` parameter became `www.example.com/data.json`. Finally, when you called `ajax.load('localhost/moredata.json')`,

the default wasn't used and the `url` parameter was set to `localhost/moredata.json`.

The parameter's default value is tied to its index in the parameter list. You may be tempted to use a default value to make an optional param follow a required param, as in our range example. If two values are passed, then you want them to be `min` and `max`, respectively, but if only one value is passed, you want it to be the `max` with a `min` defaulted to 0. As nice as this would be, this is not how default params work, as shown in the next listing.

Listing 15.2. Mandatory parameters must precede optional ones

```
function range(min = 0, max) {  
  // ... create a range from min to max  
}  
range(5, 10);           1  
range(10);              2
```

- **1 min would be 5 and max would be 10.**
- **2 min would be 10 and max would be undefined.**

Because default params are tied to their index, you can't use them to make a parameter optional unless it's the last param. The params passed to the function are assigned by their index, so no matter what, the first value is going to be assigned to the `min`, and the second value will be assigned to the `max`. The only way you could set the `max` while using the default `min` would be if you invoked range like `range(undefined, 10)` because that maintains the correct indices for each param. The first param is `undefined` and so becomes the default value of 0; the second param is 10.

With the power of expressions in default parameters, you could do something clever to check the length of the arguments and set the `min` param to the correct value

conditionally, as shown in the next listing.

Listing 15.3. Fudging default parameters (not recommended!)

```
function range(temp = 0, max = temp, min = arguments.length > 1
? temp : 0)
{
    // ... create a range from min to max
}
```

In this `range` function, you first defined a `temp` param with a default value of `0`, then a `max` that defaults to the `temp` value. Finally you defined a `min` param that checks the length of `arguments` and conditionally defaults to `0` or `temp`. This works because of the following conditions:

- Invoked with `10`:
 - `temp` is set to `10`.
 - `max` defaults to `temp` so `max` is `10`.
 - `arguments.length` is not greater than `1` so `min` gets set to `0`.
 - Effect: `min` is `0`, `max` is `10`.
- Invoked with `5, 10`:
 - `temp` is set to `5`.
 - `max` is set to `10`.
 - `arguments.length` is greater than `1` so `min` gets set to `temp (5)`.
 - Effect: `min` is `5`, `max` is `10`.
- Invoked with `5, 5`:
 - `temp` is set to `5`.
 - `max` is set to `5`.
 - `arguments.length` is greater than `1` so `min` gets set to `temp (5)`.
 - Effect: `min` is `5`, `max` is `5`.

Even though this works, I would advice against code like this because it obscures what's happening, wastes a

param, and could lead to bugs if users of the function started invoking it with all three params. This was merely an exercise to demonstrate the possibilities.

Notice how in [listing 15.3](#) you're using the `arguments` object in the parameter list to calculate a default value. Remember, the expression used for a default param gets invoked in the context of the function body! Does this mean you can use other variables declared in the function body? Effectively no: remember in [lesson 4](#) we talked about temporal dead zones. Other variables in the function body are technically in scope, but they're in a temporal dead zone and can't be accessed yet.

Quick check 15.1

Q1:

In the following function invocation, what will be the value of the `args` parameter?

```
function processArgs(args = Array.from(args)) {  
  // ...  
}  
processArgs(1, 2, 3);
```

QC 15.1 answer

A1:

It will be 1. Remember you called it with a value. So even though the default value creates an array from the `arguments` object, it doesn't matter: the default doesn't get calculated because a value was passed. If you want to create an array of the arguments, use the rest operator.

Usually default parameters allow you to use a sane default in the case that a value wasn't given. This makes functions more useful, because falling back to a default is usually better than throwing an error (depending on the scenario). That isn't the only thing you can do with default parameters though. In the next section we will explore using default parameters to prevent recalculating data.

15.2. USING DEFAULT PARAMS TO SKIP RECALCULATING VALUES

Remember that in the beginning of this lesson, you created an `approvalsChart` function to calculate approval ratings. You're going to take it further and create a library for calculating the ratings and generating the charts. A first attempt may be something like this.

Listing 15.4. Calling `getRatings` twice

```
const chartManager = {
  getRatings() {
    // some intensive task to calculate all the ratings
  },

  nationalRatings() {
    const ratings = this.getRatings()
    return approvalsChart(ratings)
  },

  stateRatings(state) {
    const ratings = this.getRatings()
    stateRatings = ratings.filter(function(rating) {
      rating.state === state
    })
    return approvalsChart(ratings, 400)
  },

  stateAndNationalRatings(state) {
    const nationalChart = this.nationalRatings()      1
    const stateChart = this.stateRatings(state)       2
    return {
      nationalChart,
      stateChart
    }
  }
}

const charts = stateAndNationalRatings('georgia')
```

- **1 Calculates the ratings**
- **2 Calculates the ratings (again)**

Notice how it ends up calling the `getRatings` method twice? If this method were resource-intensive it would make sense to avoid wasting the second computation.

You can solve this with default params, as shown in the following listing.

Listing 15.5. Calling `getRatings` only once

```
const chartManager = {
  getRatings() {
    // some intensive task to calculate all the ratings
  },

  nationalRatings(ratings = this.getRatings()) {
    return approvalsChart(ratings)
  },

  stateRatings(state, ratings = this.getRatings()) {
    stateRatings = ratings.filter(function(rating) {
      rating.state === state
    })
    return approvalsChart(ratings, 400)
  },

  stateAndNationalRatings(state) {
    const ratings = this.getRatings()
    const nationalChart = this.nationalRatings(ratings)
    const stateChart = this.stateRatings(state, ratings)
    return {
      nationalChart,
      stateChart
    }
  }
}

const charts = stateAndNationalRatings('georgia')
```

Now the methods `nationalRatings` and `stateRatings` call `getRatings` as a default parameter. This means that each function can be invoked by itself and it will calculate the ratings. But if you want to get both national and state, you can pre-calculate the ratings and pass the data to each method, preventing the extra calculation step.

Default params are a great way of handling expected parameters by ensuring they have values. They don't offer much help with unexpected (or indeterminate) parameters, but for that you can use the new *rest* operator.

Quick check 15.2

Q1:

In the following snippet what will be returned from `getC()`?

```
function getC(a = 'b', b = c, c = a) {  
  return c;  
}  
getC();
```

QC 15.2 answer

A1:

Nothing, it will be a `SyntaxError` because `b` defaults to `c` but `c` is declared *after* `b`.

15.3. GATHERING PARAMETERS WITH THE REST OPERATOR

Back in [lesson 9](#), you learned how to get an array from the `arguments` object using `Array.from`. As useful as this is, there's an even more convenient way to get the arguments as an array using `rest`:

```
function avg() {                                1
  const args = Array.from(arguments);
  //...
}

function avg(...args) {                          2
  //...
}
```

- **1 Gather arguments into array using `Array.from`**
- **2 Gather arguments into array using `rest`**

The way `rest` works is by declaring a function param that starts with three dots. This is called a `rest param`, although the name of the param itself can be any valid variable name. This will group all the remaining params starting from the position of the `rest param` into an array. You can also only have one `rest parameter` and it must be the last argument in the list:

```
function countKids(...allMyChildren) {
  return `You have ${allMyChildren.length} children!`;
}

countKids('Talan', 'Jonathan');                1

function family(spouse, ...kids) {
  return `You are married to ${spouse} with ${kids.length} kids`;
}

family('Christina', 'Talan', 'Jonathan');      2
```

- **1 You have two children!**

- **2 You are married to Christina with two kids.**

The name comes from the idea that you first specify which arguments you want to put into individual variables, then specify the argument you want the rest of the params grouped into (see the following listing). Some languages, like Ruby, refer to this concept as a *splat*, but that's most likely because those languages use an asterisk instead of three dots as the operator, and an asterisk resembles the shape of a splat.

Listing 15.6. Error occurs if you put params after rest in JavaScript

```
function restInMiddle(a, ...b, c) {           1
  return [a, b, c];
}
```

- **1 SyntaxError: rest parameter must be last**

The rest param must come last in the arguments list or a syntax error is thrown. You may opine that this is the only logical way for rest params to work, but other languages, like CoffeeScript, for example, do allow for params to be specified after the rest param, as the following listing shows.

Listing 15.7. Params after rest in CoffeeScript

```
restInMiddle = (a, b..., c) -> [a, b, c];
restInMiddle(1,2,3,4,5);                      1
```

- **1 [1, [2, 3, 4], 5];**

Now that you know how rest works, lets take a look at the priming function reimplemented using rest:

```
function pluck(object, ...props) {
  return props.map(function(property) {
    return object[property];
  });
}
```

At a glance, isn't it much more readable? It no longer

requires a comment to explain an obscure line of code.

Quick check 15.3

Q1:

Implement the following `cssClass` function so that it generates a CSS class list by taking the first argument and adding it to the *rest* of the arguments.

```
cssClass('button', 'danger', 'medium');    1
```

- 1 Should return “button button-danger button-medium”

QC 13.3 answer

A1:

```
function cssClass (primary, ...others) {  
  const names = others.reduce(function(list, other) {  
    return list.concat(`${primary}-${other}`);  
  }, [primary]);  
  return names.join(' ');  
}
```

15.4. USING REST TO PASS ARGUMENTS BETWEEN FUNCTIONS

Imagine you're using a library that does some image processing, but you need to hook into the processing function to add some logging. Many libraries provide middleware^[1] for injecting code like this, but not all do; in such cases *monkey patching* will have to do. Monkey patching is the process of redefining a function to inject some custom logic before invoking the original function. In [listing 15.8](#) you monkey-patch the process function of a fictitious `imageDoctor` library and use rest to gather all the arguments and pass them along to the original function. This ensures that the original function is always invoked with the same arguments as the wrapping function.

1

Middleware are custom hooks provided by library authors that allow for injecting custom behaviors. Some JavaScript libraries that provide middleware are `express.js` (<http://expressjs.com/>) and `redux.js` (<http://redux.js.org/>). Also see <https://en.wikipedia.org/wiki/Middleware>.

Listing 15.8. Using rest to forward parameters when monkey-patching

```
{
  const originalProcess = imageDoctor.process;           1
  imageDoctor.process = function(...args) {               2
    console.log('imageDoctor processing', args);          3
    return originalProcess.apply(imageDoctor, args);      4
  }
}
```

- **1 First get a reference to the original method.**
- **2 Define a new function that gathers all the arguments.**
- **3 Inject your logging.**
- **4 Return the result of the original function invoked with args.**

When we get into classes later in this book, you'll learn

about subclasses extending superclasses. When a subclass extends a superclass, it can override methods on the superclass. The overridden method can be invoked within the method that overrides it, and rest can again be used to handle the passing of arguments between the two.

Quick check 15.4

Q1:

Assume you're using a function called `ajax` to load data. Use `rest` to monkey-patch the function to log all its arguments.

QC 15.4 answer

A1:

```
{
  const originalAjax = ajax;
  ajax = function(...args) {
    console.log('ajax invoked with', args);
    return originalAjax.apply(null, args);
  }
}
```

SUMMARY

In this lesson you learned how to use default parameters and rest.

- Default parameters allow for sane defaults.
- Default parameters can be calculated using an expression.
- Default parameter expressions are executed in the context of the function body.
- Default parameters are tied to their index in the arguments list.
- Rest parameters gather all the remaining arguments as an array.
- Rest parameters can be used to gather all arguments as an array.
- Rest parameters must be the last argument.
- There can only be one rest parameter per arguments list.

Let's see if you got this:

Q15.1

Create a function called `car` that allows you to create a car object. The function should accept the number of seats available as a parameter with a default value. The car object should have a `board` method that accepts a driver and any number of passengers using rest. The `board` method should then log who the driver is, who the passengers that fit are (according to the number of seats), and also list any passengers that could not fit.

Lesson 16. Destructuring parameters

After reading lesson 16, you will

- Know how to destructure array parameters
- Know how to destructure object parameters
- Know how to simulate named parameters
- Know how to create aliased parameters

In lesson 11 you learned about destructuring objects and arrays. These same principles can be used directly within the parameter list of a function. This makes array and object parameters easier to deal with and more self-documenting, and opens the door for useful techniques such as simulating named parameters.

Consider this

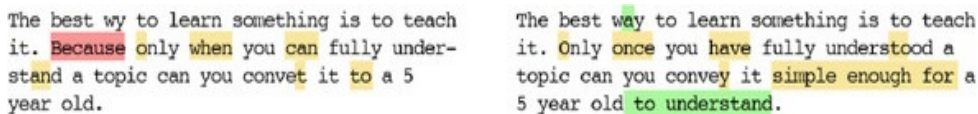
The following function takes three parameters. How would you go about making all of the parameters optional so the function could be called setting only the necessary values? For example, what if you only wanted to set the height and use a default width? How would the caller of the function specify that it just wants to specify the height? By name? What if it got the name wrong? For example, the caller might use `h` when the full word `height` was required.

```
function image(src, width, height) {  
  // build image  
  
}
```

16.1. DESTRUCTURING ARRAY PARAMETERS

Imagine you're writing a program to compare the differences between two files. You're using a library that provides a `diff` function that takes two strings and computes the difference between them. It returns an array with three values. The first value is the text that has been added. The second is the text that has been removed, and finally the third value is the text that has been modified.

Figure 16.1. Figuring out which text has been added (green), changed (yellow), and deleted (red)



The best way to learn something is to teach it. Because only when you can fully understand a topic can you convey it to a 5 year old.

The best way to learn something is to teach it. Only once you have fully understood a topic can you convey it simple enough for a 5 year old to understand.

You need to write a function that can take these diffs and render a visualization, displaying them like in [figure 16.1](#). Let's call this function `visualize`. If it were to accept three individual parameters—inserted, deleted, and modified—you could connect these two functions together like so:

```
function visualize(inserted, deleted, modified) {  
  // ... render visualization to screen  
}  
  
const [ inserted, deleted, modified ] = diff(fileA, fileB);  
visualize(inserted, deleted, modified);
```

This works but it's tedious having to extract the values out of the `diff` function just so you can pass them to the `visualize` function. A cleaner solution would be to make the `visualize` function's input match the output of the `diff` function. Then connecting them would be much simpler:

```
visualize( diff(fileA, fileB) );
```

Now the connection between the two functions is much cleaner. I would even make the argument that the code is more self-documenting, as it now reads *visualize diff*, which is exactly what you're doing: visualizing the diff! But in order to make this possible, the `visualize` function now only takes one parameter, an array with the three values. So you have to reimplement your `visualize` function to take account of this:

```
function visualize(diff) {
  const inserted = diff[0];
  const deleted = diff[1];
  const modified = diff[2];
  // ... render visualization to screen
}
```

At this point, you've moved the tedious extraction step from outside the function to inside it. Yes, it's now much cleaner looking when you connect the `visualize` and `diff` functions together, but it comes at a cost of muddying up the implementation of the `visualize` function.

Hopefully you're already thinking that you can clean this up using array destructuring like so:

```
function visualize(diff) {
  const [ inserted, deleted, modified ] = diff;      1
  // ... render visualization to screen
}
```

- **1 Destructure the `diff` parameter into the three values you need.**

If so, you're correct! However, you can remove a step and make this even cleaner by doing the destructuring in the parameter list:

```
function visualize([ inserted, deleted, modified ]) {
```

```
// ... render visualization to screen
}
```

Now you've successfully removed some of the cognitive load between connecting these two functions. You made the code more self-documenting and you did it without having to sacrifice readability elsewhere.

What if the `diff` function returned an object with `inserted`, `modified`, and `deleted` properties instead of an array? In the next section you'll update your `visualize` function to handle such a case.

Quick check 16.1

Q1:

Let say you are writing a widget library. You want to allow whoever installs your widget to be able to set the colors the widget uses. Implement the following `setColors` function so that it destructures the array of colors. Use any names you like for each of the three colors.

```
setColors([ '#4989F3', '#82D2E1', '#282C34' ]);
```

QC 16.1 answer

A1:

```
function setColors([primary, secondary, attention]) {
  // ...
}
```

16.2. DESTRUCTURING OBJECT PARAMETERS

In the previous section you worked with a `diff` function that returned an array containing the inserted, deleted, and modified lines between two files. But what if you updated the version of the `diff` library you were using and the function returned an object with inserted, deleted, and modified properties? How could you update your `visualize` function to correctly extract the data from the new format? The solution is almost exactly the same:

```
function visualize({ inserted, deleted, modified }) {  
  // ... render visualization to screen  
}  
  
visualize( diff(fileA, fileB) );
```

As you can see, the only thing you had to change was that instead of wrapping your parameter list with `[` and `]` brackets, you swapped them out for `{` and `}` braces. Instead of grabbing values from an array by using their position in the array, you're now grabbing particular properties from an object using the actual names of those properties.

In the *array* destructuring, you can use whatever names you want as long as the position of the fields is correct. In the *object* destructuring, you can list the properties in any order, as long as the names match up correctly. This is just like regular object and array destructuring, only taking place in the parameter list of a function.



Quick check 16.2

Q1:

Rewrite the `setColors` function to destructure colors using object destructuring. Remember: this time the names, not the order, must align.

```
setColors({ primary: '#4989F3', danger: '#DB231C',  
success: '#61C084' });
```

QC 16.2 answer

A1:

```
function setColors({ primary, success, danger }) {  
  // ...  
}
```

16.3. SIMULATING NAMED PARAMETERS

Let's imagine you're writing a function to build a car. You decide that you want to let the caller of the function set the make, model, and year of the car, so you add a parameter for each:

```
function car(make, model, year) {  
  // ... make the car  
}
```

However, you want each parameter to be optional and have a default value. You may think that since JavaScript now supports default values, it will solve your problem. But what if the caller only wants to set the year (the third parameter)? With default parameters, the first two must either be set or explicitly passed as `undefined`:

```
function car(make = 'Ford', model = 'Mustang', year = 2017) {  
  // ... make the car  
}  
  
let classic = car(undefined, undefined, 1965);
```

This is not ideal. You can use object destructuring in the parameter list to simulate *named parameters* (the ability to set parameters by name instead of position):

```
function car({ make, model, year }) {  
  // ... make the car  
}  
  
let classic = car({ year: 1965 });
```

Now it looks and behaves like you're using named parameters, but in reality you're just using a single parameter of an object, as shown in [figure 16.2](#).

Figure 16.2. Simulating named parameters

**You pass a single
parameter, an object...**

```
let classic = car({ year: 1965, make: 'Ford' });
```

**...but it behaves like you're passing
two params by name: year and make.**

Single object

```
function car({ make, model, year }) {  
}
```

Values by name

You still need to figure out a way to set defaults for each. Since this is only one parameter you could default that parameter (the entire object) to an object with all the values preset:

```
function car({ make, model, year } =  
  {make:'Ford',model:'mustang',year:2017}  
  ) {  
  // ... make the car  
}
```

But this will only work if no object is passed at all. By passing an object like `{ year: 1965 }`, even though it's missing some desired keys, the parameter is still passed. Remember, in reality you're only dealing with a single parameter, so the default does not get set:

```
function car({ make, model, year } =  
  {make:'Ford',model:'mustang',year:2017}  
  ) {  
  // ... make the car  
}
```

```
let modern = car();           1  
let classic = car({ year: 1965 }); 2
```

- **1 No parameter passed, so defaults to object with all values set**
- **2 A parameter is passed, so the default isn't used and you don't get the make or model.**

A better way is possible. You can actually destructure the object and give each individual key a default value independently:

```
function car({ make = 'Ford', model = 'Mustang', year = 2017 })  
{  
  // ... make the car  
}  
  
let classic = car({ year: 1965 }); 1
```

- **1 The year gets set while the make and model fall back to defaults.**

Now that you're setting default values for each individual key that you're destructuring, you can successfully set what values you want to change and have the others fall back to defaults. The only problem remaining is when you invoke the function without a parameter at all:

```
function car({ make = 'Ford', model = 'Mustang', year = 2017 })  
{  
  // ... make the car  
}  
  
let modern = car(); 1
```

- **1 An error is thrown here because you attempt to destructure a missing object.**

As it stands, you still need to call the function with an empty object like `car ({})`, even when you don't want to change any of the default values. This is because if you don't pass in an object, the parameter will be

undefined and you won't be able to perform object destructuring on an undefined value. To fix this, you can continue to default each key to a specific value but also default the entire parameter to an empty object:

```
function car({ make = 'Ford', model = 'Mustang', year = 2017 }  
= {}) {  
  // ... make the car  
}  
  
let modern = car();  
let classic = car({ year: 1965 });
```

- **1 All values fall back to defaults.**
- **2 The year gets set while the make and model fall back to defaults.**

Now if you call the function without a parameter at all, it defaults to an empty object, and because the empty object is missing the make, model, and year, they all still fall back to their default values. And of course, if you do pass in an object, any of the missing keys get set to their respective default values.

Quick check 16.3

Q1:

Create a pagination function that simulates named parameters for `currentPage` and `resultsPerPage` and that default to 1 and 24 respectively.

QC 16.3 answer

A1:

```
function pagination({ currentPage = 1, resultsPerPage = 24
} = {}) {
  // ...
}
```



16.4. CREATING ALIASED PARAMETERS

Let's imagine you have a function called `setCoordinates` that takes an object with latitude and longitude properties. The problem is that you're using two different mapping libraries: one for rendering maps, another for doing reverse geo lookups. One uses the properties `lat` and `lon`, while the other uses `lat` and `lng`. Because of this, you want to make your function smart enough to handle both. You may be inclined to define your function like so:

```
function setCoordinates(coords) {  
  let lat = coords.lat;  
  let lng = coords.lng || coords.lon;           1  
  // ... use lat and lng  
}
```

- **1 Assign either `coords.lng` or `coords.lon` (whichever exists) to your variable.**

You can use a combination of parameter destructuring and default values to do this as well:

```
function setCoordinates({ lat, lon, lng = lon }) {  
  // ... use lat and lng  
}
```

Here you're using destructuring to grab `lat`, `lon`, and `lng` directly. But you're defaulting `lng` to `lon`; this means that no matter which one is given to you, you'll be able to capture it as `lng`.

If `lng` defaults to `lon` to create an alias, what if you actually want to set a real default value to `lng`? Well, since `lng` defaults to `lon`, you can simply add the default value to `lon`:

```
function setCoordinates({ lat = 33.7490, lon = -84.3880, lng =
lon }) {
  // ... use lat and lng
}
```

Now your function accepts `lon` or `lng` just as it did before, but this time it also defaults to a specific location (Atlanta). The way this works is that if `lng` is set, it doesn't matter what `lon` is set to; you ignore it. If `lng` is not set, it defaults to `lon`. If `lon` is not set, it defaults to `-84.3880`, which transitively also makes `lng` default to `-84.3880`.

This useful technique can also be used without destructuring by using default values:

```
function setCoordinates(lat = 33.7490, lon = -84.3880, lng =
lon) {
  // ... use lat and lng
}
```

Notice how you didn't use the `{}`: which means this function doesn't take a single object as a parameter and then destructure values from it like the previous one. This time, you're actually accepting three different arguments and setting defaults for them in a similar way.

Quick check 16.4

Q1:

Use this same technique to make a function that can take either `width` and `height` properties or `w` and `h` properties with a default size.

QC 16.4 answer

A1:

```
function setSize({ width = 50, height = width, w = width,  
  h = height }) {  
  // use w and h  
}
```



SUMMARY

In this lesson you learned how to apply destructuring techniques to function parameters, including how to

- More elegantly connect the outputs of one function to the inputs of another.
- Make all arguments optional by simulating named parameters.
- Make the names of arguments have aliases by combining destructuring with default values.

Let's see if you got this:

Q16.1

In the following code, there are three functions. Each interacts with an internal map object, updating a different property. Combine these three functions into a single function called `updateMap` that simulates named parameters for `zoom`, `bounds`, and `coords`. Additionally, create an alias so that `coords` can also be set by the named `center`.

```
function setZoom(zoomLevel) {
  _privateMapObject.setZoom(zoomLevel);
}

function setCoordinates(coordinates) {
  _privateMapObject.setCenter(coordinates);
}

function setBounds(northEast, southWest) {
  _privateMapObject.setBounds([northEast, southWest]);
}
```

Lesson 17. Arrow functions

After reading [lesson 17](#), you will

- Know how to make your code succinct with arrow functions
- Know how to maintain context with arrow functions

Arrow functions in JavaScript are directly inspired by the *fat arrow functions* of CoffeeScript. They behave similarly to CoffeeScript's by providing a much more succinct way of writing a function expression and also maintaining their context (what `this` refers to). The syntax isn't always the same as CoffeeScript's but they are just as useful and a great addition that makes things like anonymous functions and inline callbacks much more elegant.

Sometimes extraneous syntax can make code much harder for the human mind to parse, because there are several extra pieces to think about. It's not a hard and fast rule that few characters mean easy to understand. For example, single-letter variable names or overly clever `_code_golf_`^[1] solutions are terrible to read. However, if you can convey a meaning eloquently with fewer characters, it's almost always more understandable than it would be with more characters. The human brain has a hard time parsing lots of information at once, so the more noise you can reduce, the better.

¹

A game in which a programmer writes a program with as few characters as possible.

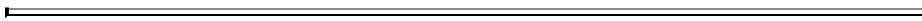


Consider this

Here's some code that maps a set of numbers, applying an exponent to each of them. Notice how you're using

`that=this` to maintain context inside the anonymous function passed to `map`. The `map` function in particular accepts a second parameter that sets the context, but many functions don't provide such a convenience, leaving developers to come up with work-arounds like the `that=this` used here. How many times have you had to write code like this? Wouldn't it be nice if there were an elegant way to maintain the outer context?

```
const exponential = {
  exponent: 5,
  calculate(...numbers) {
    const that = this;
    return numbers.map(function(n) {
      return Math.pow(n, that.exponent);
    });
  }
}
exponential.calculate(2, 5, 10); // ???
```



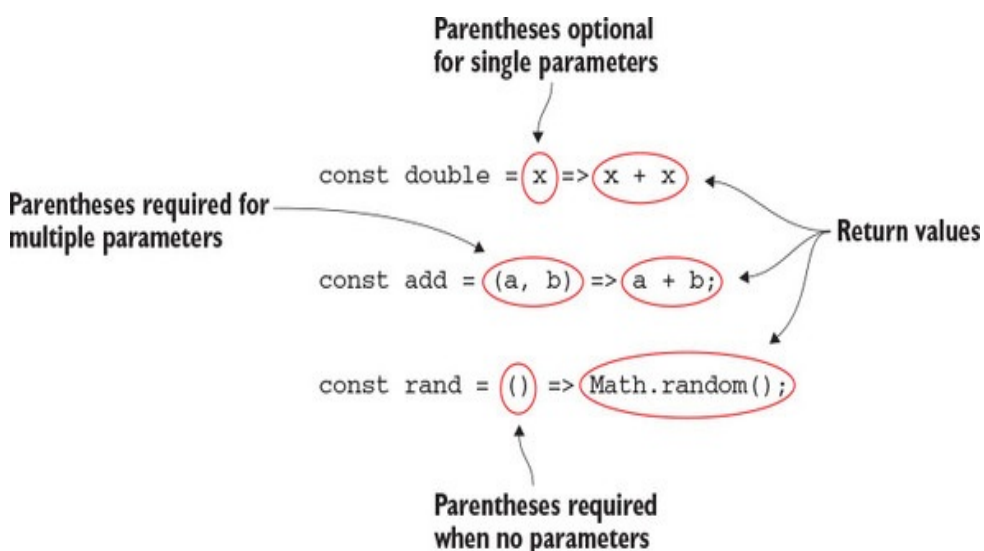
17.1. SUCCINCT CODE WITH ARROW FUNCTIONS

The arrow function has a couple of different syntaxes depending on the number of arguments or the number of expressions in the function body. The most elegant is when there's exactly one parameter and exactly one expression in the function body. The double function we just saw is such an example. It's functionally equivalent to the function expression defined just before it. It has the syntax `singleParam => returnExpression`. If there isn't exactly one param (either zero, or two or more), then the parameters must be wrapped in parentheses:

```
const add = (a, b) => a + b;           1
const rand = () => Math.random();      2
```

- **1 Multiple parameters need to be wrapped in parentheses.**
- **2 You must use empty parentheses when there are no parameters.**

Figure 17.1. Arrow function parentheses rules



You can still wrap a single parameter in parentheses as well if desired (see [figure 17.1](#)). There are a couple of

other situations when you must wrap your single parameter with parentheses if it's a rest parameter or a destructured parameter:

```
const rest = (...args) => console.log(args);      1
const rest = ...args => console.log(args);        2
const destruct = ([ first ]) => first;             3
const destruct = [ first ] => first;              4
```

- **1 Correct syntax**
- **2 Syntax error**
- **3 Correct syntax**
- **4 Syntax error**

If there's more than one expression in the function body, it must be wrapped in curly braces:

```
const doTasks = (a, b) => {
  taskOne();          1
  taskTwo();          2
}
```

- **1 First expression**
- **2 Second**

This is similar to `if` statements and `for` loops, where the curly braces are optional only if the body (the code that would be inside the curly braces) is a single expression or statement. If there's more than one expression or statement, then curly braces must be used:

```
if (true) {
  doFirstThing();
  doSecondThing();      1
}

if (true)
  doFirstThing();
  doSecondThing();      2

const doTasks = (a, b) => {
  taskOne();
  taskTwo();            3
}
```

```
}  
  
const doTasks = (a, b) =>  
  taskOne();  
  taskTwo();
```

4

- **1 This is part of the if statement.**
- **2 This is not part of the if statement.**
- **3 This is part of the function.**
- **4 This is not part of the function.**

When the curly braces are omitted, there's an additional benefit besides not having to type the extra two characters: there's an implicit return of the single expression. This implicit return of single expression arrow functions is elegant because it reads as the input value pointing to the return value. For example, a no-op function^[2] would look like `x => x`, or a function that wraps its value in an array would look like `x => [x]`. See how it reads as `from => to` or `start => finish` or `give => get`. This concise syntax has almost zero cognitive load, and is extremely easy to mentally parse while also being self-documenting.

2

No operation, often used to allow hooking into functionality. See <https://en.wikipedia.org/wiki/NOP>.

Another scenario where arrow functions make code extremely concise is when you need to make a function that returns another function. Let's say you want to make a function called `exponent` that takes a number and returns another function that will apply the exponent to a given base number:

```
const exponent = exp => base => base ** exp  
  
const square   = exponent(2)  
const cube     = exponent(3)  
const powerOf4 = exponent(4)
```

square(5)	1
cube(5)	2
powerOf4(5)	3

- **1 25**
- **2 125**
- **3 625**

Also notice the `**` operator. This is a new operator introduced in ES2016 that applies an exponent. Previously you would have to use `Math.pow` to achieve the same thing.

Let's take a look at the ES5 equivalent implementation of this exponent function:

```
var exponent = function(exp) {  
  return function(base) {  
    return Math.pow(base, exp)  
  }  
}
```

I definitely think the arrow function version is much simpler and easier to read.

In [lesson 15](#) you defined a `pluck` function that grabs a specified set of values from an object. Here's the original `pluck` function followed by a version using arrow functions:

```
function pluck(object, ...props) {  
  return props.map(function(property) {  
    return object[property];  
  });  
}  
  
function pluck(object, ...props) {  
  return props.map( prop => object[prop] );  
}
```

Doesn't the latter version with the arrow function read more nicely? Higher-order functions like `map`, `reduce`,

`filter`, and so on lend themselves to arrow functions beautifully. This is true even more so when those higher-order functions need to be executed in their containing context.

Quick check 17.1

Q1:

Convert the following summation function to use arrow functions:

```
const sum = function(...args) {  
  return args.reduce(function(a, b) {  
    return a + b;  
  });  
}
```

QC 17.1 answer

A1:

```
const sum = (...args) => args.reduce( (a, b) => a + b );
```

17.2. MAINTAINING CONTEXT WITH ARROW FUNCTIONS

Let's take the `pluck` function from the previous section a little further. Instead of it getting the object to operate on as an argument, let's assume a `Model` object that is a wrapper object for a record from a database. The `pluck` function will be a method on this model object. If you implemented it without any considerations for context, it would break when trying to refer to itself using the keyword `this` inside of a callback function:

```
const Model = {  
  // ... other methods  
  
  get(propName) {  
    // return the value for `propName`  
  }  
  
  pluck(...props) {  
    return props.map(function(prop) {  
      return this.get(prop);  
    });  
  }  
}
```

- **1 The keyword `this` does not point back to the model object.**

A common convention that's pretty ugly in my opinion is to first declare a variable called `that` and assign it to `this` so that, inside the callback function, you can still access the variable `that` even after losing context:

```
// ...  
pluck(...props) {  
  const that = this;  
  return props.map(function(prop) {  
    return that.get(prop);  
  });  
}  
// ...
```

- **1 Get a reference to this that the callback can use.**
- **2 Using the that reference**

This `that equals this` monkey business works, but it doesn't make the code any more readable. There's actually a final argument to most higher-order functions like `this` that allows for setting the invocation context. A better version would be like this:

```
// ...
pluck(...props) {
  return props.map(function(prop) {
    return this.get(prop);           1
  }, this);                         2
}
// ...
```

- **1 Setting the context to this (the model object)**
- **2 The `this` keyword now points to the Model object.**

This is already much better than the `that equals this` solution, but you can skip this whole contextual step by using an arrow function. An arrow function's context is bound directly to the context it's defined in. What that means is the keyword `this` inside of an arrow function is always the same as it is outside of the arrow function. So a version using the arrow function would look like this:

```
// ...
pluck(...props) {
  return props.map( prop => this.get(prop) );  1
}
// ...
```

- **1 The `this` keyword still points to the model object.**

Just for fun, compare this to how it might look with all of the JavaScript. Next features removed:

```
// ...
```

```
pluck: function() {
  var props = [].slice.call(arguments);
  return props.map(function(prop) {
    return this.get(prop);
  }, this);
}
// ...
```

Wow, what a difference it makes for the readability of even a small method like this. Imagine how much more readable an entire app could become, and you're just getting started!

Quick check 17.2

Q1:

Reimplement the exponent method from the priming exercise to use arrow functions.

QC 17.2 answer

A1:

```
const exponential = {
  exponent: 5,
  calculate(...numbers) {
    return numbers.map( n => Math.pow(n, this.exponent) );
  }
}
exponential.calculate(2, 5, 10); // [32, 3125, 100000]
```

17.3. ARROW FUNCTION GOTCHAS

An important thing we should note about arrow functions is that they're always a function expression, not a function definition:

<code>function double (number) { return number * 2; }</code>	1
<code>const double = function (number) { return number * 2; }</code>	2
<code>const double = number => number * 2;</code>	3

- **1 Function definition**
- **2 Function expression**
- **3 Arrow function (equivalent to function expression)**

This means that arrow functions can't hoist like function definitions:

<code>const ids = getIds()</code>	1
<code>const items = getItems()</code>	2
<code>function getIds() { //... }</code>	
<code>const getItems = () => { //... }</code>	

- **1 Works because function definitions are hoisted**
- **2 ReferenceError because constants can't be accessed before being declared**

This throws a reference error because `const` and `let` variables can't be accessed before they're declared. But `var` variables can be accessed before they're declared, yet are always undefined. So even if you used a `var` to

create an arrow function, you would still get an error if you tried to use the arrow function before it was declared, only now it would be a type error:

```
const ids = getIds()           1
const items = getItems()       2

function getIds() {
  //...
}

var getItems = () => {
  //...
}
```

- **1 Works because function definitions are hoisted**
- **2 TypeError undefined is not a function.**

There's another common situation with arrow functions that can be puzzling. This is when you try to implicitly return an object literal from an arrow function. You would probably write something like this:

```
const getSize = () => { width: 50, height: 50 };  1
```

- **1 SyntaxError**

This would actually be a syntax error. Anytime the `=>` of the arrow function is followed by curly braces, it means that those curly braces are the opening and closing of the function body (no matter how many expressions). To get around this, you can wrap the object literal in parentheses:

```
const getSize = () => ({ width: 50, height: 50 });  1
```

- **1 Works as expected**

Finally, arrow functions can't have their context changed with `bind`, `call`, or `apply`. This means that if you're using a library that attempts to change the context of

callback functions, it could lead to bugs. For example, jQuery changes the context of functions supplied to `$.each()` so that `this` refers to the given DOM node like so:

```
const $spans = $('span');
$spans.each(function() {
  const $span = $(this);           1
  $span.text( $span.data('title') );
});
```

- **1 jQuery sets this to point to the individual span DOM node.**

jQuery achieves this by invoking the callback with either `call` or `apply` and setting the context to the given node. If you were to use an arrow function, jQuery wouldn't be able to change the context:

```
const $spans = $('span');
$spans.each(function() {
  const $span = $(this);           1
  $span.text( $span.data('title') );
});
```

- **1 jQuery couldn't set this so `$(this)` may not work.**

If you can avoid these traps, arrow functions will become a great new tool in your developer tool belt.

Quick check 17.3

Q1:

What will be logged to the console?

```
console.log(typeof myFunction);
var myFunction = () => {
  //...
}
```

QC 17.3 answer

A1:

undefined

SUMMARY

In this lesson, you learned about the syntax and mechanics of arrow functions.

- Arrow functions are a concise way to write functions.
- Arrow functions use the operator `=>` after the arguments list, as opposed to the keyword `function` before the arguments list.
- An arrow function's curly braces are optional when there's only one expression in the function body.
- Arrow functions implicitly `return` when the curly braces are omitted.
- An arrow function's context (`this`) is bound to the context it's defined in.

Let's see if you got this:

Q17.1

Here you have a `translator` function that, when called with a country code, returns a template literal tagging function that translates interpolated values to the corresponding language using the `TRANSLATE` function. The `TRANSLATE` function is just a mock to test that the `translator` function works. Building an actual translating library is beyond the scope of this book. Reimplement the `translator` function to use arrow functions. There should be three total arrow functions used:

```
function TRANSLATE (str, lang) {  
  // mock translator as building a real translator is  
  beyond  
  // the scope of this book.  
  switch (lang) {  
    case 'sv':  
      return str.replace(/e/g, 'ë');  
    break;  
    case 'es':
```

```
        return str.replace(/n/g, 'ñ');
    break;
    case 'fr':
        return str.replace(/e/g, 'é');
    break;
}
}

const translator = function (lang) {
    return function(strs, ...vals) {
        return strs.reduce(function(all, str) {
            return all + TRANSLATE(vals.shift(), lang) + str;
        });
    }
}

const fr = translator('fr');
const sv = translator('sv');
const es = translator('es');

const word = 'nice';

console.log( fr`${word}` ); // nicé
console.log( sv`${word}` ); // nicē
console.log( es`${word}` ); // ñice
```

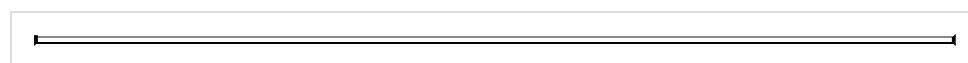
Lesson 18. Generator functions

After reading [lesson 18](#), you will

- Know how to define generator functions
- Know how to yield values from generator functions
- Understand the generator function lifecycle

Generator functions are one of the harder things to comprehend among all the recent additions to JavaScript. They introduce a new form of code execution and processing that most JavaScript developers haven't seen before. Generators aren't new concepts, though—they're already part of Python, C#, and other languages. This lesson is only meant to be a gentle introduction to the topic.

You'll see how good generators are at creating lists, but that isn't all generators are good for. That would be like saying objects are only useful for storing key/value pairs. To the contrary, generators, like objects, are a powerful multipurpose tool that can be used for all sorts of problems. For example, we'll see how useful they can be for working with iterable and asynchronous tasks later on in [Units 5](#) and [7](#). For now, though, you need to master the basics.



Consider this

Inside of a function, once you `return` a value, the rest of the function is no longer processed and the function exits. This means the `return` statement stops the function from going any further. What if you could return a value, but only pause the function at that point instead of exit, and then later tell the function to

continue where it left off?

18.1. DEFINING GENERATOR FUNCTIONS

A generator function is a special type of function. It's a factory for returning new generator objects. Each generator object returned from a generator function behaves according to the body of the function it came from. That is, the body of the generator function defines the blueprint for each generator that it returns:

```
function* myGeneratorFunction() {           1
  // ...
}

const myGenerator = myGeneratorFunction();  2
```

- **1 A generator function is denoted with an asterisk.**
- **2 The generator function is invoked without new.**

To denote that a function is a generator function, it's defined with an asterisk. The asterisk can either be touching the name of the function like in the above example or touching the keyword `function`, such as `function* . . .`. It seems like the latter is becoming more popular, although I like the former because it's consistent with concise generator functions on object literals. Finally, the asterisk can also have spaces on both sides, such as `function * myFunction`:

```
const myObj = {
  * myGen() {           1
    // ...
  }
}
```

- **1 A concise generator method on an object literal**

The code inside the generator function behaves like a regular function with a single, albeit huge caveat:

yielding. There's a new keyword in JavaScript only to be used inside of a generator function: `yield`. This `yield` creates a two-way communication channel between the inside and outside of the function.

Whenever a `yield` is encountered, the execution of the function halts, returning control back to the external code that invoked the function. The `yield` can also pass along a value to the outside process, similar to the `return` statement:

```
function* myGeneratorFunction() {  
  // ...  
  const message = 'Hello'  
  yield message;           1  
  // ...  
}
```

- **1 The execution will halt here and return the value Hello to the containing process.**

Unlike the `return` statement, `yield` is actually an expression that can capture a value to be used later in the function:

```
function* myGeneratorFunction() {  
  // ...  
  const message = 'Hello'  
  const response = yield message;    1  
  // ...  
}
```

- **1 Capturing a value from the containing process**

This detail is important because it's unlike anything else in JavaScript. When that `yield` occurs, the body of the function isn't short-circuited, as happens when you `return` a value. Instead it halts, returns an object with `value` and `done` properties, then waits for the containing process to tell it to continue. At this time, the containing process may pass a value back in, which you

capture here as the response. This is the two-way communication channel that generator functions create. See [figure 18.1](#).

If you're confused, that's OK: you need to understand how generators work from both the inside of the function and the external process before you can really begin to understand them. So far we've only talked about the internal behavior, which won't make sense until contrasted with the external behavior. We'll go over this next.

Figure 18.1. A two-way communication channel

```
const myGenerator = myGeneratorFunction();

let gotA = myGenerator.next(0);
console.log('gotA:', gotA);

let gotB = myGenerator.next(1);
console.log('gotB:', gotB);

let gotC = myGenerator.next(2);
console.log('gotC:', gotC);
```

```
function *myGeneratorFunction() {
  console.log('Started');
  let recievedA = yield 'a';
  console.log('recievedA:', recievedA);
  let recievedB = yield 'b';
  console.log('recievedB:', recievedB);
}
```

A When the generator function is called, a new generator object is returned. The code that makes up the generator function doesn't execute yet.

B The first `next()` starts the execution of the code up to the first `yield`. The value passed to `yield` becomes the `value` property in the object returned from `next()`.

C The second call to `next()` starts the execution of the code again, starting from where it left off. It executes until it encounters another `yield`. The value passed to `yield` again becomes the `value` property in the object returned from `next()`.

D The third call to `next()` starts the execution of the code again. And again, it starts from where it left off. If it had encountered another `yield`, it would have stopped again and returned the yielded value, but this time it never encountered a `yield`, so it ran to completion. If the function had returned a value, it would have been the `value` property in the object returned to the final `next()`.

Quick check 18.1

Q1:

Which generator function declaration is using the correct syntax?

```
function* a() { /* ... */ }
function * b() { /* ... */ }
function *c() { /* ... */ }
```

QC 18.1 answer

A1:

They are all valid.

18.2. USING GENERATOR FUNCTIONS

When you invoke a generator function, you get back a new generator object. This generator object begins in a halted state and doesn't do anything until you call `next()` on it, which instructs it to start executing the code in the body of the generator function:

```
function* myGeneratorFunction() {  
  console.log('This code is now running');  
}  
  
myGeneratorFunction();           1  
  
const myGenerator = myGeneratorFunction();  
myGenerator.next();             2  
                                3
```

- **1 The log isn't evaluated here.**
- **2 The log isn't evaluated here, either.**
- **3 Only now is the log evaluated.**

If the generator function body encounters a `yield`, it will halt again and won't continue until you call `next()` again on the generator object:

```
function* myGeneratorFunction() {  
  console.log('A');  
  yield;  
  console.log('B');  
}  
  
const myGenerator = myGeneratorFunction();  
myGenerator.next();           1  
myGenerator.next();           2
```

- **1 The first log now occurs and then the generator halts again at the yield.**
- **2 The second log now occurs.**

Every time you call `next()`, you get back an object with two properties: a `value` property that contains any value

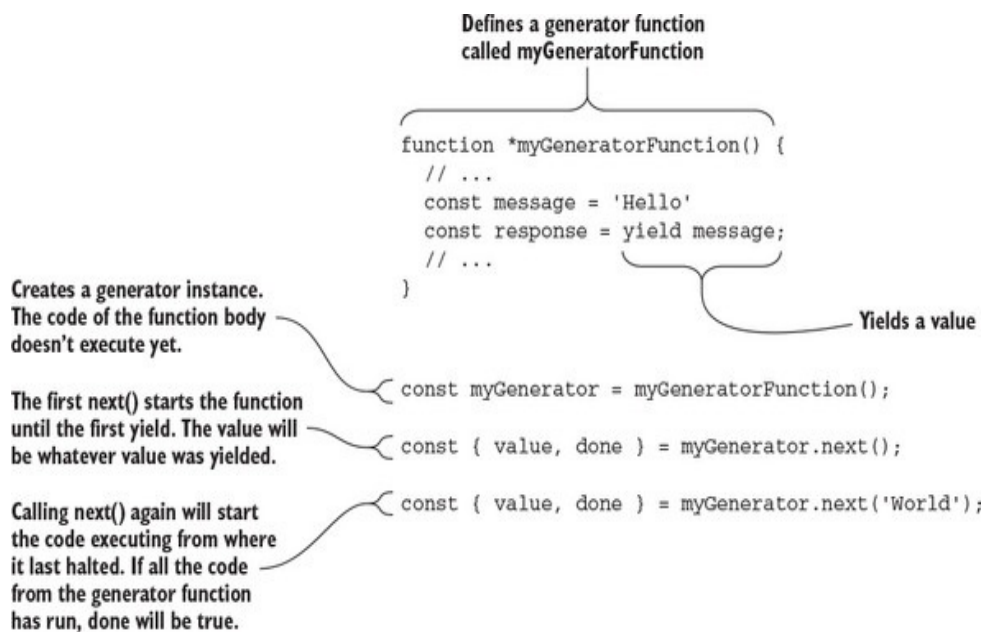
that was yielded and a done property that indicates whether the generator is halted or finished. See the following listing and [figure 18.2](#).

Listing 18.1. A generator function in action

```
function* myGeneratorFunction() {  
  console.log('Started')  
  let recievedA = yield 'a';  
  console.log('recievedA:', recievedA);  
  let recievedB = yield 'b';  
  console.log('recievedB:', recievedB);  
}  
  
const myGenerator = myGeneratorFunction();  
  
let gotA = myGenerator.next(0);           1  
console.log('gotA:', gotA);               2  
let gotB = myGenerator.next(1);           3  
console.log('gotB:', gotB);               4  
let gotC = myGenerator.next(2);           5  
console.log('gotC:', gotC);               6
```

- **1 Started**
- **2 gotA: { value: “a”, done: false }**
- **3 recievedA: 1**
- **4 gotB: { value: “b”, done: false }**
- **5 recievedB: 2**
- **6 gotC { value: undefined, done: true }**

Figure 18.2. Generator function execution order



Notice how the final object returned from the generator has `done` set to `true` and `value` set to `undefined`. `done` being `true` means that this generator has reached the end of its function and has no more code to execute. This can be used by code that's trying to process the generator to determine when to stop calling `next()` on it. The only time the `value` property will have anything other than `undefined` when `done` is if the generator actually returned (not yielded) a value at the end.

Conversely, notice how the `0` passed in to the initial `next()` was never used. This is because the first `next()` isn't tied to a `yield` like all thereafter are. The first `next()` is what starts the generator from its initially halted state. If you do need to pass an initial value, you can do so using regular function parameters:

```
function* myGeneratorFunction(initialValue) {  
  // ...  
}  
  
const myGenerator = myGeneratorFunction(0);
```

OK, that was a lot to grasp. Feel free to read through this

section again or play with your own generator functions until it clicks. I don't expect you to understand why or when you would use a generator function yet. You just had to get through all this completely new functionality before you could dive into some useful applications.

Quick check 18.2

Q1:

Create a generator function that yields two different values before completing.

QC 18.2 answer

A1:

```
function* simpleGen() {  
  yield 'Java';  
  yield 'Script';  
}
```

18.3. CREATING INFINITE LISTS WITH GENERATOR FUNCTIONS

Some languages like Haskell allow for the creation of infinite lists because they are lazy evaluated. This means that technically the list contains infinite values, but each value is only allocated to memory when it's requested. JavaScript, on the other hand, is eagerly evaluated. So if you tried to make an array with infinite values, it would attempt to add them all to memory until it ran out of memory. You can easily simulate this, though, with generator functions. Simulate an infinite list from 0 to infinity, as shown in the next listing.

Listing 18.2. Generating an infinite list with a generator function

```
function* infinite() {  
  let i = 0;  
  while (true) {  
    yield i++;  
  }  
}  
  
const list = infinite();  
console.log( list.next().value );      1  
console.log( list.next().value );      2  
console.log( list.next().value );      3
```

- **1 Logs 0**
- **2 Logs 1**
- **3 Logs 2**

As you continue to call `next()`, you get the next value from your infinite list. Also notice how you used a `while(true)` that you never stop. If this were a regular function, this would create an infinite loop and crash. However, since it encounters a `yield` each time, it halts and only yields a single value at a time. So you're now lazily evaluating only the items that are requested.

Define a small helper function to assist in taking values from your infinite lists:

```
function take(gen, count) {  
  return Array(count).fill(1).map(  
    () => gen.next().value  
  );  
}  
  
take(infinite(), 7);           1
```

- **1 [0, 1, 2, 3, 4, 5, 6]**

OK, now you'll make a more interesting infinite list of numbers. The Fibonacci sequence begins with the numbers zero and one, and is defined by each integer being the sum of the previous two integers in the sequence. So if you start with 0 and 1, it continues with 1, 2, 3, 5, 8 and so on, as shown in the next listing.

Listing 18.3. Generating a Fibonacci sequence

```
function* fibonacci() {  
  let prev = 0, next = 1;  
  
  for (;;) {  
    yield next;  
    let tmp = next;           1  
    next = next + prev;      2  
    prev = tmp;              3  
  }  
}  
  
take(fibonacci(), 7);       4
```

- **1 First grab a reference to the unchanged next value.**
- **2 Add the unchanged prev value to next.**
- **3 Finally, update prev to your tmp value.**
- **4 [1, 1, 2, 3, 5, 8, 13]**

Awesome! However, you can use array destructuring to update your prev and next values without having to rely on an intermediate tmp value:

```
function* fibonacci() {  
  let prev = 0, next = 1;  
  
  for (;;) {  
    yield next;  
    [next, prev] = [next + prev, next];  
  }  
}
```

Hopefully by now you can see how easy it is to create a list with a generator. This is no accident, and you'll soon discover why in [unit 5](#), which covers iteration.

Quick check 18.3

Q1:

Implement a generator function that creates an infinite list of all the natural odd numbers.

QC 18.3 answer

A1:

```
function* odd() {  
  let i = 1;  
  while (true) {  
    yield i;  
    i += 2;  
  }  
}
```

SUMMARY

This lesson presented a gentle introduction to the syntax and lifecycle of generator functions.

- A generator function is defined with a asterisk.
- A generator function returns a generator object.
- A generator can yield values with the `yield` keyword.
- A generator pauses at each `yield` until told to continue by calling `next()`.
- You can pass values back to a generator as arguments to `next()`.

Let's see if you got this:

Q18.1

Create a generator for an infinite list of dates. The first time `next()` is called, it should yield the current day. Each subsequent call should yield the next day. Alternatively, you could make the generator function accept an argument that tells it what date to start at and only default to the current day.

Lesson 19. Capstone: The prisoner's dilemma

Let's pretend you've been given the honor of organizing a hack-a-thon—a programming competition where competitors are given a coding challenge they're expected to solve.

Being the lover of game theory that you are, you decide to use the prisoner's dilemma as the challenge for the hack-a-thon. How did I know you are a lover of game theory, you ask? Ahem, *who isn't*? But on the off chance that you're not, I'll explain the prisoner's dilemma. Two prisoners are interrogated in separate rooms, each asked to snitch on the other prisoner. Exactly what happens when each prisoner snitches on the other changes depending on who tells the story, but the basic idea is this. The best outcome for both prisoners is if neither snitches on each other. If one snitches on the other, it's good for the one who snitched but bad for the one getting snitched on. If they both snitch on each other, it's bad for both, but less severe than being the only one getting snitched on. For the purpose of the hack-a-thon, you decide the outcomes are as follows:

- If both prisoners snitch, they each serve one year.
- If only one prisoner snitches, that prisoner goes free and the other serves two years.
- If neither prisoner snitches, they both go free.

You'll explain to each contestant that they need to program a prisoner that gets the fewest total years in prison after being pitted against the other contestants' prisoners. You decide that this is a great hack-a-thon problem because it doesn't rely on subjective grading,

but on a metric that no one can argue with.

You want to do a proof of concept before the hack-a-thon to get an idea of the results you might see, so you need to

1. Figure out the Prisoner API: what the interface of the individual prisoner looks like
2. Figure out the Prisoner Factory API: what the interface for *generating* a prisoner looks like
3. Build a couple of different prisoner factories to battle
4. Build a simulation that takes two prisoners, compares their actions, and keeps score
5. Build a way to present the scores at the end

19.1. GENERATING PRISONERS

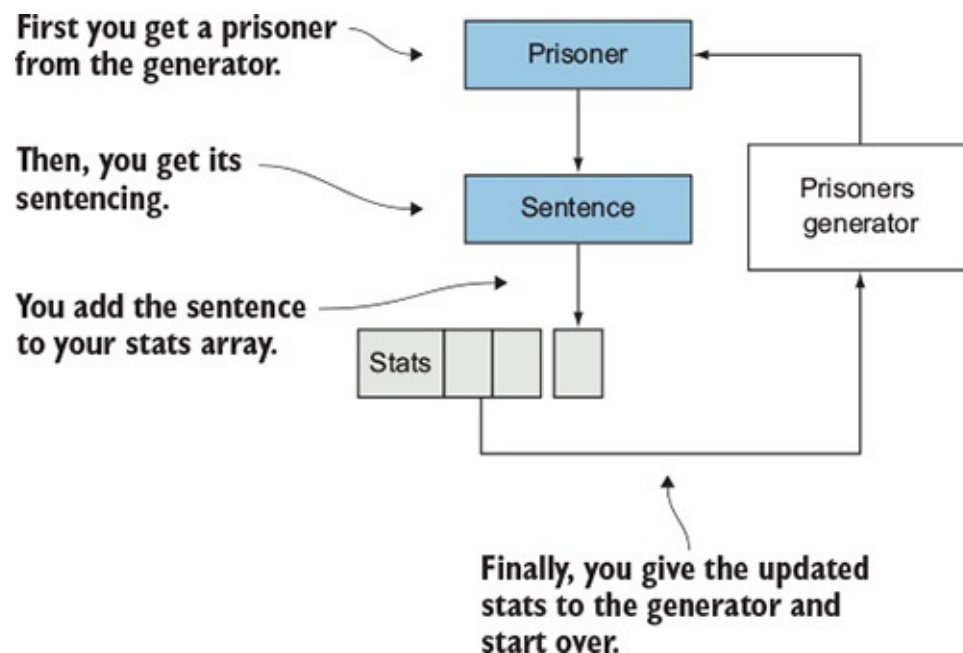
In order to run each prisoner through a simulation, each contestant's submission would have to implement a common interface that all will adhere to. You need an interface for generating prisoners; you'll call this the Prisoner Factory API. The Prisoner Factory API will produce prisoner objects. You need to figure out the interface of each prisoner as well. You'll call this simply the Prisoner API.

The Prisoner API is going to be really simple: it just needs a method called `snitch` that returns `true` or `false` indicating whether or not the prisoner snitched. You'll also add another method called `sentencedTo` to each prisoner that will allow you to dole out a prison sentence to the prisoner. This won't be a part of the API that you ask the contestants to build, though, because you don't want the contestants handling the prison sentencing.

The Prisoner Factory API is a little more complicated because it should always generate a new prisoner for you to interrogate, but it needs to be able to allow the prisoners that are generated to adapt in real time. To do this, every time you request a prisoner from the factory, you'll provide the factory with stats of how well they've done so far. The stats will be an array of numbers indicating the number of years each of the previous prisoners has been sentenced to. If there was no jail time served, it would be a 0. This will allow the factories to adapt in real time, altering the prisoner they produce if they're doing well or poorly. This means that for each interrogation, you'll need to pass data (their stats) to the

factory before getting back another prisoner object. For this two-way communication it makes sense to use a generator, since the generator can yield prisoners to your simulation and your simulation can pass back statistics (see figure 19.1).

Figure 19.1. Prisoner lifecycle



Define a generator that yields a prisoner:

```
function* prisoner() {
  for(;;) {
    yield {
      snitch() {
        return true
      }
    }
  }
}
```

This prisoner generator always yields a new prisoner. The prisoner is an object with a `snitch` method. The contestants won't know how many times their submission is going to be run through the simulator, so it makes sense for this generator to yield an unlimited number of prisoners by using the never-ending `for(;;)` loop. You could test this out like so:

```
const prisoners = prisoner()  
prisoners.next().value.snitch()      1
```

- **1 true**

It appears this is a prisoner that always snitches. But you still need to pass the previous outcomes back. If you pass back an array of all the previous outcomes every time you call `next`, the prisoner generator can capture these when it yields like so:

```
function* prisoner() {  
  let stats = []      1  
  for(;;) {  
    stats = yield {    2  
      snitch() {  
        return true  
      }  
    }  
  }  
}
```

- **1 The first prisoner yielded wouldn't have any stats yet, so you default stats to an empty array.**
- **2 After each prisoner is yielded, you get back the stats from all previous outcomes.**

Now before yielding a prisoner, the code can take a look at the previous outcomes and use this information to guide its decision on whether or not to snitch. Of course, this always-snitching prisoner isn't making use of these stats; you're going to change that. Make a prisoner who snitches if they recently had to serve jail time, but who otherwise doesn't snitch:

```
function* prisoner() {  
  let stats = []  
  for(;;) {  
    stats = yield {  
      snitch() {  
        return stats.pop() > 0      1  
      }  
    }  
  }  
}
```

```
}
```

- **1 If the prisoner went to jail last time, snitch; otherwise don't.**

The final piece of the Prisoner Factory API is a way for the contestants to name their submission. This way your simulation can tell you which prisoner has won the contest. To achieve this, you decide that each Prisoner Factory should be an object with a `name` property and a `generator` property. So the previous example would be changed as shown in the next listing:

Listing 19.1. A Prisoner Factory with name and generator

```
const retaliate = {
  name: 'retaliate',
  *generator() {
    let stats = []
    for(;;) {
      stats = yield {
        snitch() {
          return stats.pop() > 0
        }
      }
    }
  }
}
```

19.2. GETTING PRISONERS TO INTERACT

That about wraps up the Prisoner and Prisoner Factory APIs that each contestant needs to use to build their entry. Now you need a way to pit each prisoner against every other prisoner in an interrogation and see which one gets the least amount of prison time. To do this, define an `interrogate` function that takes two prisoners, asks each to snitch, and then doles out prison sentences, as shown in the following listing.

Listing 19.2. `interrogate` function pits two prisoners against each other

```
function interrogate(criminal, accomplice) {  
  
    const criminalsntched = criminal.snitch()  
    const accomplicesntched = accomplice.snitch()  
  
    if (criminalsntched && accomplicesntched) {           1  
        criminal.sentencedTo(1)  
        accomplice.sentencedTo(1)  
    } else if (criminalsntched) {                         2  
        criminal.sentencedTo(0)  
        accomplice.sentencedTo(2)  
    } else if (accomplicesntched) {                       3  
        criminal.sentencedTo(2)  
        accomplice.sentencedTo(0)  
    } else {                                             4  
        criminal.sentencedTo(0)  
        accomplice.sentencedTo(0)  
    }  
}
```

- **1 They both snitched; each gets one year.**
- **2 Only criminal snitched; they go free while accomplice gets two years.**
- **3 Only accomplice snitched; they go free while criminal gets two years.**
- **4 Neither snitched; they both go free.**

This `interrogate` function takes two prisoners, asks each to snitch, and then doles out prison sentences according to the predefined rules. Notice that you're now

calling the `sentenceTo` method on each prisoner.
Remember you didn't want the contestants
implementing this method themselves to spare them the
urge to cheat, so you'll need to add that method to each
prisoner yourself.

19.3. GETTING AND STORING THE RESULTS

In order to attach your `sentencedTo` to each prisoner, you'll write your own generator function that takes the Prisoner Factory as a parameter, internally grabs prisoners from the factory's generator, and yields them after adding your `sentencedTo` method:

```
function* getPrisoners({ name, generator }) {  
  1  
    const stats = [], prisoners = generator()  
  2  
  
    for(;;) {  
        const prisoner = prisoners.next(stats.slice()).value  
  3  
  
        yield Object.assign({}, prisoner, {  
  4  
            sentencedTo(years) {  
                stats.push(years)  
  5  
            }  
        })  
    }  
  }  
}  
  
const wrappedRetaliate = getPrisoners(retaliate)  
  6  
  
const prisoner = wrappedRetaliate.next().value  
  7  
  
if (prisoner.snitch()) {  
    prisoner.sentencedTo(10)  
  8  
}
```

- **1 You can grab the name and generator directly with parameter destructuring.**
- **2 The stats array will keep track of previous outcomes**
- **3 By calling `.slice` you pass a copy.**
- **4 Add the `sentencedTo` method to a copy of the prisoner.**
- **5 When the prisoner is sentenced, add it to your stats.**
- **6 Wrap your previous `retaliate` factory in your**

getPrisoners generator.

- **7 Grab prisoners just like before.**
- **8 The produced prisoners now have a sentencedTo method.**

This function wraps the contestants' code and yields each contestant's prisoner after adding the needed `sentencedTo` method. The `sentencedTo` method adds the outcome (or sentence) to an array keeping track of all the outcomes. These are the stats that get passed back to the prisoners generator, allowing it to adapt. You call `stats.slice()` when passing them back to the contestant's code. This makes a copy to make sure the contestant's code cannot alter your statistics.

Your simulation needs to be able to get the statistics and the name back from this function as well. That's tricky because this function is yielding the prisoner every time you call `next`. You could make it yield the name, stats, and prisoner every single time, but that's unnecessary as you really only need the name and stats once you're done asking for prisoners. So when you're done asking for prisoners, you can pass `true` in your last `next` invocation, indicating that you're finished. That can tell the `getPrisoners` generator that you no longer want prisoners, and instead would like the stats and name of the prisoner you were testing, as shown in the next listing.

Listing 19.3. Getting stats and prisoner name from the generator

```
function* getPrisoners({ name, generator }) {  
  const stats = [], prisoners = generator()  
  
  for(;;) {  
    const prisoner = prisoners.next(stats.slice()).value  
  
    const finished = yield Object.assign({}, prisoner, {  
1
```

```
        sentencedTo(years) {  
            stats.push(years)  
        }  
    })  
  
    if (finished) {  
        break  
    }  
  
    return { name, stats }  
}
```

- **1 A finished flag is captured after each yield.**
- **2 Break out of the for loop once finished.**
- **3 The final value will be the name and stats.**

19.4. PUTTING THE SIMULATION TOGETHER

Now you need to write a test function that actually does the simulation of taking the contestants' prisoners and interrogating them against each other, as the next listing shows.

Listing 19.4. Putting it all together

```
function test(...candidates) { 1

  const factories = candidates.map(getPrisoners) 2
  const tested = []

  for(;;) {
    const criminal = factories.pop()
    tested.push(criminal)

    if (!factories.length) { 3
      break
    }

    for (let i = 0; i < factories.length; i++) {
      const accomplice = factories[i]
      interrogate(criminal.next().value,
        accomplice.next().value)
    }
  }

  return tested.map(testee => testee.next(true).value) 4
}
```

- **1 Use rest to get any number of prisoners to test.**
- **2 Map each prisoner with the getPrisoner wrapper.**
- **3 Break out of the loop after testing each prisoner against every other.**
- **4 Get the final values and return them.**

This test function uses rest to allow any number of contestants' prisoners to be tested. By popping each prisoner out of the array and testing them against all the prisoners remaining in the array, you ensure that you test every prisoner against every other prisoner. You then push that prisoner onto a tested array so you can

return the results at the end. Quickly define some other prisoners so you can test them against each other, as shown in the following listing.

Listing 19.5. Defining more prisoners

```
const never = {
  name: 'never',
  * generator() {
    for(;;) {
      yield {
        snitch() {
          return false
        }
      }
    }
  }
}

const always = {
  name: 'always',
  * generator() {
    for(;;) {
      yield {
        snitch() {
          return true
        }
      }
    }
  }
}

const rand = {
  name: 'random',
  * generator() {
    for(;;) {
      yield {
        snitch() {
          return Math.random() > 0.5
        }
      }
    }
  }
}
```

Now you have three other prisoners to pit against your retaliation prisoner: a `never` prisoner that never snitches, an `always` prisoner that always snitches, and a `rand` prisoner that randomly snitches. Lets see how they do against each other:

```
test(retaliate , never, always, rand);
```

1

- **1 [Object, Object, Object, Object]**

19.5. WHICH PRISONER DOES BEST?

Woohoo! You successfully tested your prisoner entries (hopefully). But it was rather hard reading the results in their current format. Currently you get back an array of objects; each object has a name property array of all the results (each result being a number of years sentenced). It would be much easier to read if the result were a single object with the names of each prisoner type as keys and the sum of all the years sentenced as values. Write a helper function to convert your results into that format:

```
function getScore({ value }) {  
  const { name, stats } = value  
  const score = stats.reduce( (total, years) => total + years)  
  return {  
    [name]: score  
  }  
}
```

- **1 Assign the property as the prisoner name using a computed property name.**

This `getScore` function takes the last yielded value from a prisoner, sums the total years, and returns an object with the prisoner name as the property and the sum as the value. You'll need to update the `test` function to map each result into the `getScore` function to format it:

```
function test(...candidates) {  
  
  // ... omitted for brevity  
  
  return Object.assign.apply({}, tested.map(criminal => (  
    getScore(criminal.next(true)  
  ))))  
}
```

Run your test again:

```
test(retaliate , never, always, rand);
```

1


- **1 {random: 2, always: 0, never: 6, retaliate: 2}**

So far it looks like it pays to always snitch. But you can enhance your simulator by making each prisoner get interrogated by every other prisoner for 50 times instead of just once. And you'll also make each prisoner get pitted against themselves. First define a quick helper function to execute a callback 50 times:

```
function do50times(cb) {  
  for (let i = 0; i < 50; i++) {  
    cb()  
  }  
}
```

Finally, update the `test` function to make your final changes, as shown in the following listing.

Listing 19.6. Interrogating the prisoners 50 times

```
function test(...candidates) {  
  
  const factories = candidates.map(getPrisoners)  
  const tested = []  
  
  for(;;) {  
    const criminal = factories.pop()  
    tested.push(criminal)  
  
    do50times(() => interrogate(criminal.next().value,  
criminal.next().value))1  
  
    if (!factories.length) {  
      break  
    }  
  
    for (let i = 0; i < factories.length; i++) {  
      const accomplice = factories[i]  
      do50times(() =>  
        interrogate(criminal.next().value,  
accomplice.next().value)) 1  
    }  
  }  
  
  return Object.assign.apply({}, tested.map(criminal => (  
    getScore(criminal.next(true)  
  ))))  
}
```



```
}
```

- **1 Your changes**

Now you're testing each prisoner against every other prisoner for 50 times, including testing against themselves. Run test one final time:

```
test(retaliate , never, always, rand);
```

1

- **1 {random: 187, always: 177, never: 156, retaliate: 90}**

Now it's looking much better to retaliate. It makes sense that the only prisoner who made use of the stats ended up being the one with the best solution.

SUMMARY

In this capstone you created a Prisoner's Dilemma simulation runner. The point of the runner was to allow contests to program their own prisoner simulations, to see what kind of prisoner fares best in the prisoner dilemma scenario. You only made a few simple prisoner simulations; you could easily take this further by writing a more thoughtful prisoner simulation and see how well it does against the prisoner simulations you wrote in this capstone.

Unit 4. Modules

Until recently, most front-end applications followed the same paradigm for using libraries: *exposing globals*. For example, if someone wanted to use jQuery (<https://jquery.com>) in a project, they would need to go to jQuery's website, download either `jquery.js` or `jquery.min.js`, add it to their project's `js` or `vendor` folder, and then include a `<script>` tag importing the library into their application. This was not ideal, because it required modules to export themselves using global variables that could potentially collide with other global variables. It also required dependencies to be included before any library that relied on them or there would be an error.

Modules solve these problems by allowing scripts to be included by a module loader. The modules themselves can specify and load their dependencies, and thus relieve the application author from having to manage them.

Modules allow for dividing large amounts of code into small, cohesive units. Each module is contained in its own file. Anything in the file that isn't exported is private, without needing to be wrapped in an immediately invoked function expression (IIFE) to create a private scope. Because the code inside a module doesn't attach anything to the global namespace, any code outside of the module must explicitly import what it needs from the module in order to gain a reference to it.

Using modules makes your code easy to reason about, because you know exactly where every value came from. You either defined it directly in your module, or you imported it from somewhere else, so you don't ever find

yourself looking at a variable and wondering where it was defined. You also don't have to worry about what parts of your module are being used elsewhere. If the value isn't exported, it's private, and if the value is exported, it's likely used by something else.

Just like breaking your code into many small functions, breaking your application into many small modules will make your program easy to reason about and keep it maintainable as it grows in complexity.

We'll start this unit by looking at how to create your own modules. Then we'll take a look at various ways of importing and combining modules. Finally, you'll wrap up the unit by creating a hangman game and seeing how much cleaner your code can be when using modules.

Lesson 20. Creating modules

After reading [lesson 20](#), you will

- Understand what a module is
- Know how JavaScript behaves in modules
- Create modules and export values

Before you learn how to create and use ES2015 modules, let's define what a module in JavaScript is. At its most basic, a module is a JavaScript file that has its own scope and rules, and may import or export values to be used by other modules. A module isn't an object: it has no data type, and you can't store one in a variable. It's simply a tool to break apart, encapsulate, and organize your code.

Modules are a great way to separate your logic into cohesive units and share logic across files without the cumbersome use of global variables. They also allow you to import only what's needed, keeping the cognitive load down and making maintenance easier.

Consider this

Imagine you're writing a large application. Would you use a single file for the entirety of the code or break it up into several smaller files? How would you make the various components of your application communicate and share resources across files?

20.1. MODULE RULES

The rules of JavaScript are slightly different inside a module. The code is always executed in strict mode, so you never need to add `"use strict";`. The whole reason the `use strict` string became popular was to allow scripts to opt-in to strict mode. This is because if browsers flipped a switch and started running everything in strict mode, many legacy applications would break. Having to opt-in to strict mode was a way to preserve backward compatibility and prevent breaking the web. But modules are a completely new context, so there's no need to make them backward-compatible. Because of this, it was decided that modules would always be in strict mode.

Another difference is what the `this` refers to at the root context. In normal JavaScript, `this` will default to the global object (`window` in the browser) at the root level. In an ES2015 module though, it will be undefined:

```
var obj = {
  foo() {
    return this
  }
}
function bar() {
  return this
}
obj.foo()           1
bar()              2
console.log(this)   2
```

- **1 obj**
- **2 undefined**

The value returned from `obj.foo()` is `obj`, just like it would be outside of a module. But outside of a module,

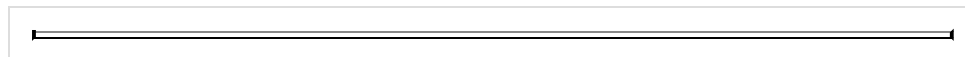
`bar ()` would return `window` or the global object because it's invoked without a context. In a module, though, `bar ()` being invoked without a context would return `undefined`. Additionally, any reference to `this` at the root level inside a module would also be `undefined` instead of `window`, like it would outside of a module.

Normally in JavaScript, any variable defined at the root scope is automatically set as a global variable. Many JavaScript developers in the past would define their variables inside of an IIFE (immediately invoked function expression) to prevent the variables from becoming global. You also learned in the first unit of this book that in ES2015 you can now use a simple block `{ }` to add a scope and prevent variables from becoming global. But in a module, variables defined are never global. You can imagine your entire module is running inside of a block `{ }` or IIFE. This is because modules are designed to expose only what they explicitly export to the rest of the world. You can still attach values to the global object inside of a module, though:

```
window.someName = 'my value'
```

In most cases, though, setting globals from a module would be a bad practice.

Now that you know the rules of how code works in modules, let's take a look at making some in the next section.



Quick check 20.1

Q1:

Inside of a module, if you were to call `obj . foo` like

the following, what would be the result?

```
obj.foo.call(this)
```

QC 20.1 answer

A1:

It would be undefined because you set the context to this at the root level and in a module this is undefined at the root level.

20.2. CREATING MODULES

Imagine you're writing an application and you want a reusable function that can log statistics to a database. You want this function to be used all over the place. You would probably just set it as a global function like this:

```
window.logStats = function(stat) {  
  // log stat to data base  
}
```

There are several reasons why global variables are a bad practice, but for many years in JavaScript, we didn't have a choice. But now with modules, you could put this function in a module and share it across your application without making it a global:

```
export default function logStats(stat) {  
  // log stat to data base  
}
```

By putting the keyword `export` in front of your function, you're saying that this module exports the `logStats` function. You also specify that this is the *default* export; more on that in a bit.

Now say your `logStats` function needs a helper function. The `logStats` function needs access to the helper, but you don't want to expose it to anything else. Again, without modules you would now have to resort to wrapping everything in a private context to prevent leaking global values:

```
{  
  function statsHelper() {  
    // do some stat processing  
  }  
  
  window.logStats = function(stat) {
```

```
    // log stat to data base using statsHelper
  }
}
```

But within a module, everything is already in its own scope and thus already protected from leaking values (unless explicitly exported or set as globals):

```
function statsHelper() {
  // do some stat processing
}

export default function logStats(stat) {
  // log stat to data base using statsHelper
}
```

When you create a module, you're generally focusing on a specific task. When you want to create a reusable piece of self-contained code, you can put it in a module (a separate file) and only expose what's needed while keeping most of the details of the module private. Often your module will only export a single value. In that case, it makes sense to export the single value as the default value:

```
export default function currency(num) {
  // ... return a formatted currency string
}
```

The function expression is completely normal; it's just prefixed with the statement `export default`. The syntax is `export default <expression>`. The value evaluated from the expression ends up being what's exported. So if you just wanted to export the number 5, you could do this:

```
export default 5
```

If you were to think of a module as a function, the default export would be analogous to the `return` value from the

function. A function can return only one value and a module can only have one default export. But a module isn't limited to a single default export: it can export several values, but only one can be the default. All other exports will be *named* exports. Going back to your currency function, if you wanted to export it by name, all you would need to do is remove the `default` keyword:

```
export function currency(num) {  
  // ... return a formatted currency string  
}
```

Since you removed the `default` keyword, the function is exported by name. But notice you don't specify what name. The name gets automatically picked up and exported as `currency`.

The syntax is `export <declaration>`. Did you spot the difference? With a default export, you're exporting an expression, which evaluates to a value: the value that's exported. But with a named export, you're exporting a declaration. A function declaration, variable declaration, or even a class declaration always has a name, and that's the name by which the value is exported. You can export as many declarations as you want with this syntax.

There's an alternate syntax for named exports like so:

```
function currency(num) {  
  // ... return a formatted currency string  
}  
  
export { currency }
```

This does the same thing but with a different syntax. The syntax is `export { binding1, binding2, ... }`. You can specify one name or a comma-separated list of as many as you want. When you export a value this way,

you must use a name that points to a value (a binding).
You can't export a raw value this way:

```
export { currency }           1
export { 'currency' }        2
export { 5 }                 2
```

- **1 OK**
- **2 Syntax Error**

The benefit of this syntax is that it allows you to specify an alternate name to use for the export. Say internally you're referring to a value as `formattedCurrentUsername` but you want to export it simply as `username`. You could do so like this:

```
export { formattedCurrentUsername as currency }
```

The syntax for this is `export { originalName as exportedName }`.

When exporting multiple values, you can either export them using individual export statements for each value as you declare them, or you can use one export statement and list all the names you want to export:

```
export function currency(num) {
  // ... return a formatted currency string
}

export function number(num) {
  // ... return a number formatted with commas
}
```

This is equivalent to

```
function currency(num) {
  // ... return a formatted currency string
}

function number(num) {
  // ... return a number formatted with commas
}
```

```
export { currency, number }
```

Whichever you decide is personal preference, although the majority of developers seem to prefer the first example.

The same thing works for variable declarations as it does for functions:

```
export let one = 1           1  
export const two = 2        2  
export 3                     3
```

- **1 Exported as the name one**
- **2 Exported as the name two**
- **3 Invalid. No way to infer name.**

The export statement must be at the top level, meaning you can't conditionally export values. By top level, I mean at the root of most of the body of code in the file, not inside the body of a block, `if` statement, function, and so on:

```
export const number = num => /* format number */           1  
  
if (currencyNeeded) {  
  export const currency = num => /* format currency */      2  
}  
  
function exportMore() {  
  export const date = dateObj => /* format date */          2  
}
```

- **1 Top level, valid**
- **2 Not top level, invalid**

This is by design: modules are meant to be statically analyzable and thus must always export and import the same way unconditionally. If a value were allowed to be exported in a function or `if` statement, that would mean

that sometimes a value could be exported and sometimes not. Modules are designed to always have their exports executed, so they must be at the root level.

Now when creating a module that exports multiple values, you may ask, should one of these be the default and the rest be named, or should all be named? There's no cut-and-dry answer to that question. It really depends on how you're designing your module. I would say as a rule of thumb, if everything you're exporting is at the same level of importance, than use all named exports and no default. But if you have one thing that stands out as the main export and everything else sort of enhances or revolves around that main export, then it should be the default and everything else should be named exports. If you find yourself in a situation where you want to have several named exports but also two or more default exports, then your module is probably doing too much and should be broken down into smaller modules. We'll cover techniques for breaking big modules into smaller modules in the next lesson.

Quick check 20.2

Q1:

In the following snippet there are two functions. Alter the code to make the function `ajax` the default export and export `setAjaxDefaults` by name.

```
function ajax(url) {  
  // perform ajax  
}  
  
function setAjaxDefaults(options) {  
  // store options  
}
```

QC 20.2 answer

A1:

```
export default function ajax(url) {  
  // perform ajax  
}  
  
export function setAjaxDefaults(options) {  
  // store options  
}
```

20.3. WHEN DOES A JAVASCRIPT FILE BECOME A MODULE?

If a module is just a file, what determines if a given JavaScript file is a module or not? The WHATWG (Web Hypertext Application Technology Working Group), the organization responsible for specifications for the web-specific JavaScript environment, has proposed a new script type module. Basically, when using a script tag, instead of using `type="javascript"` you would specify `type="module"` and the JavaScript loader used by browsers would know to execute this file as a module:

```
<script type="module" src="./example.js">
```

This wouldn't work in Node.js, though, since it doesn't use script tags or HTML. So there's another proposal for inferring that a file is a module by whether or not it exports a value. To specify that a file is a module when it doesn't export a value, you would use a named export without actually exporting anything:

```
export {} 1

// rest of the module
```

- **1 An empty export to signify this file is a module**

Don't confuse this with exporting an object. This is the named export syntax where you would normally list your exports by name between the braces. Without naming anything, nothing gets exported, but with the proposal, it would identify the file as a module. If this proposal fails, the Node.js team plans to use the alternate file extension `.mjs` to specify the file is a module.

Further reading

<https://blog.whatwg.org/js-modules>

<https://github.com/bmeck/UnambiguousJavaScriptGrammar>



SUMMARY

In this lesson, you learned to teach you how to create modules.

- Exporting a declaration creates a named export.
- Named exports can also be listed in braces.
- Named exports listed in braces can be exported using alternate names.
- There can be several named exports, but only one default.
- Export statements must be declared at the top (root) level.
- The keyword `this` is undefined at the root level.
- Modules default to being in strict mode.

Let's see if you got this:

Q20.1

Create a module called `luck_number.js`. Give it an internal variable (not exported) called `luckyNumber`. Export a function called `guessLuckyNumber` as the default export that accepts a parameter called `guess`, checks to see if the `guess` is the same as `luckyNumber`, and returns `true` or `false` indicating if they made a correct guess.

Lesson 21. Using modules

After reading lesson 21, you will

- Understand how to specify the location of a module you intend to use
- Understand all the various ways of importing values from modules
- Understand how to import modules for side effects
- Understand the order of execution of code when importing modules
- Be able to break apart large modules into smaller ones

Modules are a great way to separate your logic into cohesive units and share logic across files without the cumbersome use of global variables. They also allow you to import only what is needed, keeping the cognitive load down and making maintenance easier. In the previous lesson you learned what a module is and the basics of how to create a module and export values. In this lesson we'll look at using other modules, the various ways of importing values, and how to break apart and organize your code using modules.

Consider this

Imagine that you're writing a web application and you need to use a few third-party open source libraries. The problem is that two of those libraries both expose themselves using the same global variable. How would you make the two libraries work together?

21.1. SPECIFYING A MODULE'S LOCATION

You import code from other module files using the `import` statement, which is made up of two key pieces, the *what* and the *where*. When you use the `import` statement, you must specify *what* (variables/values) you're importing, and from *where* (the module file) you're importing them. This is in contrast to including multiple files by using several `<script>` tags and communicating or sharing values across files by using global variables. The basic syntax is `import X from Y` where `X` specifies what you're importing and `Y` specifies where the module is. A simple import statement may look like this:

```
import myFormatFunction from './my_format.js'
```

Then when you specify where or from what module you're importing, you have to use a string literal value. The following is not valid:

```
const myModule = './my_format.js'
import myFormatFunction from myModule    1
```

- **1 Invalid because `myModule` is a variable, not a string literal**

You can't use a variable to define where (from) the module is, even if that variable points to a string. You must use a string literal. This again is because all imports and exports in JavaScript are designed to be statically analyzable. All of your imports will be executed before any other code in the current file. JavaScript will scan your file, figure out all the imports, execute those files first, and then run your current file with the correct

values imported. This means you couldn't import based on a variable because that variable wouldn't be defined yet!

Other than using a string to determine where a module is, there aren't any other rules set forth by JavaScript. There will be what are called *loaders* for each JavaScript environment (mainly the browser and Node.js), and those loaders will define what the string actually looks like. Loaders for the web and Node.js are still being figured out. Today, most people using ES6 modules are doing so using something such as Browserify or Webpack. Both of these tools treat a file path such as `./file` or `./file.js` as relative to the current file:

```
import myVal from './src/file'           1
import myVal from './src/file.js'        1
```

- **1 Both are equivalent, specifying the relative path to the file.**

The file extension is optional and most people omit it. Without a file extension, the path could also be a directory containing an `index.js`.

```
import myVal from './src/file'           1
import myVal from './src/file.js'        2
```

- **1 Matches both `./src/file.js` and `./src/file/index.js`**
- **2 Matches only `./src/file.js`**

Specifying a name without a path such as `jquery` would indicate that this is an installed module that should be looked for in the `node_modules` directory.

Now that you know how to specify where the module is, let's take a look at how to specify *what* you want to import from it.

Quick check 21.1

Q1:

Where would the presumed loader most likely look for the module files for the following imports:

```
import A from './jquery'  
import B from 'lodash'  
import C from './my/file'  
import D from 'my/file'
```

QC 21.1 answer

A1:

1. From the file ./jquery.js or ./jquery/index.js relative to current file.
2. From the main field specified in node_modules/lodash/package.json.
3. From the file ./my/file.js or ./my/file/index.js relative to current file.
4. From either ./file.js or ./file/index.js relative to the main field specified in node_modules/my/package.json.

21.2. IMPORTING VALUES FROM MODULES

In the previous lesson you created a module for formatting currency strings. It had a single default export of a function to achieve this like so:

```
export default function currency(num) {  
  // ... return a formatted currency string  
}
```

Let's say you put this module in the location `./utils/format/currency.js` and want to import the currency function to format some currencies in the shopping cart system you're building. You could do so using the following `import` statement:

```
import formatCurrency from './utils/format/currency'  
1  
  
function price(num) {  
  return formatCurrency(num)  
}
```

- **1 You're importing the default value.**

Notice how the name of the function was `currency` but you imported using the name `formatCurrency`. When you say `import <name>` you're deciding what name to use, just like when you use `var <name>`, `const <name>`, or `let <name>`, only instead of assigning a value using the equals sign, you're importing a value from another file, a module. You can use any name you want when importing the default value from a module, as you can see in the following silly example:

```
import makeNumberFormattedLikeMoney from  
  './utils/format/currency'  
  
function price(num) {
```

```
    return makeNumberFormattedLikeMoney(num)
}
```

The previous two examples would behave exactly the same, assuming there were no changes to the `./utils/format/currency` module they were importing from. Remember the analogy from the previous lesson: If you were to think of a module as a function, the default export would be analogous to the `return` value from the function. If we continue to use this analogy, importing the default value would be like assigning the `return` value of a function to a variable:

```
function getValue() {
  const value = Math.random()
  return value
}

const value = getValue()
const whatchamacallit = getValue()
```

Notice how the `getValue` function returned a variable called `value`, and it didn't matter if you assigned that to a matching named variable, or a variable with a completely different name such as `whatchamacallit`. This is because the function only returns a single value, and you're merely capturing that value and deciding on a name to store it. This is the same when you import a default value from a module. The module can only export a single default value, and it only exports the value, so when you import it, you specify any name you want for storing that value.

Now as you learned in the previous lesson, in addition to a single default export, a module can also have one or more named exports. As the name implies, with these the name does matter. One of the syntaxes you learned in the previous lesson for named exports was the following:


```
function currency(num) {
  // ... return a formatted currency string
}

function number(num) {
  // ... return a number formatted with commas
}

export { currency, number }
```

Conveniently, the syntax to import these is similar:

```
import { currency, number } from './utils/format'

function details(product) {
  return `
    price: ${currency(product.price)}
    ${number(product.quantityAvailable)} in stock ready to
  ship.
`
}
```

Here the names `currency` and `number` must match the names they were exported as. But you can specify a different name to assign them to:

```
import { number as formatter } from './utils/format' 1
```

- **1 Specified to import `number` but assign it to the name `formatter`**

This is handy if you're importing named values from multiple modules that export using the same name. Say, for example, that you're importing a function called `fold` from a `functional tools` module, but also importing a function called `fold` from an `origami` module. You could use `as` to map one or both of the imports to a different name to avoid naming conflicts:

```
import { fold } from './origami'
import { fold as reduce } from './functional_tools'
1
```

- **1 `fold` is imported under the name `reduce` to avoid conflict**

with the other imported value.

If you want to import all of the named exports from a module, you can do so using the asterisk like so:

```
import * as format from './utils/format'      1

format.currency(1)
format.number(3000)
```

- **1 A new object named `format` is created and assigned all the values from the module.**

This will create a new object with properties correlating to all the named exports from the module. This is handy if you need to import all values a module exports, possibly for introspection or testing, but normally you should only import what you're going to use. Even if you happen to be using everything that a module exports, it doesn't necessarily mean that you'll continue to use everything from that module as it grows.

Imagine, for example, that the `format` module only exported the format functions `currency` and `date`, and you need both for your product module, so you import them all using the asterisk. But later as you're building your application, you continue to add new format functions to your format module. You don't need these new functions in your product module, but since you were importing using the asterisk, you're going to continue to get them all, not just the ones you're using. Some situations may require you to import everything, but as a general rule, you should be explicit in what you import by specifying each value by name.

When importing all values using the asterisk, this doesn't include the default export, just the named exports. You can combine importing the default and named exports

from a module by separating them with a comma:

```
import App, * as parts from './app'  
1  
import autoFormat, { number as numberFormat } from  
'./utils/format' 2
```

- **1 The default value gets named App and all the named exports are set as properties of a newly created object called parts.**
- **2 The default value gets named autoFormat and the value exported by the name number gets named numberFormat.**

Once you import a value from a module, it doesn't create a binding like when declaring a variable. In the next section we'll look at how that works.

Quick check 21.2

Q1:

In the following `import` statement, which is the default import and which is the named import?

```
import lodash, { toPairs } from './vendor/lodash'
```

QC 21.2 answer

A1:

`lodash` is the default import and `toPairs` is the named import.

21.3. HOW IMPORTED VALUES ARE BOUND

Both default and named imports create read-only values, meaning you can't reassign a value once it's imported:

```
import ajax from './ajax'

ajax = 1 1
```

- **1 Error: ajax is read only**

But named imports, unlike default imports, are bound directly to the variables that were exported. This means that if the variable changes in the file (module) that exported it, it will also change in the file that imported it.

Let's imagine a module that exports a variable named `title` that has an initial value but also exports a function called `setTitle` that allows you to change the title like so:

```
export let title = 'Java'
export function setTitle(newTitle) {
  title = newTitle
}
```

If you were to import both, you couldn't directly change the value of `title` via assignment, but you could indirectly change the value of `title` by calling `setTitle`:

```
import { title, setTitle } from './title_master'

console.log(title) 1
setTitle('Script')
console.log(title) 2
```

- **1 “Java”**
- **2 “Script”**

This is very different from how you're used to retrieving values in JavaScript. Normally when you retrieve a value in JavaScript—either from a function call or destructuring or some other expression—and you assign it to a variable, you're retrieving the value, and creating a new binding pointing to that value. But when you import values from modules, you're importing not just the value but also the binding. This is why the module can internally change the value and your imported variable will reflect the change.

Once the value changes, it not only changes in the current file and the file that exported the value, but it also changes in all files that imported that value. On top of that, there's no notification that the value changed. There's no event broadcasting the change. It changes silently, so take care when changing values that you've exported.

In the next section, you'll learn how and why you would import a module without importing any values at all.

Quick check 21.3

Q1:

There are five bindings in the following snippet. Which ones can be reassigned values in the shown context?

```
import a, { b } from './some/module'
const c = 1
var d = 1
let e = 1
```

QC 21.3 answer

A1:

Only d and e.

21.4. IMPORTING SIDE EFFECTS

Sometimes you just want to import a module for side effects, meaning you want the code in the module to execute, but you don't need a reference to use any values from that module. An example of this would be a module that contains the code that sets up Google Analytics. You wouldn't need any values from such a module; you just need to execute so it can set itself up.

You can import a module for side effects like so:

```
import './google_analytics'
```

It's just like any other import; you omit any default or named values and you omit the keyword `from` as well. When you import a file for side effects, all of the code from the module you import will execute before any of the code in the file from which you imported it. This is regardless of where the import happens:

```
setup()  
  
import './my_script'
```

In the previous example, all of the code in the module `my_script` is executed before the `setup()` function is executed, even though it's imported afterward.

In the next section we'll take a look at organizing and grouping smaller modules into bigger modules.

Quick check 21.4

Q1:

Assume the module `log_b` contains the statement

`console.log(' B ')`. After running the following code, what will be the order of the output?

```
console.log('A')
import './log_b'
console.log('C')
```


QC 21.4 answer

A1:

B, A, C

21.5. BREAKING APART AND ORGANIZING MODULES

Sometimes a module will grow too big, and it may make sense to break it apart into smaller modules. But you may have a large code base that's already using this module all over the place. You want to refactor this module into smaller, more focused modules, but you don't want to have to refactor your entire application because of it. Let's explore how you can take a module that's already in use and break it down into smaller chunks without having any effect on the rest of your application.

Let's assume you have a format module. It starts off small with just a few format functions, but as the application grows, you continue to need new formatters for different needs. Some formatters share logic, while others need their own helper functions. Keeping all of this in a single module has grown too complex. For brevity let's assume four formatters, as shown in the following listing; in a real application, it could be many more.

Listing 21.1. `src/format.js`

```
function formatTime(date) {  
    // format time string from date object  
}  
  
function formatDate(date) {  
    // format date string from date object  
}  
  
function formatNumber(num) {  
    // format number string from number  
}  
  
function formatCurrency(num) {  
    // format currency string from number  
}
```

Now let's assume many modules are using these, such as the following product module.

Listing 21.2. `src/product.js`

```
import { formatCurrency, formatDate } from './format'
```

This product module is just one of many that are using formatters. You want to refactor your format module in a way that won't break this one or any others.

Break the module into two separate modules, one for numbers and one for dates, and group them in a format folder.

Here's the date format module.

Listing 21.3. `src/format/date.js`

```
function formatTime(date) {  
  // format time string from date object  
}  
  
function formatDate(date) {  
  // format date string from date object  
}
```

And here's the number format module.

Listing 21.4. `src/format/number.js`

```
function formatNumber(num) {  
  // format number string from number  
}  
  
function formatCurrency(num) {  
  // format currency string from number  
}
```

You've nicely broken your large format module into smaller, more focused, modules. But to make use of these, you would have to refactor all of the other modules, like the product module, that are importing values. You want to prevent that. Create another index module inside the format module that imports the values

from the more focused modules and exports them, as shown in the next listing.

Listing 21.5. `src/format/index.js`

```
import { formatDate, formatTime } from './date'
import { formatNumber, formatCurrency } from './number'

export { formatDate, formatTime, formatNumber, formatCurrency }
```

This is great because now when other modules try to import from `./src/format`, it will actually import from `./src/format/index.js` if `./src/format.js` isn't found. This means you no longer need to refactor any of the other modules. This is a great argument for omitting the file extension when specifying module paths in your imports, because if you had specified the file extension, this refactoring would have been much more painful.

This type of organization is so common that there's actually a syntax directly for it. The `src/format/index.js` module could be written like so.

Listing 21.6. `src/format/index.js`

```
export { formatDate, formatTime } from './date'
export { formatNumber, formatCurrency } from './number'
```

If your only use for a value you're importing is to turn around and export it, you can skip a step and export it directly from that module! Now, the format module is always supposed to export all the values from all the more focused formatters, right? Well instead of having to list all the names and then come back and add names for any new formatters you add in the future, you can export them all like so.

Listing 21.7. `src/format/index.js`

```
export * from './date'
export * from './number'
```

Cool! With this simple facade type module, you've successfully and quite elegantly broken your large module into much smaller and more focused modules, and you did it seamlessly in a way that's opaque to the rest of the application!

Quick check 21.5

Q1:

Assume you are going to add another module at `./src/format/word` and update the index file to export all of the word formatters.

QC 21.5 answer

A1:

```
export * from './date'
export * from './number'
export * from './word'
```

SUMMARY

In this lesson, you learned how to use and organize modules.

- Default imports can be set using any name.
- Named imports are listed in braces, similar to how they're exported.
- Named imports must specify the correct name.
- Named imports can use alternate names via `as` after specifying the correct name.
- You can import all named values using an `*`.
- Named imports are direct bindings (not just references) to the exported variables.
- Default imports aren't direct bindings but are still read-only.
- Values can be exported directly from other modules.

Let's see if you got this:

Q21.1

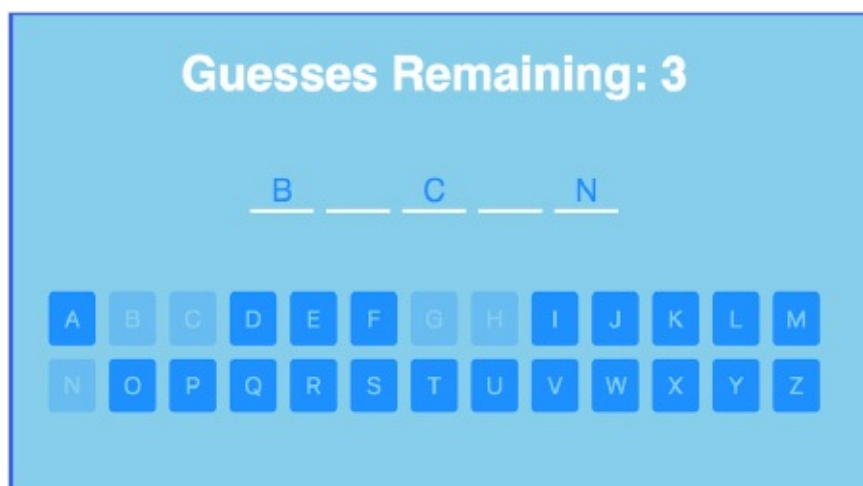
Make a module that imports the `luck_numbery.js` from the previous lesson and attempts to guess the lucky number and log how many attempts it made before guessing the correct number.

Lesson 22. Capstone: Hangman game

In this capstone you're going to build a hangman game. The game will incorporate a status message, letter slots for the word, and buttons for guessing letters ([figure 22.1](#)).

You'll start your project using the start folder included in the code accompanying this book. If at anytime you get stuck, you can also check out the final folder with the completed game. The start folder is a project already set up to use Babel and Browserify (see [lessons 1–3](#)); you just need to run `npm install` to get set up. If you haven't read [lessons 1–3](#), you should before doing this capstone. There's also an included `index.html` file: this is where the game will run. It already includes all the HTML and CSS it needs; you just need to open it in a browser once you bundle your JavaScript files. The `src` folder is where you'll put all your JavaScript files, and the `dest` folder is where the bundled JavaScript file will go after you run `npm run build`.

Figure 22.1. A hangman game



22.1. PLANNING

You're going to break your game into several modules, so it makes sense to start by identifying what modules you're going to break the game into. Before you can start the game, you'll need a random word. It makes sense to create a module for generating random words. Second, you'll need to keep track of the status of your game—whether the game was won, lost, and so on. So you'll need a status module as well. There will be three parts of the UI that need to be displayed to the user. First is a representation of the game's status; call this *status-display*. You'll also need to show the letter slots for the word the player is guessing; call this *letter-slots*. The third UI element you'll need will be buttons for each letter of the alphabet so the player can make their guesses. Call this module *keyboard*. Finally, you'll need the glue that puts all these modules together to create the actual game. There won't be much glue so you'll just do this in the *index*.

22.2. THE WORDS MODULE

Start with a simple function that just returns an array of words.

Listing 22.1. src/words.js

```
function getWords(cb) { 1  
  cb(['bacon', 'teacher', 'automobile']) 2  
}
```

- **1 The function accepts a callback as a parameter.**
- **2 Invoke the callback with your array of words.**

Instead of returning an array of words, you use a callback to pass back an array of words. In its current state, that might not make sense, but this will allow rewriting the `getWords` function later to use an AJAX request to get words from an API or some external resource.

Now that you have your words, you need a function that will return a random word, as shown in the following listing.

Listing 22.2. src/words.js

```
export default function getRandomWord(cb) {  
  getWords(words => {  
1  
    const randomWord = words[Math.floor(Math.random() *  
words.length)] 2  
    cb(randomWord.toUpperCase())  
3  
  })  
}
```

- **1 Pass a cb to getWords.**
- **2 Get a random word from the words array.**
- **3 Invoke the callback given to getRandomWord with your random word.**

This is it for your words module. You only exported the

`getRandomWord` function from the module because the rest of the game won't need an array of words, just a single random word at a time. Also since you only exported a single function, you set it as the default export. Next you'll build the status module.

22.3. THE STATUS MODULE

There are four statuses that you'll use for the game: how many guesses are remaining, whether or not the player has won, whether or not the player has lost, and whether or not the game is still in play (when they still have guesses remaining and they haven't won or lost). The random word along with the player's guesses will be needed to determine any of these statuses, so you'll export a function for each one that accepts the current word and guesses as parameters.

Listing 22.3. src/status.js

```
const MAX_INCORRECT_GUESSES = 5
1

export function guessesRemaining(word, guesses) {
  const incorrectGuesses = guesses.filter(char =>
!word.includes(char)) 2
  return MAX_INCORRECT_GUESSES - incorrectGuesses.length
}

export function isGameWon(word, guesses) {
  return !word.split('').find(letter =>
!guesses.includes(letter)) 3
}

export function isGameOver(word, guesses) {
  return !guessesRemaining(word, guesses) && !isGameWon(word,
guesses) 4
}

export function isStillPlaying(word, guesses) {
  return guessesRemaining(word, guesses) &&
    !isGameOver(word, guesses) &&
    !isGameWon(word, guesses)
5
}
```

- **1 You don't need to export this because nothing else should need it.**
- **2 Find all the letters that have been guessed and aren't in the word.**
- **3 Determine if all the letters in the word have been**

guessed.

- **4 If the game hasn't been won and there are no guesses yet it's game over.**
- **5 As long as there are guesses and the game hasn't been won or lost, the player is still playing.**

Notice how you didn't export the `MAX_INCORRECT_GUESSES` constant. This is because nothing else will use it, and you should only export what's needed by other modules and nothing more to keep the API surface as small as possible, which will make future changes and debugging easier. The other four functions will all be used by other modules to make the game work, so you export all of them. That doesn't mean you should always export every function, though. If one of these functions used a helper function to determine its value, you wouldn't export such a helper function because it wouldn't need to be used elsewhere.

You may be asking yourself, why pass around the `word` and `guesses` as function arguments? Why not just import them? You could do that and it would work. But doing so would tightly couple all the individual modules to the main game logic (where the `word` and `guesses` get stored), which would make them much harder to isolate and test. In this small game you won't be adding any tests, but it's still a good practice.

Next we'll focus on the three UI elements: the status display, the letter slots, and the keyboard.

22.4. THE GAME'S INTERFACE MODULES

As we said before, you have three pieces of UI: the status display that will show how many guesses are left or if the game is over or won, the letter slots, and the keyboard. You could make a single UI module that exports each one; that wouldn't be wrong, but I would prefer to put each into its own module. They don't share any logic or have anything else that would suggest they belong together other than all being parts of the UI, so I think this makes the most sense. I would prefer a few simple modules over a single more complicated one, and if you later decided to add more pieces of UI, it would further suggest that each one should be its own module.

Each UI module will export a single function that returns the HTML (as a string) representing that part of the game's interface. All the UI modules will require the player's guesses, while the status display and letter slots will also require the current random word. So they'll also accept them as parameters similar to the status module.

OK, so you have a simple API that each UI module will adhere to. It will export a single function (you'll make it a default export) and that function will accept the data it needs and return a string of HTML. Start with the status display module.

Listing 22.4. /src/status_display.js

```
import * as status from './status'
1

function getMessage(word, guesses) {
  if (status.isGameWon(word, guesses)) {
2
    return 'YOU WIN!'
  } else if (status.isGameOver(word, guesses)) {
2
```

```

    return 'GAME OVER'
  } else {
    return `Guesses Remaining: ${status.guessesRemaining(word,
guesses)}` 2
  }
}

export default function statusDisplay(word, guesses) {
  return `<div>${getMessage(word, guesses)}</div>`
}

```

- **1 Import all values into a single status object.**
- **2 Invoke the imported functions from the generated status object.**

The status display will need most of the status functions, so instead of individually importing them, import everything from the status module and group it all together into a generated status object. You then can invoke any of the functions directly from the created status object.

Other than that, this module is pretty simple. It just generates a message determined by the status of the game. Next you'll build the letter slots module.

Listing 22.5. /src/letter_slots.js

```

function letterSlot(letter, guesses) {
  if (guesses.includes(letter)) {
    return `<span>${letter}</span>`
  } else {
    return `<span>&nbsp;</span>`
  }
}

export default function letterSlots(word, guesses) {
  const slots = word.split('').map(letter => letterSlot(letter,
guesses))

  return `<div>${ slots.join('') }</div>`
}

```

This module is simple as well: you generate a bunch of spans corresponding to each letter in the word. The span will either be empty or reveal the letter, depending on

whether the player has guessed that letter or not. Again you export our one default function as planned. Now do the final bit of UI, the keyboard module, as shown in the next listing.

Listing 22.6. /src/keyboard.js

```
const alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'.split('')
1

const firstRow = alphabet.slice(0, 13)
2
const secondRow = alphabet.slice(13)
2

function key(letter, guesses) {
  if (guesses.includes(letter)) {
    return `<span>${letter}</span>`
3
  } else {
    return `<button data-char=${letter}>${letter}</button>`
4
  }
}

export default function keyboard(guesses) {
  return `
    <div>
      <div>${ firstRow.map(char => key(char, guesses)).join('')}
    </div> 5
      <div>${ secondRow.map(char => key(char,
guesses)).join('')} </div> 5
    </div>
  `
}
```

- **1 A quick way to get an array of all the letters in the alphabet**
- **2 You want the first 13 letters in the first row and the last 13 in the last row.**
- **3 If the letter was guessed, you don't want to give them the option to guess it again, so use a span.**
- **4 If the letter wasn't guessed, you used a button to allow them to select it.**
- **5 Map each letter into a button or span depending on whether the letter was guessed.**

This module is also pretty simple: it generates a list of all

the letters in the alphabet, making the letters that haven't been guessed yet as buttons and the letters that have been guessed as spans.

That's all the UI you'll need. In the next section you'll put it all together and create a working game.

22.5. THE INDEX

The index is the entry point of your application. Here you'll orchestrate all the individual modules, creating a functional game. First import everything you're going to need, as shown in the next listing.

Listing 22.7. /src/index.js

```
import getRandomWord from './words'
import { isStillPlaying } from './status'
import letterSlots from './letter_slots'
import keyboard from './keyboard'
import statusDisplay from './status_display'
```

You import the default function from the words module as well as all the UI modules. But you only need the `isStillPlaying` function from the status module to determine if you should still interact with the player.

In the next listing you just need to make a function that renders the actual game.

Listing 22.8. /src/index.js

```
function drawGame(word, guesses) {
  document.querySelector('#status-display').innerHTML =
    statusDisplay(word, guesses)
  document.querySelector('#letter-slots').innerHTML =
    letterSlots(word, guesses)
  document.querySelector('#keyboard').innerHTML =
    keyboard(guesses)
}
```

Here you invoke each of the UI functions you imported and use `innerHTML[1]` to insert them into your web page where needed. You don't need any other logic for the interface because each UI module handles that itself using the word and guesses. So the only other things you need to do are get a random word, listen for button clicks, and add each guess to a list of guesses, as the following listing shows.

See <https://developer.mozilla.org/en-US/docs/Web/API/Element/innerHTML>

Listing 22.9. /src/index.js

```

getRandomWord(word => {
1
  const guesses = []
2

  document.addEventListener('click', event => {
3
    if (isStillPlaying(word, guesses) && event.target.tagName
=== 'BUTTON') {4
      guesses.push(event.target.dataset.char)
5
      drawGame(word, guesses)
6
    }
  })

  drawGame(word, guesses)
7
})

```

- **1 First you need to get a random word.**
- **2 You also need a place to store the player's guesses.**
- **3 Use event delegation to listen to all click events.**
- **4 If the game is still in play and the player clicked a button...**
- **5 ...add the letter the player guessed to your array.**
- **6 Redraw the game.**
- **7 Draw the initial game UI.**

After you get a random word, you start listening for button clicks using event delegation.^[2] Every time the player makes a guess, the entire UI is destroyed and recreated. This could be optimized, but for this small game it's OK and makes the game much simpler. Event delegation allows you to add the click event listener once without needing to reregister it every time the UI is rebuilt.

See: <https://davidwalsh.name/event-delegate>

You now have a working game. You can build the game using `npm run build` in your terminal and then open `index.html` in your browser!

SUMMARY

In this capstone you created a hangman game. You started by making a word module for generating the random word used for your game. You then worked on the status and interface components of the game. You finally put all the pieces together in the index file. You're currently only using three random words, which doesn't make it very hard to guess. Feel free to update the game with a longer list of words or use an API. You can also take this game further by adding a Play Again button to restart the game once it's over.

Unit 5. Iterables

In JavaScript, `Strings` and `Arrays` have always had a couple of things in common. They both contain an indefinite amount of *stuff*—characters in the case of strings and any datatype in the case of arrays. They both also have a `length` property indicating how many items they have. But there was never a common protocol that described how these things worked. Starting in ES2015, there are two new protocols that describe these JavaScript behaviors, known as the *iterable* and *iterator* protocols.

`Strings` and `Arrays` are now known as *iterables*. This means they adhere to the new iterable protocol and can be predictably interacted with in common ways, including being used with the new `for . . of` statement and the new *spread* operator. There are also two new iterables: `Maps` and `Sets`. Additionally, you can define your own iterables or even customize the behavior of the built-in ones. And because they all follow the iterable protocol, they will always behave in predictable ways.

We'll start the unit by looking at the iterable protocol itself: how it works, how to create your own iterables, and how to use iterables with the `for . . of` statement and spread operator. You'll then learn about the new built-in iterable types, `Sets` and `Maps`. Finally, you'll wrap up the unit by building a Blackjack game that make uses of `Maps` and `Sets`, as well as using `for . . of` and `spread`.

Lesson 23. Iterables

After reading [lesson 23](#), you will

- Know what an iterable is and how to use it
- Know how to use the spread operator on iterables to ungroup an object's items
- Know how to use iterables in a `for . . in` statement to loop over an object's values
- Know how to create your own iterables
- Know how to customize the behavior of built-in iterables

JavaScript in ES2015 has introduced a couple of new protocols: the iterable protocol and the iterator protocol. Together these protocols describe the behavior and mechanics of objects that can be *iterated*—that is to say, objects that produce a series of values and can have their values looped over. When you think of objects that can be iterated, `Array` probably springs to mind, but you can also iterate `Strings`, the `arguments` object, `NodeLists`, and as we'll see in the upcoming lessons, `Sets` and `Maps`. On top of having all these built-in iterables, you can, as you'll discover in this lesson, build your own!

Consider this

The following code logs all of the indices of an array. But what if you wanted to log all of the values?

```
for (const i in array) {  
  console.log(i);  
}
```

23.1. ITERABLES—WHAT ARE THEY?

An iterable is any object that follows the iterable protocol that's new in JavaScript, introduced in ES2015.

Common built-in iterables are `Strings` and `Arrays` but so are `Sets` and `Maps`. Because all of the objects follow a common protocol, it means they all behave in similar ways. You can also use this protocol to create your own iterables or customize the behavior of default ones!

As I mentioned, not all iterables are new. JavaScript has always had strings and arrays. But the protocol that sets a contract for how strings, arrays, and other iterators behave is new. Strings and arrays have been updated to use this new protocol, and other objects like `Sets` and `Maps` are new objects that also make use of the protocol.

Along with the new iterable protocol, we get some new ways to interact with objects that make use of it, specifically the `for...of` statement and the spread operator.

23.2. THE FOR..OF STATEMENT

How many times have you written code that looks like this?

```
for (var i = 0; i < myArray; i++) {  
  var item = myArray[i]  
  // do something with item  
}
```

With the `for . . of` you can now achieve the same thing with this:

```
for (const item of myArray) {  
  // do something with item  
}
```

JavaScript has long had the `for . . in` statement that allows you to enumerate an object's keys (property names), but now with the `for . . of` statement you can iterate an iterable's values, as shown in the next listing.

Listing 23.1. Comparing `for . . in` and `for . . of`

```
const obj = { series: "Get Programming", publisher: "Manning"  
};  
const arr = [ "Get Programming", "Manning" ];  
  
for (const name in obj) {  
  console.log(name);           1  
}  
  
for (const name of arr) {  
  console.log(name);           2  
}
```

- **1 series, publisher**
- **2 Get Programming, Manning**

This is similar to `Array.prototype.forEach`, but it has a few benefits. It can be used with any iterable, not just arrays. It is an imperative operation and thus can be

optimized to perform better than the higher-order `forEach` method. Additionally, it can be broken out of early using `break`.

One final benefit of using `for...of` is that you can't `yield` from a non-generator function, even if that function is inside a generator:

```
function* yieldAll(...values) {
  values.forEach(val => {
    yield val
  })
}
```

- **`1 SyntaxError: unexpected identifier`**

You get this error because the `yield` is actually inside of the arrow function that's the callback for `forEach`. The `yield` doesn't bubble up to the nearest generator function in the stack. If it's used inside a non-generator function, it's a syntax error. However, you can get around this by using `for...of` like so:

```
function* yieldAll(...values) {
  for (const val of values) {
    yield val
  }
}
```

The `for...of` statement isn't a game changer, but it is yet another tool in your JavaScript Next tool belt. So far we've been looking at ways to process existing iterables; in the next section you'll create your own iterables.

Quick check 23.1

Q1:

How many times will the following `for...of` loop run?


```
for (const x of "ABC") {  
  console.log('running')  
}
```

QC 23.1 answer

A1:

3

23.3. SPREAD

One thing all iterables in JavaScript share is their ability to be used with the *spread operator*. The spread operator allows you to treat the passing of a single iterable as if you passed all of its items individually. Let's look at what that means. Imagine you're running an online marketplace. For any item, there are a series of vendors who offer that item, all at different prices. You want to show the user the lowest price available. You can easily get an array of all the prices, but once you have an array of prices, how do you find the lowest price? You want to give this array of prices to `Math.min` but that requires you pass in all the prices individually, not as an array. Before the spread operator, you would likely use something like this:

```
const prices = // get array of prices
const lowestPrice = Math.min.apply(null, prices);
```

However this can be greatly simplified with the use of the spread operator:

```
const prices = // get array of prices
const lowestPrice = Math.min(...prices);
```

Instead of passing the prices array to `Math.min` as a single parameter, it's passing all of the values of the array as separate parameters. This doesn't just work on arrays, but on any iterable, even strings. In the case of a string, though, it would pass each individual character as a separate parameter.

You might have noticed that spread uses the exact same syntax as rest. This is by design, as spread is the exact

opposite of rest. When writing a function you expect to be given a series of values as parameters, you can group all of the values you're given into a single array using rest, as shown in the next listing.

Listing 23.2. Grouping parameters into an array using rest

```
function findDuplicates(...values) {  
  // gather all values as an array  
  // return an array of the duplicates  
}  
  
findDuplicates("a", "b", "a", "c", "c");           1
```

- 1 ["a", "c"]

The function `findDuplicates` takes any number of parameters and returns the ones that are duplicates. Internally it groups all those values into an array using spread.

On the flip side, if you already have an array, you can't just pass it to the `findDuplicates` function because it will only have one array and will be looking for duplicates of the array. However you can *ungroup* the array and send its contents as individual arguments, as shown in the next listing:

Listing 23.3. Ungrouping parameters using spread

```
const letters = ["a", "b", "a", "c", "c"]  
  
findDuplicates(...letters);           1
```

- 1 ["a", "c"]

Again, this doesn't need to be an array to use spread; it can be any iterable such as a string (or Map or Set, which you will learn about later in this unit), as shown in the following listing and [figure 23.1](#).

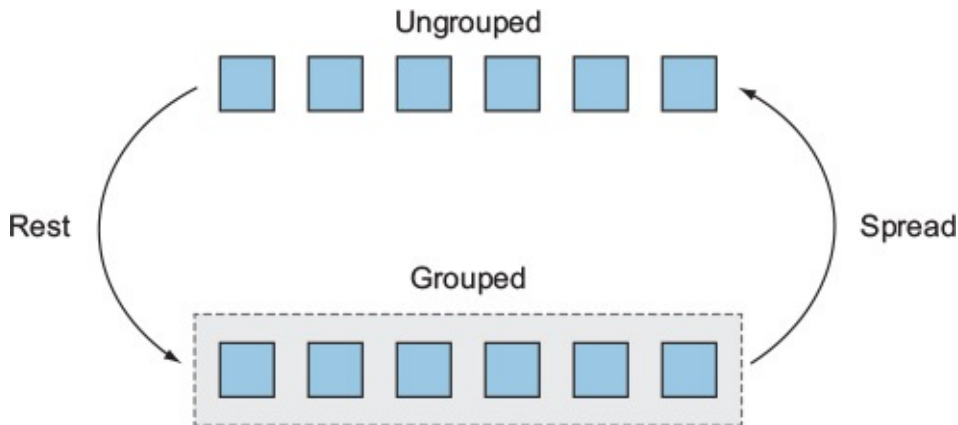
Listing 23.4. Ungrouping parameters using spread

```
findDuplicates(..."abacc");
```

1

- 1 ["a", "c"]

Figure 23.1. The rest-spread relationship



Spread isn't limited to function arguments, either. It can also be used to spread an iterable across an array literal:

```
const surname = "Isaacks"
const letters = [ ...surname ]
console.log(letters)
```

1

- 1 ["I", "s", "a", "a", "c", "k", "s"]

It doesn't have to be the only item, either, and unlike rest, it doesn't even need to be the last item:

```
const easyAs = [ ...'123', 'ABC' ]
console.log(easyAs)
```

1

- 1 ["1", "2", "3", "ABC"]

You can even combine multiple spread operators together:

```
const vowels = ['A', 'E', 'I', 'O', 'U']
const consonants = 'BCDFGHJKLMNPQRSTVWXYZ'

const alphabet = [ ...vowels, ...consonants ].sort()

console.log(alphabet.length)
```

1

- 1 26

Notice how we're spreading an array and spreading a string. It doesn't matter what type they are, just that they're both iterables. The result is a new array with all 26 letters.

23.3.1. Using spread as an immutable push

This technique can be used as an immutable form of `Array.prototype.push`. Sometimes you may want to add something to an array, but instead of modifying the original array, get a copy with the value added. This is a common technique in libraries such as `Redux.js`, where you must take the existing state and some data and derive the next state without modifying the existing state. If the existing state is an array, and you want to add an item to the array, using `push` would modify the existing state (the current array), which could cause bugs. But you can use spread to copy an existing array into a new array with the new item:

```
function addItemToCart(item) {  
  return [ ...cart, item ]  
}
```

This would create a new array and leave the previous array untouched. In order to achieve this using `push`, you would have to first create a copy of the array, then `push` to the copy, then return the copy:

```
function addItemToCart(item) {  
  const newCart = cart.slice(0)  
  newCart.push(item)  
  return newCart  
}
```

Notice how much more concise the version using spread is. You can make an operation like this even more expressive with use of an arrow function like so:

```
const addItemToCart = item => [ ...cart, item ]
```

Here it looks as if you were merely describing what the function does. In fact, this description is actually the implementation as well.

You can also use this technique to make a shallow copy of an array. This is really useful if you need to perform some destructive actions on an array but don't want to alter the original array:

```
function processItems(items) {  
  copy = [ ...items ]  
  
  // do destructive things to copy without altering items  
}
```

In the capstone of [unit 1](#) you wrote some helper functions, `createTag` and `interlace`, to make creating tagged template functions much easier. You used them both to create a tagged template function called `htmlSafe`. The original code is repeated in the next listing.

Listing 23.5. Original functions from unit 1 capstone

```
function createTag(func) { 1  
  return function() {  
    const strs = arguments[0];  
    const vals = [].slice.call(arguments, 1);  
    return func(strs, vals);  
  }  
}  
  
function interlace(strs, vals) { 2  
  vals = vals.slice(0); 3  
  return strs.reduce(function(all, str) {  
    return all + String(vals.shift()) + str;  
  });  
}  
  
const htmlSafe = createTag(function(strs, vals){ 4  
  return interlace(strs, vals.map(htmlEscape));  
});  
  
const greeting = htmlSafe`<h1>Hello, ${userInput}</h1>` 5
```

- **1 This whole function is just to abstract away how you're grouping all but the first parameter.**
- **2 Get a copy of the vals array.**
- **3 Put the string template together.**
- **4 HTML escape all the interpolated values.**
- **5 Example usage**

We now have some tools that could make this much easier. You reimplement the same functionality using spread and rest, as shown in the following listing.

Listing 23.6. Updated functions from unit 1 capstone

```
function interlace(strs, vals) {  
  vals = [ ...vals ];  
  1  
  return strs.reduce((all, str) => {  
    return all + String(vals.shift()) + str;  
  });  
}  
  
const htmlSafe = (strs, ...vals) => interlace(strs,  
vals.map(htmlEscape)); 2
```

- **1 Copying the array using spread instead of slice**
- **2 Using rest to gather values**

You were now able to condense this down quite a bit. In the `interlace` function, You used spread to make a copy of `vals` instead of `slice`. Also by using rest to gather the parameters, you didn't even need the `createTag` function anymore.

While you're at it, use default function parameters to remove the need to map all your values before passing them to your reducer:

```
function interlace(strs, vals, processor=String) {  
  vals = [ ...vals ];  
  return strs.reduce((all, str) => {  
    return all + processor(vals.shift()) + str;  
  });  
}
```

```
}  
  
const htmlSafe = (strs, ...vals) => interlace(strs, vals,  
htmlEscape);
```

Here you made `interlace` take a third parameter, called `processer`, that defaults to `String`, and used that for each value in the reducer. By doing that, calling the `interlace` function with only two parameters works just the same as before, but calling it with a third parameter (in this case `htmlEscape`) will instead run that function on each value in the reducer. This eliminates the need to iterate the entire list of values before reducing them.

Quick check 23.2

Q1:

What will the following lengths be in the logs?

```
const a = '123'  
const b = ['123']  
const c = [1, 2, 3]  
console.log([ ...a ].length)  
console.log([ ...b ].length)  
console.log([ ...c ].length)
```

QC 23.2 answer

A1:

1. 3
2. 1
3. 3

23.4. ITERATORS—LOOKING UNDER THE HOOD OF ITERABLES

Any object with an `@@iterator`^[1] property that follows the iterator protocol is an iterable. That is to say, an object becomes iterable when it has the property `@@iterator` pointing to another object that's an iterator.

1

The terminology `@@name` is a shorthand way to describe a property symbol of name. If an object is said to have the property `@@foo`, that is a shorthand way of saying it has the property `Symbol.foo`.

The purpose of an iterator is to produce a series of values. An iterator needs to be able to give each value in order, one at a time. It also needs to know when it has finished, so it can stop trying to produce values.

An iterator is an object that implements a `next` function. The `next` function must return an object with the two properties, `value` and `done`.

The `done` property indicates whether or not the iterator has completed iterating its properties. The spread operator and `for...of` will both continue to call `next` on an iterator until it specifies it has no more values by setting `done` to `true`. You never have to set `done` to `true`; but it's perfectly valid to make an infinite value iterator. But it's ill-advised to use spread or `for...of` on an infinite iterator, as it will never complete.

The `value` property returned from `next` indicates the next value that the iterator is producing.

Think about the example where you use spread to get an array of characters from a string:

- 1 ["I", "s", "a", "a", "c", "k", "s"]

It's the string's `@@iterator` that produces those values, not the string itself. It's because the string has this `@@iterator` property that the string is an iterable.

That is to say, to make an object iterable, it must have an `@@iterator` method that returns a new iterator object.

Next you'll create a function that you can use as an object's `@@iterator`. That means the function needs to return a new object with a `next` method, and the `next` method needs to return another object with `done` and `value` properties. Start with a simple iterator that just produces the first three prime numbers:

```
function primesIterator () {  
  const primes = [2, 3, 5]  
  return {  
    next() {  
      const value = primes.shift()  
      const done = !value  
      return {  
        value,  
        done  
      }  
    }  
  }  
}
```

Now create an iterable that uses this as its iterator:

```
const primesIterable = {  
  [Symbol.iterator]: primesIterator  
}  
  
const myPrimes = [ ...primesIterable ]
```

1

- 1 [2, 3, 5]

This works but it was a pretty cumbersome way to create an iterator. There's a much easier way. You may have

even already picked up on it. In the functions unit, we already covered a type of function that returns objects with `next` and `done` properties: the generator function. The fact is that a generator is both an iterator and an iterable. That means you can iterate a generator directly or use it as an `@@iterator` property to make another object an iterable.

Recreate the same iterator using a generator function:

```
function* primesIterator () {  
  yield 2  
  yield 3  
  yield 5  
}  
  
const primesIterable = {  
  [Symbol.iterator]: primesIterator  
}  
  
const myPrimes = [ ...primesIterable ]      1
```

- 1 [2, 3, 5]

Wow, that was much easier! But because the generator is also an iterable itself, you can use it directly without having to set it as a `Symbol.iterator` first:

```
[ ...primesIterator() ]      1
```

- 1 [2, 3, 5]

Here you just created a simple iterator using a generator to understand how it works. However, the possibilities are endless for how you can use generators to create custom iterators.

Let's go back to the string. When iterated it produces each character in a series. But why each character? Why not each word? This isn't actually a decision that the string makes; it's decided by the default iterator that the

string uses. Override a string's iterator with your own that produces words instead of characters:

```
const myString = Object("Iterables are quite something");
myString[Symbol.iterator] = function* () {
  for (const word of this.split(' ')) yield word;
}
const words = [ ...myString ]
```

1

- **1 [“Iterables”, “are”, “quite”, “something”]**

Here you created a string. You wrapped it in an `Object` invocation to create an object string; otherwise when you update a property it won't stick around. You then set the string's `@@iterator` to your own that naively produces words instead of characters. Now when you spread the string, you get an array of words!

Be careful when overriding an object's `@@iterator`. If you try to iterate the object within its own iterator, you'll create an endless loop! Let's look at an example. Say you want to create an array that, when iterated, produces its values in reverse order. You might try something like this:

```
myArray[Symbol.iterator] = function* () {
  const copy = [ ...this ];
  copy.reverse();
  for (const item of copy) yield item;
}
const backwards = [ ...myArray ]
```

1
2

- **1 Oops, you're trying to iterate yourself within your iterator!**
- **2 Uncaught RangeError: maximum call stack size exceeded**

You see, what's happening here is that within the iterator, you use `[...this]`, which in turn tries to iterate `this` to get the values. This in turn must use the

iterator, but you're already in the iterator, so it's a recursive call!

Oftentimes you'll find yourself wanting to iterate an object's keys and values. It's pretty simple to iterate one or the other using either `for...in` or `for...of`. But to iterate both, most people resort to code that looks like this:

```
for (const key in Object.keys(obj)) {  
  const val = obj[key]  
  // Do something with key and val  
}
```

Here you're iterating the object's keys and then using each key to get the appropriate value. This isn't very elegant. You can use a generator to create an iterator that iterates both:

```
function* yieldKeyVals (obj) {  
  for (const key in obj) {  
    yield [ key, obj[key] ];  
  }  
}
```

This generator takes an object and `yields` an array of the property name and property value for each property. You can use it like so:

```
var address = {  
  street: '420 Paper St.',  
  city: 'Wilmington',  
  state: 'Delaware'  
};  
  
for (const [ key, val ] of yieldKeyVals(address)) {  
  // Do something with key and val  
}
```

Here you're using `for...of` to iterate the key/value pairs. You then use array destructuring to grab those values directly. Pretty neat!

Let's say you're building a social app that allows friends to like status updates. You want to list which friends have liked something. You already have a function called `sentenceJoin` that takes a list of names and joins them for use in a sentence:

```
sentenceJoin(['JD', 'Christina'])  
1  
sentenceJoin(['JD', 'Christina', 'Talan', 'Jonathan'])  
2
```

- **1 JD and Christina**
- **2 JD, Christina, Talan, and Jonathan**

The problem is that if the list of names is very long, you only want to list the first two names, then the number of remaining friends. You can create an iterator for that like so:

```
function* listFriends(friends) {  
  const [first, second, ...others] = friends  
  if (first) yield first  
  if (second) yield second  
  if (others.length === 1) yield others[0]  
  if (others.length > 1) yield `${others.length} others`  
}
```

Now you can format your friends list like so:

```
const friends = ['JD', 'Christina', 'Talan', 'Jonathan']  
  
const friendsList = [ ...listFriends(friends) ]  
  
const liked = `${sentenceJoin(friendsList)} liked this.`  
1
```

- **1 JD, Christina, and two others liked this.**

The iterable and iterator protocol set a foundation for supporting all kinds of iterable objects. In this lesson we primarily looked at strings, arrays, and custom iterables. In the rest of this unit we'll look at completely new

iterables.

Quick check 23.3

Q1:

1. How do you make an object an iterable?
2. Is a generator object an iterable or an iterator?

QC 23.3 answer

A1:

1. By setting its `__iterator` (`Symbol.iterator`) property.
2. It's both.

SUMMARY

In this lesson, you learned the basics of how to use and create your own iterables and iterators.

- An iterable is an object with an `__iterator` property.
- The `__iterator` property must be a function that returns new iterator objects.
- An iterator object must have a `next` method.
- The `next` method on an iterator object must return an object with `value` and/or `done` properties.
- The `value` property is the next value of the iterable.
- The `done` property indicates whether or not all the values have been iterated.

Let's see if you got this:

Q23.1

As we discussed, using `spread` on an infinite iterable would break, because it would continue to ask for values and never stop. Write a function called `take` that accepts two parameters: `n` the number of items to take, and `iterable` the iterable object to take the items from. Create an infinite iterable and take the first 10 values from it. Bonus points if `take` stops early if the iterable runs out of values before `n` has been reached.

Lesson 24. Sets

After reading lesson 24, you will

- Know how to use and create sets
- Know how to perform array operations on sets
- Understand when to use arrays and when to use sets
- Understand what WeakSets are and when to use them

Sets are a new type of object in JavaScript. A set is a unique collection of data. It can store any data type but won't store duplicate references to the same value. Sets are iterables, so you can use spread and `for...of` with them. Sets are most closely related to arrays; however, when you use an array your focus is generally on the individual items in the array. When dealing with a set, you're usually dealing with the set as a whole.

Consider this

Imagine you're building a video game where the player starts with a set of skills and as the player encounters new skills, they're added to the player's skill set. How would you make sure that the player never ends up with duplicate skills?

24.1. CREATING SETS

There's no literal version of a `Set`, such as `[...]` for arrays or `{ ... }` for objects, so sets must be created using the `new` keyword like so:

```
const mySet = new Set();
```

Additionally, if you want to create a set with some initial values, you can use an iterable as the first (and only) parameter:

```
const mySet = new Set(["some", "initial", "values"]);
```

Now this set will have three strings as initial values. Whenever you use an iterable as the parameter, the iterable's individual values get added as items in the set, not the iterable itself:

```
const vowels = new Set("AEIOU");
```

1

- **1 Set {"A", "E", "I", "O", "U"}**

The string "AEIOU" is an iterable for the letters A-E-I-O-U, so the set ends up containing those five individual characters as separate values, not the entire string as a single value. If you want to initialize a `Set` with a single string value, you can do so by putting it in an `Array`:

```
const vowels = new Set(["AEIOU"]);
```

1

- **1 Set {"AEIOU"}**

You can even use another set as the iterable argument to create a new set. This is actually a convenient way of cloning or making a copy of an existing set:

```
const mySet = new Set(["some", "initial", "values"]);
const anotherSet = new Set(mySet);
```

If the iterable passed as an argument to the set constructor has any duplicate values, they'll be ignored and only the first occurrence of each value will be used:

```
const colors = new Set(["red", "black", "green", "black",
"red"]);      1
```

- 1 Set {"red", "black", "green"}

If you create a set with a non-iterable argument, an error will be thrown:

```
const numbers = new Set(36);      1
```

- 1 Uncaught TypeError: undefined is not a function

You get this error because the number 36 has no `Symbol.iterator` function. The error is confusing, but when a `Set` is initialized with a value, the first thing it does is try to iterate the value using its `Symbol.iterator`. If the value given doesn't have an `@@iterator` (as a number doesn't), you get the vague *undefined is not a function* error.

If you want to initialize a `Set` with a single number in it, just wrap the number in an `Array` like so:

```
const numbers = new Set([36]);      1
```

- 1 Set {36}

Now that you know how to create a set, in the next section we'll turn our focus to using them.

Quick check 24.1

Q1:

What will be the difference between the following two sets?

```
const a = new Set("Hello");  
const b = new Set(["Hello"]);
```

QC 24.1 answer

A1:

1. Set {"H", "e", "l", "o"}
2. Set {"Hello"}

24.2. USING SETS

Most of the time, an array is good enough. But if you find yourself needing a unique list of things, you may want to use a set. You should also make your decision based on what you want to do with the list of items. If the operations you want to perform on the list are more array-centric, such as interacting with elements at specific indices or using a method like `splice`, you probably want to use an array. But if you find the API of `Set` more in alignment with the operations you want to perform, such as adding, checking for presence, and removing based on value (as opposed to index), then `Set` is probably your choice. If you find you need a mix of both, it's probably easier to use a set and convert it to an array whenever you need to perform an array operation on it.

Let's imagine you're building a video game that allows a character to move across an area. You render the area using a series of tiles. Every time the character moves, you're given a new set of tiles to render to draw the current state of the game. But to speed up the render time of your game, at each frame you only want to render those tiles which are not already painted on the screen. If you have the currently rendered tiles stored in a set, you can check if they're already rendered using `Set.prototype.has` like so:

```
if ( !frame.has(tile) ) {  
  // paint the tile to the screen  
}
```

Here you have a set named `frame` and you're using `.has()` to determine if the tile has already been drawn

to the screen. Of course, once you paint the tile to the screen, you will want to add it to the set to remember that this tile is painted for the next frame. You can do so using `Set.prototype.has` like so:

```
if ( !frame.has(tile) ) {  
  // paint the tile to the screen  
  frame.add(tile);  
}
```

Now you would also want to remove any frames that are no longer being drawn on the current frame. So you would need to delete all the tiles from your set that are no longer being drawn, as well as add all the new tiles that need to be drawn. Write a function that does this for you, as shown in the next listing.

Listing 24.1. Drawing the next frame

```
function draw(nextFrame) {  
  for (const tile of frame) {  
    if ( !nextFrame.has(tile) ) {  
      frame.delete(tile);  
    }  
  }  
  for (const tile of nextFrame) {  
    if ( !frame.has(tile) ) {  
      // paint the tile to the screen  
      frame.add(tile);  
    }  
  }  
}
```

- **1** Use `for..of` to iterate all the tiles that make up the current frame.
- **2** Check if the next frame has a given tile.
- **3** Remove the tile if the next frame does contain it.
- **4** Use `for..of` to iterate all of the next frame's tiles.
- **5** Check if the current frame doesn't yet have the tile.
- **6** Add the tile if the current frame doesn't already contain it.

OK, let's imagine that as your player travels around, they

can collect new quests. If the player already has a quest in their quest book, you don't want to add a duplicate. If you were using an array, you would have to come up with a strategy to ensure that no duplicate quests get added, but if you were using a set, you would get that functionality for free.

If you want to give an indicator letting the player know how many quests they have, you could use `Set.prototype.size`:

```
const questDisplay = `You have ${quests.size} things to do.`
```

The property `Set.prototype.size` is equivalent to the `length` property of a string or an array. Additionally the method `add` that you've been using is very similar to the `push` method of an array, which adds an item to the end of the array's list of items. But the `delete` function has no array counterpart. You can easily remove an item from an array using `pop` or `shift`. (`pop` removes and returns the last item from the array; `shift` removes and returns the first item, which also causes all the other items indices to shift down by one.) But these functions delete values based on their position, last and first, respectively. The `delete` method of a set specifies to delete a specific value from the set, no matter its position. This concept wouldn't carry over to arrays easily because an array could have the specific value in more than one location. You could convert an array to set and then use `delete` to remove the item, but it would have the side effect of also uniquing the array, which may not be desired. On the flip side, sets have no equivalent methods for `pop` or `shift`. But sets do maintain insertion order, so you could easily convert a set to an

array to get the first or last item:

```
function pop(set) {  
  return [ ...set ].pop();  
}
```

When you use spread with a Set like `[...set]`, we're creating a new Array with all the individual values from the set used as items in the array. This function would return the last item in a set, but it wouldn't remove it because using spread on an iterable does not alter the iterable. The newly created array would have its last item removed, but not the set. To also remove the last item from the set, you would have to make sure you delete it from the set as well:

```
function pop(set) {  
  const last = [ ...set ].pop();  
  set.delete(last);  
  return last;  
}
```

Making a `shift` function would be as simple as internally using `shift` instead of `pop`:

```
function shift(set) {  
  const first = [ ...set ].shift();  
  set.delete(first);  
  return first;  
}
```

Sets maintain their insertion order, and there's no way to rearrange them, short of emptying them completely and adding back the items in a new order. Imagine you're creating a game and storing a set of players. After each round, you want to put the first player in the last position to keep cycling which player goes first for each round. With an array, you could combine `shift` and `push` like so:

```
function sendFirstToBack(arr) {  
  arr.push( arr.shift() );  
}
```

If you wanted to create a new set with the new order, you could convert it to an array, set the order, and return a new set like so:

```
function sendFirstToBack(set) {  
  const arr = [ ...set ];  
  arr.push( arr.shift() );  
  return new Set(arr);  
}
```

If you needed to actually change the order of the existing set, you could do so by emptying out the entire set using `Set.prototype.clear`. But there's no method to add multiple items to a set at once. In order to add the items back to the set after the order has been changed, you would need to add them back individually using `Set.prototype.add` like so:

```
function sendFirstToBack(set) {  
  const arr = [ ...set ];  
  set.clear();  
  arr.push( arr.shift() );  
  for(const item of arr) {  
    set.add(item);  
  }  
}
```

- **1 First get all the items from the set into an array.**
- **2 Remove all items from the set.**
- **3 Reorder the array.**
- **4 Add the items back to the set in the new order.**

Now that you know how to use a set, in the next section we'll take a look at when you might want to use a set versus an array, and vice versa.



Quick check 24.2

Q1:

What's the fundamental difference between the array methods `Array.prototype.shift`, `Array.prototype.pop`, and the set method `Set.prototype.delete`?

QC 24.2 answer

A1:

The array methods `Array.prototype.shift` and `Array.prototype.pop` remove items based on their position (index) in the array, first and last, respectively. The set method `Set.prototype.delete` removes an item based on the value of the item itself.

24.3. WHAT ABOUT THE WEAKSET?

A `WeakSet` is a specialized type of `Set`. Its sole purpose is to contain objects *weakly* in the sense that it doesn't prevent them from being garbage-collected. Normally if you add an object to an array and remove all other references to it, it will still not be eligible for garbage collection because the array still has a reference to the object. This is the same for sets as well. Sometimes you may want to store an object without preventing it from being garbage-collected, though.

Say you're building an MMO (massively multiplayer online) game and want to use a set to determine which players are currently spawning to keep them from being killed while spawning. You could add them to a spawning set like so:

```
function spawn(player) {
  spawning.add(player);
  // ... do stuff, (possibly set a timeout for N seconds)
  spawning.delete(player);
}
```

Then anything else that tries to attack the player can first check if the player is currently spawning before adding damage to it:

```
function addDamage(player, damage) {
  if (!spawning.has(player)) {
    // add damage to player
  }
}
```

If spawning is a normal set, then if the player leaves the game or possibly loses their internet connection while spawning, you'll need to make sure that you remove the player from the spawning set. That might not seem too

difficult, but what happens if you have several sets that are keeping track of players for various reasons? Having to make sure you remove all references to the player when they exit the game can become a hassle. But if we used `WeakSets` that wouldn't be necessary, as the `WeakSet` doesn't prevent an item from being garbage-collected.

In order to achieve this, a `WeakSet` has no reference to any of the items it contains. You have to already have a reference to the item in order to check if a `WeakSet` contains it. This is OK for our use case, as in your game you already have a player and want to check if that player is in the spawning `WeakSet`, so it works.

Because a `WeakSet` has no reference to its items, `WeakSets` cannot be iterated. Thus, a `WeakSet` isn't actually an iterable like the `Set` is. You can't use `for...of` or `spread` on it. A `WeakSet` also can only contain object values: primitive values aren't allowed, and trying to add one will throw an error.

SUMMARY

In this lesson, you learned how to create and use sets and why you would use one over an array.

- A set has a new literal and must be created using `new`.
- You can create a set using an iterable as an argument.
- You can clone a set by creating a new set with an existing set as an argument.
- Sets are iterables and thus can be used with spread and `for...of`.
- You add a value to a set using `Set.prototype.add`.
- You can tell if a set has a value using `Set.prototype.has`.
- You can remove a value from a set using `Set.prototype.delete`.
- You can empty a set by using `Set.prototype.clear`.
- You can determine how many items a set contains with `Set.prototype.size`.
- A `WeakSet` is not an iterable.
- A `WeakSet` can only contain objects.
- A `WeakSet` doesn't prevent its contents from being garbage-collected.
- A `WeakSet` has no way to inspect its contents.

Let's see if you got this:

Q24.1

Create the following helper functions to make dealing with sets even more useful:

- `union`—A function that takes two sets and returns a new set with all the values from both sets
- `intersection`—A function that takes two sets and returns a new set with only the values that are in both sets
- `subtract`—A function that takes two sets and returns a new

set with all of the values from the first set, excluding any values that are in the second set

- `difference`—A function that takes two sets and returns a new set of only the values they aren't in both sets (the opposite of intersection)

Go further. For more fun you can update these functions to work on any number of sets, not just two.

Lesson 25. Maps

After reading lesson 25, you will

- Know to create a Map
- How to convert a plain object into a Map
- How to add and access values on a Map
- How to iterate the keys and values from a Map
- How Maps are destructured
- Understand when it makes sense to use a Map over an Object
- Understand why it makes sense to use a WeakMap over a Map

Like Sets, Maps are a new type of object in JavaScript. Not to be confused with `Array.prototype.map`, which is a higher-order array method, Maps are another type of iterable that's new to JavaScript in ES2015. A Map is a lot like a generic object in JavaScript with *keys* and *values*. But a Map can be iterated, whereas an object can't. Plain objects are also limited to only being able to have string values as keys;^[1] Maps, on the other hand, can have any datatype as a key, even another Map!

¹

Technically, strings and symbols.



Consider this

When you need a container of data with no need for qualified identifiers for each datum, you may reach for an array. When you need to store data and be able to retrieve specific data based on a string identifier, you may reach for an object. But what if you need to retrieve data based on a more complex identifier such as a DOM node or something else that can't be used as the key of an

object? What do you reach for then?

25.1. CREATING MAPS

As with `Sets`, there's no literal version of a `Map` and they must be created using the `new` keyword like so:

```
const myMap = new Map();
```

Remember from the last lesson that you can instantiate a `Set` with a single array argument, to define the initial values for the set? You may be thinking that because a `Map` is a series of keys and values, you can instantiate a `Map` with initial values by passing in a single object parameter. This isn't the case, though. But if you think about it, this makes sense. An array can contain any datatype that a set can, but an object's keys are limited to strings, whereas a `Map`'s keys can be any datatype. So using an object as the initial parameter would limit what kind of keys you could use. Instead, to instantiate a `Map` with initial values, you use one large array containing smaller arrays of key/value pairs like so:

```
const myMap = new Map([
  ["My First Key", "My First Value"],
  [3, "My key is a number!"],
  [/\/S/g, "My key is a Regular Expression object!"]
]);
```

Notice how you used different types of objects as keys for your map: first a string, then a number, then a `RegExp` object. All these are valid keys for a map and they won't get converted into strings like they would for an object.

If you do find yourself wanting to convert an `Object` into a `Map`, you can do it like so:

```
const myMap = new Map(Object.keys(myObj).map(key => [ key,
myObj[key] ]))
```

Here you used `Object . keys` to get an array of all the properties of `myObj`. You then used `Array . prototype . map` to convert the array of keys into an array of key/value pairs. The key/value pairs are then used as the argument to the `Map` constructor.

Don't get confused or caught up in the fact that you're using `Array . prototype . map` in conjunction with `Map`. The `map` method of an array is an action that maps values from one array to transformed values in a new array. The `Map` object, on the other hand, maps keys to values. It isn't an action or transformation, but a data store that allows you to access values based on keys. You give a `Map` a key and get back a value; in a sense it maps the key to the value.

Quick check 25.1

Q1:

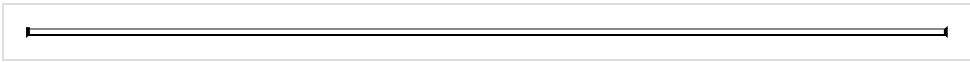
Which of the following are valid ways to create a new `Map`?

```
const a = new Map({ foo: "bar" })
const b = new Map([ "foo", "bar" ])
const c = new Map([ [ "foo", "bar" ] ])
const d = new Map([ { foo: "bar" }, [] ])
const e = new Map()
```

QC 25.1 answer

A1:

a and b are both invalid. c, d, and e are all valid.



25.2. USING MAPS

At this point you may be asking, “But aren’t Maps just Objects too? Does JavaScript treat them differently from other objects?” To answer this question we first need to define some terms. In this lesson as I’ve been comparing Maps to Objects, I’ve been implying what’s commonly referred to as a *POJO* (*plain old JavaScript object*), meaning a plain Object used as a data structure with its properties used as keys and values. A Map’s keys are not its properties. A Map extends from Object just like all other objects, and its properties must be strings. But a map stores its keys and values internally, not as properties, which is how it gets away with being able to use any datatype.

If you use `Object.keys` to get a map’s properties, you won’t get the keys that you set: instead you’ll get an empty array:^[2]

2

Even though the Map does have properties such as `size`, they aren’t listed because they’re on the Map prototype, not the specific instance.

```
Object.keys(myMap);
```

1

- **1 Returns []**

If you do want to get a Map’s keys (not properties), you can use `Map.prototype.keys` like so:

```
myMap.keys();
```

1

- **1 { “My First Key”, 3, /\S/g }**

This doesn’t return an array of the keys; instead it returns an iterator that allows you to iterate the keys. But

remember you can easily turn an iterator into an array if needed:

```
[ ...myMap.keys() ]
```

- 1 [“My First Key”, 3, /\S/g]

Maps also have `Map.prototype.values` for getting, you guessed it, the values. And likewise, it returns an iterator, not an array:

```
myMap.values();
```

- 1 { “My First Value”, “My key is a number!”, ... }

Sets actually have both these methods as well. But both `Set.prototype.keys` and `Set.prototype.values` return the same thing, since a `Set` doesn’t use keys, only values. You’ll actually notice that the APIs of `Map` and `Set` are very consistent with each other but not identical. For example, they both have the `size` property as well as the methods shown in [table 25.1](#) in common.

Table 25.1. Common methods shared by Map and Set

Method	Description
<code>clear</code>	Removes all values from the Map (or Set).
<code>delete</code>	Removes a specified value from the Map (or Set).
<code>entries</code>	Returns an iterator for accessing all the keys and values.
<code>forEach</code>	Similar to the same method on an Array. Loops over all the keys and values of a Map.
<code>has</code>	Checks if a Map has a given key (or a Set has a given value).
<code>keys</code>	Returns an iterator for accessing all the keys.
<code>values</code>	Returns an iterator for accessing all the values.

However, `Set` has the method `Set.prototype.add` for adding values and `Map` uses the confusingly named `Map.prototype.set` method for adding entries:

```
myMap.set("My next key", "My next value");
```

Map also has the method `Map.prototype.get` for retrieving a value for a specific key:

```
myMap.get("My next key");      1
myMap.get("Some other key");   2
```

- **1 “My next value”**
- **2 undefined**

I am not sure why the TC39 committee went so far as to ensure that the API be consistent, even including a redundant `keys` method on `Set`, only to fall short and not actually have identical APIs.

It’s also worth noting that while the methods on `Set` deal with the values as parameters, such as `delete(value)`, Maps deal with keys such as `delete(key)`.

Quick check 25.2

Q1:

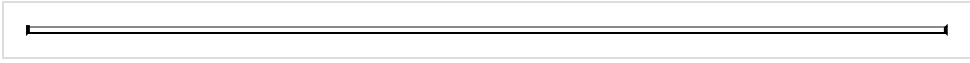
What’s the difference between the last two lines of code?

```
const myMap = new Map([[ "a", 1], [ "b", 2]]);
Object.keys(myMap);
myMap.keys();
```

QC 25.2 answer

A1:

- `Object.keys(myMap)` returns the map’s own enumerable properties (it has none).
- `myMap.keys()` returns the keys associated with the stored data (“a” and “b”).



25.3. WHEN TO USE MAPS

You may be asking, “Should I now use a Map anytime I need keys and values?” The answer is probably not. Most of the time you’ll still probably want to use regular objects. Regular objects are lighter-weight and more expressive when all you need is a structure to pass around values.

Say, for example, you have a function where you can specify options like `width` and `height`. Doing so is easy with object destructuring like so:

```
function renderWithOptions({ width, height, ... }) {  
  // do something with width, height and other options  
}
```

You can’t destructure a Map by the names of its keys; again this would assume that all the Map’s keys are strings. What if the Map’s keys were some type of object such as dates? They would be impossible to destructure this way. Maps as well as all iterables can be destructured, but with array-style destructuring:

```
const options = new Map();  
options.set('width', 400);  
options.set('height', 90);  
options.set(new Date(), 'now');  
  
const [a, b, c] = options;  
console.log(a);           1  
console.log(b);           2  
console.log(c);           3
```

- **1** [“width”, 400]
- **2** [“height”, 90]
- **3** <date object>, “now”]

You could use nested array destructuring to access the

keys and values like so:

```
const [ [ widthKey, width ], [ heightKey, height ] ] = options;
console.log(widthKey);           1
console.log(width);              2
console.log(heightKey);          3
console.log(height);             4
```

- **1 “width”**
- **2 400**
- **3 “height”**
- **4 90**

Not only is this far more verbose, it also assumes the width is the first item in the Map and the height is the second. This is because this type of destructuring is based on index or position in the Map, not on the name of the key or property:

```
const [ [ heightKey, height ], [ widthKey, width ] ] = options;
console.log(heightKey);          1
console.log(height);             2
console.log(widthKey);           3
console.log(width);              4
```

- **1 “width”**
- **2 400**
- **3 “height”**
- **4 90**

Plain objects do have some limitations, though, and that’s where the Map steps in. Plain objects have keys, so you can organize your data by specific values, but objects don’t guarantee any specific order. Arrays, on the other hand, do guarantee their order, but you can only access values in an array by an index, not a specific identifier. With maps you can do both.

So if you need to iterate keys and values and the order is

important, use a Map. Even if the order isn't important, it's still more straightforward to iterate keys and values with a Map.

Let's look at the difference. Say you're using a faceted search on your website. You know, like the kind of search that allows you to narrow your search based on facets like price, brand, or color. You want to list all the facets used for the search and display them as shown in [figure 25.1](#).

Figure 25.1. Displaying keys and values of search facets



In order to display them, you need to iterate each facet (key and value) and add it to the screen. If you were using a Map to store the facets, you could iterate them like so:

```
for (const [name, value] of facets) {  
  // render facet name and value  
}
```

If you were storing the facets in a plain object, to iterate them you would need an extra step:

```
for (const name of Object.keys(facets)) {  
  const value = facets[name];  
  // render facet name and value  
}
```

The other limitation of objects is that their properties must be strings. Sometimes you may need to use an

object as a key. Let's say you want to build a function called `Singleton` that can convert any constructor into a singleton instance.^[3] Usually special care is taken in the design of a `Singleton` object to ensure that only one instance is ever created—typically with a static `getInstance` method. But you can use a `Map` to easily add this functionality to any type of object, as the next listing shows.

3

A `Singleton` is a constructor/class that can have only one instance. See https://en.wikipedia.org/wiki/Singleton_pattern.

Listing 25.1. A module that guarantees a singleton instance of any object type

```
const instances = new Map();           1
export default function Singleton(constructor) { 2
  if (!instances.has(constructor)) {          3
    instances.set(constructor, new constructor()); 4
  }
  return instances.get(constructor);          5
}
```

- **1 Create a map to store a single instance for each constructor. This map is not exported and is hidden from the rest of the application.**
- **2 The `Singleton` function accepts a constructor as an argument.**
- **3 First checks if the map already has an instance for this constructor**
- **4 If not, it creates an instance of the constructor and adds the instance to the map with the constructor as the key.**
- **5 Retrieves and returns the single instance for the given constructor.**

That's all that's needed. Here you use a function called `Singleton` that takes the constructor function for any object. It uses a map to store constructors to instances. It first checks whether it already has an instance for a given constructor and if not, creates one. Then it returns the

instance. This ensures that only one instance is ever created:

```
import Singleton from './path/to/single/module';
Singleton(Array)                                1
Singleton(Array).length                         2
Singleton(Array).push("new value")              3
Singleton(Array).push("another value")          3
Singleton(Array).length                         4
const now = Singleton(Date)
setTimeout(() => {
  const later = Singleton(Date)
  now === later                                5
}, 10000)
```

- **1 Creates an empty array**
- **2 Array has length zero**
- **3 Adds two elements to the array**
- **4 Array has length two**
- **5 Returns true as now and later are the same object**

This is just one of many use cases where you may want to use an object as a key. You may need DOM nodes to register themselves with an action or a way to tie data to DOM nodes:

```
const domData = new Map()
function addDomData(domNode, data) {
  domData.set(domNode, data);
}
function getDomData(domNode) {
  domData.get(domNode);
}
```

Or you may want to tie data to an object without altering the object itself. This could be characters in a video, DOM nodes, or objects used by frameworks that you aren't supposed to change. Anytime you may want to store data about another object external to the object itself, a Map makes perfect sense.

Imagine you're building a real estate website. You're

working with multiple listing services; each one provides you with a collection of listings that are for sale. Each listing has a listing ID (`listingId`), and to keep from colliding with listing IDs from other providers, each listing also has a multiple listing ID (`mlsId`). You're going to be showing all the listings for a particular area, meaning you're going to need to have listings from multiple providers in memory at once. In order to identify a listing, you need to use the `listingId` and the `mlsId`. Because of this, you may think it's easier to store them in a `Map` using an array like `[mlsId, listingId]` as the key for each listing. This wouldn't be ideal, though, because in order to get a listing from the `Map` you would need a reference to the exact array you used to add it, not just any array with the correct `mlsId` and `listingId`:

```
const listings = new Map()
listings.set(['mls37', 'listing29'], /* ... some listing */ )

// ...

listings.has(['mls37', 'listing29'])      1
```

- **1 false**

Why did the `listings` `Map` tell you that it didn't have the listing that you know it has? Even though you used the correct `mlsId` and the correct listing ID, you created a new array containing them, which is a different object. A `map` looks at the object itself, not at the contents of the object. The object you provide to a `Map` as a key must be the same exact object. You can think of this as if the object must satisfy triple equals `===` equality. But that isn't 100% true, either, because you can use `NaN` as a valid key in a `Map` even though `NaN !== NaN`.

Take a look at this code. It allows setting any keys and values on an object. But what if the key is coming from an untrusted source? How do you know the key won't end up being something that would be dangerous to set on an object like `toString`, `proto`, and so on?

```
const data = {};  
function set(key, val) {  
  data[key] = val;  
}
```

By using a `Map` instead of an `Object`, it would be protected from such problems because a `Map`'s keys are not its properties and thus wouldn't collide and override built-in properties.

Quick check 25.3

Q1:

What's the difference between the `width` and `w` constants that are destructured from `myObj` and `myMap`, respectively?

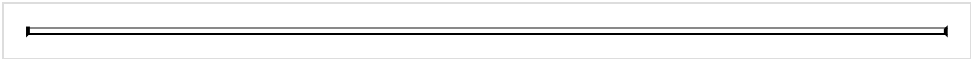
```
const myObj = {}, myMap = new Map();  
myObj['width'] = 400;  
myObj['height'] = 50;  
myMap.set('width', 400);  
myMap.set('height', 90);  
  
const { height, width } = myObj;  
const [ h, w ] = myMap;
```

QC 25.3 answer

A1:

`width` is 400 as expected but `w` is `["height",`

90].



25.4. WHAT ABOUT THE WEAKMAP?

Similar to what you learned about the `WeakSet` in [lesson 21](#), a `WeakMap` is a non-iterable subset of the `Map`. But whereas the `WeakSet` has weak references to its values, the `WeakMap` has weak references to its keys.

Additionally, a `WeakMap` can only have objects as its keys. The `WeakMap` doesn't keep its keys from becoming eligible for garbage collection. Since the `WeakMap` isn't iterable, there's no way to get a list of all the keys of a `WeakMap`. Also there's no way to check if a `WeakMap` has a specific key unless you already have a reference to that key. Once there are no more references to a key, it will no longer be accessible from the `WeakMap` and will become garbage-collected.

You may want to use a `WeakMap` when you're trying to avoid memory leaks and you don't need to iterate all the keys/values in the `Map`, but only need to access the value once you already have a reference to the key. Say, for example, you want to store metadata about some objects. Maybe the objects are logged-in players of a game and they may sign off. Or maybe the objects are DOM nodes that may be removed from the DOM as the UI changes. If you store metadata based on these objects in a regular `Map`, you're creating a memory leak because the `Map`'s reference to these objects will prevent them from being garbage-collected. A `WeakMap` wouldn't prevent them from being garbage-collected and thus wouldn't create a memory leak. If you need to use a `Map`, though, because you need some additional features like iteration, you can still use it; you just need to manage removing the objects from the `Map` when necessary.

SUMMARY

In this lesson you learned how to create and use Maps and why you would use one over a regular object.

- To instantiate a new Map with keys and values, you use an array of array pairs as the parameter, not an object.
- Maps have many of the same properties and methods as Sets.
- You use `Map.prototype.set(key, value)` to add a new key/value pair to a Map.
- A Map's keys are stored internally and not related to its properties.
- A Map uses array-style, not object-style destructuring.
- Maps are iterated more elegantly than regular objects.
- Maps guarantee their order, unlike objects.
- Maps can have any datatype as a key.
- WeakMaps must have an object as a key.
- WeakMaps don't prevent their keys from being garbage-collected.
- WeakMaps are not iterable.

Let's see if you got this:

Q25.1

Write the following three helper functions to alter Maps:

1. `sortMapByKeys`—A function that returns a copy of a Map sorted by its keys
2. `sortMapByValues`—A function that returns a copy of a Map sorted by its values
3. `invertMap`—A function that returns a copy of a map with its keys and values inverted

In real-world use, these functions need to take into account that, according to the definition of how Maps work, their keys must be unique while their values

may contain duplicates. To simplify this exercise, assume that these functions will only be operating on Maps that have unique values.

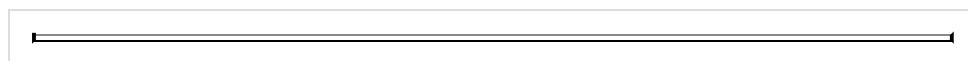
Lesson 26. Capstone: Blackjack

In this capstone you're going to build a Blackjack (21) game like the one shown in [figure 26.1](#).

Figure 26.1. A Blackjack game



A game of cards has many sets. A deck is a *set* of cards and each player's hand is also a *set* of cards. You'll also make use of *maps* and eventually use a generator to create a function that slows down the looping of an iterator so that it can be animated on the screen.



Note

You'll start your project using the start folder included in the code accompanying this book. If at any time you get stuck, you can also check out the final folder with the completed game. The start folder is a project already set up to use Babel and Browserify (see [lessons 1–3](#)); you just need to run `npm install` to get set up. If you haven't read unit 0, you should before doing this capstone.

There's also an included `index.html` file: this is where the game will run. It already includes all the HTML and CSS it needs; you just need to open it in a browser once you bundle your JavaScript files. The `src` folder is where

you'll put all your JavaScript files; there are a couple already included. The `dest` folder is where the bundled JavaScript file will go after you run `npm run build`. You'll need to remember to run `npm run build` to compile your code anytime you make a change.

Your project is going to start with a couple of modules already created, specifically an `elements` module and a `templates` module. In order to make the game's code easy to reason about, you'll continue to create modules for handling specific parts of the game. You're going to start by building a `cards` module that handles all the hard related tasks like creating a deck, shuffling it, or counting a player's hand of cards. You'll then create a `utils` module that's going to store a helper function you'll need. Finally you'll use all of these modules in your main index file that orchestrates the game. Because you're properly using modules, the index file should be rather simple and easy to follow.

26.1. THE CARDS AND THE DECK

This is a card game, after all, so get started by creating and storing your cards. Each card will be a simple object with a *suit* and a *face*. To assist in creating cards, you can store sets of the available suits and faces. Create a file called `src/cards.js` and use a `Set` to store all the possible card suits as shown in the next listing.

Listing 26.1. `src/cards.js`

```
const suits = new Set(['Spades', 'Clubs', 'Diamonds',  
  'Hearts']);
```

You can also use a set to store all the possible card faces, as the following listing shows.

Listing 26.2. `src/cards.js`

```
const faces = new Set([  
  '2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K',  
  'A'  
]);
```

You also need a way to determine each face's value. For this you can use a `Map`, shown next.

Listing 26.3. `src/cards.js`

```
const faceValues = new Map([  
  ['2', 2], ['3', 3], ['4', 4], ['5', 5], ['6', 6], ['7', 7],  
  ['8', 8],  
  ['9', 9], ['10', 10], ['J', 10], ['Q', 10], ['K', 10]  
]);
```

Here you're using a map to store the face value for each card face except for the ace. Since the ace's value is contextual and can be either 1 or 11, you can't store its value this way and will have to treat it differently.

The game is going to render cards. Each card can be displayed face down or face up. The cards are going to be

passed around from the deck to each player, so keeping track of which ones are flipped could be tricky. This becomes easy with a Map, though. You can store the actual card as the key in your Map, and whether or not it's flipped as the value. This means as long as you have a reference to the card, you can determine if it's flipped face up or face down, as shown in the following listing.

Listing 26.4. `src/cards.js`

```
export const isCardFlipped = new Map();

export function flipCardUp(card) {
  isCardFlipped.set(card, true);
}

export function flipCardDown(card) {
  isCardFlipped.set(card, false);
}
```

Now write a function for creating a deck of cards, shown in the next listing.

Listing 26.5. `src/cards.js`

```
export function createDeck() {
  const deck = new Set();
  for (const suit of suits) {
    for (const face of faces) {
      deck.add({ face, suit });
    }
  }
  shuffle(deck);
  return deck;
}
```

This function is simple. You first create a new Set for the deck. Then you're just using `for . . of` to loop through all the suits and faces to add a card to the deck for all the possible combinations (52, to be exact). The final thing you do is call `shuffle(deck)`. Any card game would require shuffling the deck before use, and Blackjack is no exception, so you'll write the shuffle function.

Sets maintain their entries in insertion order, but they don't store keys, meaning there's no index associated with a value. Because of this, to shuffle a set, you'll need to use an array.

Listing 26.6. src/cards.js

```
export function shuffle(deck) {  
  const cards = [ ...deck ];  
  1  
  let idx = cards.length;  
  while (idx > 0) {  
    idx--  
  2  
    const swap = Math.floor(Math.random() * cards.length);  
  3  
    const card = cards[swap];  
  4  
    cards[swap] = cards[idx];  
  5  
    cards[idx] = card;  
  6  
  }  
  deck.clear();  
  7  
  cards.forEach(card => deck.add(card));  
  8  
}
```

- **1 You use spread to put all the Set's values into an Array.**
- **2 The index of the current card**
- **3 Get a random index to another card to swap with your current card.**
- **4 Get the card at your swap index.**
- **5 Set the card at your swap index to be the current card.**
- **6 Set the card at your current index to be the swapped card.**
- **7 Clear the Set before adding the shuffled cards back.**
- **8 Add the shuffled cards back to the deck in the new order.**

In this function, you first grab all the cards from the set into an array using spread. You then go through each index in the array and swap the card at that index with

another card at a random index. The same index might get chosen to swap several times, and that's fine. This results in the array of cards being randomly shuffled. Finally you need to add them all back to the deck. You must first clear the deck, since each card can be in the set only once.

Now that you have a way to create and shuffle a deck, the next thing you'll need to do is deal the cards to each player. When you deal a card from the deck, you generally deal from the top. As you know, arrays have a built-in method for this type of thing called `pop`. Sets don't have a corresponding method, so you'll build your own, as shown in the next listing.

Listing 26.7. `src/cards.js`

```
export function pop(deck) {  
  const card = [ ...deck ].pop();  
  isCardFlipped.set(card, true);  
  deck.delete(card);  
  return card;  
}
```

In this `pop` function, you use `Array`'s `pop` to get the last card from the deck. You then default the card to being face up. Most games deal face down, but in this game you'll almost always want the card face up, so default it that way. You use `delete` to remove the card from the deck, since it obviously can't be in a player's hand and remain in the deck at the same time. Finally you return the card.

In Blackjack, each player starts with two cards. In your game, each player's hand of cards will also be represented with a `Set`. Write a small function that deals two cards from the deck to a given hand, as in the following listing.

Listing 26.8. src/cards.js

```
export function dealInitialHand(hand, deck) {  
  hand.add(pop(deck));  
  hand.add(pop(deck));  
}
```

The whole point of Blackjack is to have the total of your cards be as close to 21 as possible without going over, so you need a function to count the total face value of a hand of cards. You can total the face values by iterating all the cards and checking each card's face with the `faceValues` map that you created earlier. But that won't work for the ace, which can be either 1 or 11. So in your initial loop of all the cards, you can add 1 for each ace as well as keep track of each ace. After the initial loop, you can add an additional 10 for each ace when the total of cards will still be less than or equal to 21, as shown in the next listing.

Listing 26.9. src/cards.js

```
export function countHand(hand) {  
  let count = 0; 1  
  const aces = new Set(); 2  
  for (const card of hand) {  
    const { face } = card;  
    if (face === 'A') { 3  
      count += 1; 3  
      aces.add(card);  
    } else {  
      count += faceValues.get(face); 4  
    }  
  }  
  for (const card of aces) {  
    if (count <= 11) { 5  
      count += 10; 5  
    }  
  }  
  return count;  
}
```

- **1 Start your count at zero.**
- **2 Create a Set to keep track of all the aces.**
- **3 For every ace, add one, and keep track of the ace.**

- **4 If not an ace, add the value from the faceValues Map.**
- **5 For all of the aces, add an additional 10 as long as it won't result in the count going over 21.**

That's about all you need in this cards module. You now have the functions to create a deck of cards, shuffle that deck, deal the initial two cards to a hand, and count the total value of the cards in a hand. You also have a way to keep track of whether a given card is flipped up or down. In the next section, you'll write a function to delay CPU decisions enough for the player to see what's happening in real time.

26.2. MAKING THE CPU'S TURN SLOW ENOUGH TO SEE

The game will start by allowing the user to take cards from the deck until the user passes control to the dealer, at which time the dealer will be able to draw cards from the deck. Since the dealer is a CPU, it will be able to make decisions so fast that you'll need to slow them down in order to allow the human player to see what's happening.

You can do this by using a generator function and using a `yield` anytime you want to add a pause. You'll need another function that can iterate your generator and pause at every occurrence of `yield`. Create the file `src/utls.js` and add this function as shown in the next listing.

Listing 26.10. `src/utls.js`

```
export function wait(iterator, milliseconds, callback) {  
  const int = setInterval(() => {  
    const { done } = iterator.next();  
    if (done) {  
      clearInterval(int);  
      callback();  
    }  
  }, milliseconds);  
  1  
  2  
  3  
  4  
  4  
  5  
}
```

- **1 Create a new interval using the milliseconds parameter.**
- **2 Get the next iteration from the iterator.**
- **3 Check if the iterator is done.**
- **4 If the iterator is done, clear the interval and invoke the callback function.**
- **5 Create a new interval using the milliseconds parameter.**

The `wait` function takes three parameters: an iterator

object, the number of milliseconds to wait between iterations, and a callback function. You use `setInterval` with the given `milliseconds` parameter. Inside your interval function, the arrow function you pass to `setInterval`, you call `iterator.next()`. This will cause the iterator to attempt to retrieve the next value every `milliseconds` of time. You check the `done` value from your iterator to clear the interval if the iterator has finished. Now this will work with any iterator, causing a pause between each iteration, but if you use a generator it will allow you to specify when to pause the execution of your function anytime you use `yield`.

For example, when using the `wait` function in [listing 26.11](#), you would first log A after one second. Then because you invoked `yield`, it would wait for the next interval of the `wait` function. Then it would log both B and C, but it would wait another second before logging D because you used another `yield`.

Listing 26.11. Using the `wait` function

```
function* example() {  
  console.log('A');           1  
  yield;                      2  
  console.log('B');           3  
  console.log('C');           3  
  yield;                      4  
  console.log('D');           5  
}  
wait(example(), 1000, () => {  
  console.log('Done.')        6  
})
```

- **1 'A' will get logged after 1 second.**
- **2 Here you're causing another delay.**
- **3 'B' and 'C' will get logged after 2 seconds.**
- **4 Here you're causing another delay.**

- 5 'D' will get logged after three seconds.
- 6 'Done.' will get logged after the example generator has completely finished.

Later you'll use this `wait` function when the dealer is deciding to draw cards from the deck. You'll use `yield` to delay the next decision long enough to animate each card being added to the player's hand.

26.3. PUTTING THE PIECES TOGETHER

Now that you've gotten the cards module and some necessary utils taken care of, you need to put it all together to complete the game. First import everything you're going to need for your main game logic. Create the file `src/index.js` and add the following imports.

Listing 26.12. `src/index.js`

```
import { cardTemplate } from './templates';
import { wait } from './utils';

import {
  dealerEl, playerEl, buttonsEl, updateLabel, status, render,
  addCard
} from './elements';

import {
  createDeck, pop, countHand, dealInitialHand, flipCardDown,
  flipCardUp
} from './cards';
```

That sure is a lot of stuff you're importing! By separating your logic into cohesive modules, your *glue code* (the main game logic) that ties everything together will be much simpler. You wrote everything in the `utils.js` and `cards.js` files together. The `templates.js` and `elements.js` focus mostly on interacting with the DOM (Document Object Model). The logic in those files doesn't have a lot to do with iterators, so I omitted going over their contents, but feel free to browse the code.

The first thing you need to play a game of cards is the deck. So create one using the function you imported, as shown in the following listing.

Listing 26.13. `src/index.js`

```
const deck = createDeck();
```

1

- **1** Creates a new Set with 52 shuffled card objects

Now create a couple Sets for the dealer's and the player's hands, shown in the next listing.

Listing 26.14. src/index.js

```
const dealerHand = new Set();
dealInitialHand(dealerHand, deck);
flipCardDown([ ...dealerHand ][0]);          1

const playerHand = new Set();
dealInitialHand(playerHand, deck);
```

- **1 Flip the dealer's first card face down.**

Here you created a new set for each player's hand. You used the `dealInitialHand` function that you created earlier to deal each player their initial two cards. You leave all the player's cards face up so they can see what they're holding. You set the dealer's first card to be face down and leave their second card face up, like an actual casino dealer would do, as shown in [figure 26.2](#).

Figure 26.2. The dealer's initial hand



Now in order to generate that screenshot, you need to render each hand. You can do this using the `render` function you imported from the `elements` module, as the following listing shows.

Listing 26.15. src/index.js

```
render(dealerEl, dealerHand);
render(playerEl, playerHand);
```

OK, that should get the initial game rendered on the screen, but there won't be any interactivity yet. You need to write some functions to allow the user to play by deciding to *hit* or to *stay*, and you need to program the dealer to do the same. Start with the dealer while the

wait function you wrote earlier is hopefully still fresh in your memory.

Create the following generator function.

Listing 26.16. src/index.js

```
function* dealerPlay() {  
  dealerEl.querySelector('.card').classList.add("flipped");  
  1  
  while( countHand(dealerHand) < countHand(playerHand) ) {  
    2  
      addCard(dealerEl, dealerHand, pop(deck));  
    3  
      yield;  
    4  
  }  
  if ( countHand(dealerHand) === countHand(playerHand) ) {  
    5  
      if (countHand(dealerHand) < 17) {  
        5  
          addCard(dealerEl, dealerHand, pop(deck));  
          yield;  
        6  
      }  
    }  
  }  
}
```

- **1 Flip all the dealer's cards face up.**
- **2 Continue to loop while the dealer's hand is less than the player's.**
- **3 Draw another card from the deck.**
- **4 Delay long enough to animate the card being added to the dealer's hand.**
- **5 If the dealer's hand is the same as the player's and the total score is less than 17, draw another card.**
- **6 Wait again for the final card to animate.**

In this generator function, you first flip over all the dealer's cards so the player can see what the dealer has. Then the dealer should continue to add cards until they've reached the player's total score. You don't need to worry about the player's hand being over 21 because if the player busts (goes over 21), the game will end before

the dealer needs to play.

Once the dealer has tied the player, if the total score is less than 17, the dealer should draw another card to attempt to win. If the score is 17 or over, the dealer should accept the tie (push) because it's too risky to draw another card.

In this generator function, every time the dealer draws a card, you use `yield`. When combined with the `wait` function you wrote in the last section, this will cause a delay long enough to allow the card to animate on the screen before the dealer needs to decide to draw another. This will create a nice effect of cards being added to the screen as the dealer plays.

Now you just need a small function that combines this generator with your `wait` function, shown in the next listing.

Listing 26.17. `src/index.js`

```
function dealerTurn(callback) {  
  wait(dealerPlay(), 1000, callback);  
}
```

Now let's turn our attention to the user. The user playing the game will have two choices, to hit or to stay. So you need to write a function for each case. Let's start with hit.

Listing 26.18. `src/index.js`

```
function hit() {  
  addCard(playerEl, playerHand, pop(deck));  
  updateLabel(playerEl, playerHand);  
  const score = countHand(playerHand);  
  if (score > 21) {  
    buttonsEl.classList.add('hidden');  
    status('Bust!');  
  } else if (score === 21) {  
    stay();  
  }  
}
```

In this `hit` function, you start by adding a card to the player's hand from the deck. Then you call `updateLabel`. This changes the text from score to blackjack or bust if the player reaches 21 or busts, respectively. Then you check the score and if the player has gone bust, you hide the buttons and update the status of the game. Otherwise, if the player has exactly 21, you automatically invoke `stay` and turn the game to the dealer. You haven't written that `stay` function, though, so do that in the next listing.

Listing 26.19. `src/index.js`

```
function stay() {  
  buttonsEl.classList.add('hidden');  
  1 dealerEl.querySelector('.score').classList.remove('hidden');  
  2  
  dealerTurn(() => {  
  3    updateLabel(dealerEl, dealerHand);  
  4    const dealerScore = countHand(dealerHand);  
  5    const playerScore = countHand(playerHand);  
  5  
    if (dealerScore > 21 || dealerScore < playerScore) {  
      status('You win!');  
  6    } else if (dealerScore === playerScore) {  
      status('Push.');  7    } else {  
      status('Dealer wins!');  8    }  
  });  
}
```

- **1 Hide the hit and stay buttons.**
- **2 Reveal the dealer's score.**
- **3 Lets the dealer play out their turn**
- **4 Once the dealer has finished, update their label.**
- **5 Count both the dealer's and the player's hands.**

- **6 If the dealer has more than 21 or less than the player, the player wins.**
- **7 If they're tied, it's a push**
- **8 Otherwise, the dealer wins**

There's only one last step to complete this game. You need to add some event listeners to the hit and stay buttons, as shown in the next listing.

Listing 26.20. src/index.js

```
document.querySelector('.hit-me').addEventListener('click', hit);
document.querySelector('.stay').addEventListener('click', stay);
```

With that, you're finished. You should now be able to play a game of Blackjack against your computer. Building this game really helped me realize how much the odds are stacked against you when playing against the house.

SUMMARY

In this capstone, you built a blackjack game by breaking your logic into cohesive modules. You started with elements and template modules. You then created a cards module followed by a utils module. Your cards module represented a deck of cards as well as hands of cards using Sets. You also used a Map to keep track of whether specific cards were face up or face down. Once you had your game logic separated into logical modules, tying the pieces together to make your game in the root index file was nice and easy.

You can take this game further by adding a Play Again button and keep track of the odds that the player beats the dealer.

Unit 6. Classes

JavaScript is an object-oriented language. Except for a few primitives, most values in JavaScript, even functions, are objects. Most of the JavaScript code that gets written is for creating custom objects. Many times you're probably building your own objects from scratch, but sometimes you'll be starting from a base object provided from a framework or library and adding custom functionality to it. Unfortunately, there hasn't been a clear way for library authors to provide a base object prototype that can be extended by application developers. This has left many library authors needing to reinvent the wheel.

But nobody reinvents the wheel the same way. Here's how Backbone.js provides a way to extend base objects:

```
Backbone.Model.extend({  
  // ... new methods  
});
```

And here's how it's done with React.js:

```
React.createClass({  
  // ... new methods  
});
```

Backbone provides a method called `initialize` that acts as a pseudo constructor function. React does not. But React does autobind its methods when they're created with `createClass`. And this is just the beginning of the differences: we quickly compared two here, but there are dozens of JavaScript frameworks that provide a base object, which means that you'd need to remember all the subtle differences of how different

classes work depending on what library or framework you're using.

With classes, however, framework authors can now provide a base class that's extendable at the language level. This means that once you learn how to extend classes with one framework, you'll be able to transfer that knowledge, making it much easier to switch and learn new frameworks.

Lesson 27. Classes

After reading lesson 27, you will

- Understand the syntax for defining classes
- Know how to instantiate classes and how to use constructor functions
- Know how to export classes from modules
- Understand that class methods are not bound
- Know how to assign class and static properties

Classes are little more than syntactic sugar for declaring a constructor function and setting its prototype. Even with the introduction of classes, JavaScript isn't statically or strongly typed. It remains a dynamically and weakly typed language. However, classes do create a simple self-contained syntax for defining constructors with prototypes. The main advantage over constructors, though, besides syntax, is when extending classes, which we'll get to in the next lesson. Without having an easily extendable built-in construct such as classes, many libraries like Backbone.js, React.js, and several others had to continue to reinvent the wheel to allow extending their base objects. This library-specific form of extending objects will become a thing of the past as more and more libraries start to provide a base JavaScript class that can easily be extended. This means once you learn how to use and extend classes, you'll have a jump start on many of today's and tomorrow's frameworks.

Consider this

The following two functions are current JavaScript patterns of instantiating objects: the factory function and

the constructor function. If you were to design a new syntax for instantiating objects, how would you improve it?

```
const base = {
  accel() { /* go forwards */ },
  brake() { /* stop */ },
  reverse() { /* go backwards */ },
  honk() { /* be obnoxious */ }
}

function carFactory (make) {
  const car = Object.create(base);
  car.make = make;
  return car;
}

function CarConstructor(make) {
  this.make = make;
}

CarConstructor.prototype = base;
```



27.1. CLASS DECLARATIONS

Let's say you'll building a web application that needs to connect to several APIs for various resources, such as users, teams, and products. You may want to create several store objects that store the records for the various resources. You could create a store class like so:

```
class DataStore {  
    // class body  
}
```

Here you declared a class with the keyword `class` followed by the class name, *DataStore*. The name doesn't have to be capitalized, but it is a convention to do so. After the name of the class, you have a pair of braces `{}`: the class body, all the methods and properties that make up the class, go between these two braces.

Let's say, for example, that you need a method called `fetch` on your `DataStore` class to handle fetching the records from the database. You would add a method like so:

```
class DataStore {  
    fetch() {  
        // fetch records from data base  
    }  
}
```

Adding a method here is the exact same syntax as the shorthand method names on objects that you learned about in [lesson 12](#). Don't let this fool you, though: a class definition is not an object and other syntaxes won't work. The attempt to create a method in the next listing will result in a syntax error.

Listing 27.1. Incorrect syntax for adding methods to classes

```
class DataStore {  
  fetch: function() {  
    // fetch records from data base  
  }  
}
```

1

- **1 Syntax Error**

Class methods work like any other function in JavaScript: they accept parameters and can use destructuring and default values. In fact, it's not the syntax of the function here that causes the error, but the use of a colon (:). Colons are used to separate property names and property values on object literals, but a class declaration doesn't use them. Also, any use of `this` inside a method will refer to the instance, not the class.

Another difference between the class syntax and object syntax is that there are no commas separating properties in a class like there are in an object:

```
class DataStore {  
  fetch() {  
    // fetch records from data base  
  }  
  update() {  
    // update a record  
  }  
}
```

1

- **1 No comma separating the two methods**

Notice how there are no commas separating the class methods. If you were creating an object, you would need a comma or you would get a syntax error. But in a class declaration, the opposite is true in that adding commas will result in a syntax error.

Quick check 27.1

Q1:

Can you spot the two problems in the following car class?

```
class Car {  
  steer(degree) {  
    // turn the car by degree  
  },  
  accel(amount=1) {  
    // accelerate the car by amount  
  }  
  break: function() {  
    // decelerate the car  
  }  
}
```

QC 27.1 answer

A1:

Commas and colons are not allowed.

27.2. INSTANTIATING CLASSES

Once you have your class definition, you can create an instance of it using the `new` keyword:

```
const dataStore = new DataStore();          1
```

- **1 Create an instance of `DataStore` in a new constant named `dataStore`.**

If you try to instantiate the class without the `new` keyword, you receive a `TypeError`:

```
const myStore = DataStore();                1
```

- **1 Class constructor `DataStore` can't be invoked without 'new'**

You can also pass arguments to your new class instance when creating an instance like so:

```
const userStore = new DataStore('users');
```

Of course, this raises the question: how does the class receive these parameters? There's a special method name called `constructor` that gets invoked automatically when an instance is created. Any arguments given when creating a new instance will be passed to this constructor function.

```
class DataStore {  
  constructor(resource) {  
    // setup an api connection for the specified resource  
  }  
}
```

The constructor function gets invoked in the context of the instance that's being created. That means that `this` inside of the constructor function will refer to the

instance that's being created, not the class itself. The constructor function is optional, and is provided as a hook to do any initial setup on instances of your classes. The placement of the constructor function doesn't matter, but I like to keep mine at the top.

In the next section we'll take a quick look at exporting classes from modules.

Quick check 27.2

Q1:

What will be logged when creating the following class?

```
class Pet {  
  constructor(species) {  
    console.log(`created a pet ${species}`);  
  }  
}  
  
const myPet = new Pet('dog');
```

QC 27.2 answer

A1:

“created a pet dog”

27.3. EXPORTING CLASSES

When creating JavaScript classes, it's likely that you'll do so in modules. That means that you'll need to be able to export them. For example, if you wanted to export the `DataStore` class you created from a module, you could do it like this:

```
export default class DataStore {           1  
  // class body  
}
```

- **1 Specify the `DataStore` class as the default export.**

This exports the class as the default. Importing it is the same as importing any other default; you can use any name you desire:

```
import Store from './data_store'
```

Additionally you can omit the default and export the class by name:

```
export class DataStore {                   1  
  // class body  
}
```

- **1 Export the `DataStore` class under the name `DataStore`.**

As you probably guessed, this exports the class as the name `DataStore` and it must be imported using its name:

```
import { DataStore } from './data_store'
```

In most cases you'll likely want to only create one class per module and export it as the default.



Quick check 27.3

Q1:

Which of the following two class exports are valid?

```
export class A {  
  // class body  
}  
export default class B {  
  // class body  
}
```

QC 27.3 answer

A1:

They're both valid.

27.4. CLASS METHODS ARE NOT BOUND

Some developers new to classes may be surprised to find out that class methods are not automatically bound. Let's say, for example, that the `DataStore` class uses an `ajax` library internally to handle the API calls. You may have something set up like the next listing.

Listing 27.2. Using unbound methods as callbacks

```
class DataStore {  
  fetch() {  
    ajax(this.url, this.handleNewRecords)    1  
  }  
  handleNewRecords(records) {  
    this.records = records                    2  
  }  
}
```

- **1 Making an ajax call and specifying `this.handleNewRecords` as the callback**
- **2 The keyword `this` will not point to the instance because it isn't bound.**

In [listing 27.2](#) you'll use an `ajax` library to fetch records for you. The `ajax` library takes two arguments: the URL to load data from, and a callback function to invoke with the data once loaded. This may seem like everything is fine, but since the `handleNewRecords` method isn't bound, when it's invoked from the `ajax` library, `this` will no longer point to the `DataStore` instance so the records won't get properly stored.

There are a few different ways to fix this. The simplest is to use an arrow function as the callback:

```
class DataStore {  
  fetch() {  
    ajax(this.url, records => this.records = records)  
  }  
}
```


This will work just fine. But if your callback is more complex and is used in multiple places, this approach may not be suitable. You can still use an arrow function, but as an intermediary step:

```
class DataStore {
  fetch() {
    ajax(this.url, records => this.handleNewrecords(records) )
  }
  handleNewRecords(records) {
    // do something more complex like
    // merging new records in with existing records
  }
}
```

In the next lesson, you'll learn one more way of binding methods when you learn about class properties.

Quick check 27.4

Q1:

Invoking `car.delayedHonk()` will result in an error. Why?

```
class Car {
  honk() {
    this.honkAudio.play();
  }
  delayedHonk() {
    window.setTimeout(this.honk, 1000);
  }
}

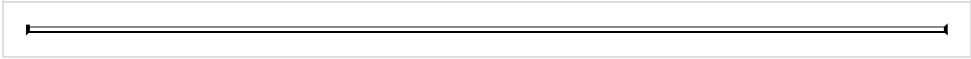
const car = new Car();
car.delayedHonk();
```

QC 27.4 answer

A1:

Because `this.honk` is used as a callback but is not

bound.



27.5. SETTING INSTANCE PROPERTIES IN CLASS DEFINITIONS

There's another aspect of classes that, at the time of this writing, is just a proposal but is gaining lots of use, and that's setting instance properties in class definitions. Setting properties in classes (unlike methods) may seem simple at first, but actually has more to it than might initially meet the eye. This is because the methods are added to the prototype, but the properties are assigned to each instance.

Let's assume you want to set up a property on your `DataStore` to determine the URL of the resource. You also want an array property to store the actual records in. You could do so as shown in the following listing.

Listing 27.3. Setting properties on classes

```
class DataStore {  
  url = '/data/resources';           1  
  records = [];                     2  
  
  fetch() {  
    ajax(this.url, records => this.records = records)  
  }  
}  
console.log(DataStore.url);          3  
  
const store = new DataStore();  
console.log(store.url);              4  
console.log(store.records.length);  5
```

- **1 A property named url**
- **2 A property named records**
- **3 undefined**
- **4 “/data/resources”**
- **5 0**

The properties, as you can see, will be available on the

instance that gets created, not the class itself. Notice how class properties look like an assignment directly in the class definition. This is because they technically are an assignment. Unlike class methods, class properties are a syntactic sugar that gets rewritten directly in the constructor function. That means the `url` and `records` assignments you just set would actually end up being as if you had made them inside the constructor function:

```
class DataStore {  
  constructor() {  
    this.url = '/data/resources';  
    this.records = [];  
  }  
}
```

This is different from class methods, because those get added to the class's prototype. The instances then inherit those methods via prototypical inheritance. But the class properties don't get set on the prototype; they end up being set directly on the instance itself. Setting properties directly on the prototype would lead to bugs. Let's explore why in an example:

```
class DataStore {};  
  
DataStore.prototype.records = [];  
  
const storeA = new DataStore();  
  
console.log(storeA.records.length);           1  
  
const storeB = new DataStore();  
  
console.log(storeB.records.length);           1  
  
storeB.records.push('Example Record')  
  
console.log(storeA.records.length);           2  
console.log(storeA.records[0]);               3
```

- **1 0**

- **2 1**

- **3 ‘Example Record’**

Here you set the `records` property directly on the `DataStore`’s prototype instead of the instance. This means all instances will use the same `records` array because it’s set on their prototype, not on the instance objects themselves. When object `storeB` adds a record, `storeA` receives a new record as well. They’re sharing the same `records` array!

One obvious benefit of class instance properties is declaring bound methods. In the last section, you were running into some issues because your `handleNewRecords` method wasn’t bound to the instance. You could make it bound by declaring it as a property pointing to an arrow function:

```
class DataStore {
  fetch() {
    ajax(this.url, this.handleNewrecords)
  }
  handleNewRecords = (records) => {
    // do something more complex like
    // merging new records in with existing records
  }
}
```

In the next section we’ll take a look at another type of class property, static properties.

Quick check 27.5

Q1:

The following class suffers from a syntax mistake common with class properties. Can you spot it?

```
class Nachos {
  toppings: ['cheese', 'jalapenos']
}
```

QC 27.5 answer

A1:

It should be

```
toppings = ['cheese', 'jalapenos'];
```

27.6. STATIC PROPERTIES

Static properties on classes are a special type of property that doesn't set a property on the instance or even the prototype, but on the class object (the constructor) itself. Static properties make sense for properties that won't change across instances. For example, your `DataStore` could potentially have a static property for what domain to use when connecting to APIs:

```
class DataStore {  
  static domain = 'https://example.com';      1  
  static url(path) {                          2  
    return `${this.domain}${path}`  
  }  
  constructor(resource) {                    3  
    this.url = DataStore.url(resource);  
  }  
}  
  
const userStore = new DataStore('/users');  
console.log(userStore.url);                  4
```

- **1 Assigning a static property**
- **2 Setting a static method**
- **3 Invoking the static method**
- **4 “https://example.com/user**

You use a static property `domain` and a static method `url`[1]` to generate the instance's URL from the specified resource in the constructor function.

1

While static properties are at the time of this writing still just a proposal, static methods were introduced in ES2015.

Static properties are just syntactic sugar for assigning them directly on the classes themselves, as shown in the next listing.

Listing 27.4. Desugaring static properties

```
class DataStore {
  constructor(resource) {
    this.url = DataStore.url(resource);
  }
}

DataStore.domain = 'https://example.com';
DataStore.url = function(path) {
  return `${this.domain}${path}`
}
```

That sums up creating and using classes, but we aren't done with classes yet. In the next lesson we'll take a look at extending classes.

Quick check 27.6

Q1:

Which of the following `console.log`s actually logs the number of wheels?

```
class Bicycle {
  static numberOfWheels = 2;
}
const bike = new Bicycle();

console.log(bike.numberOfWheels);
console.log(Bicycle.prototype.numberOfWheels);
console.log(Bicycle.numberOfWheels);
```

QC 27.6 answer

A1:

```
console.log(Bicycle.numberOfWheels);
```

SUMMARY

In this lesson you learned how to define and use your own classes.

- Classes are created with the `class` keyword followed by the class name and class body.
- Class methods use the shorthand method syntax for declaring methods.
- Classes don't support colons for declaring properties or methods.
- Classes shouldn't have commas separating methods or properties.
- The constructor function is executed when the class is instantiated.
- Class methods aren't automatically bound to the instance.

Let's see if you got this:

Q27.1

Take the following constructor function and prototype and convert it into a class:

```
function Fish(name) {
  this.name = name;
  this.hunger = 1;
  this.dead = false;
  this.born = new Date();
}
Fish.prototype = {
  eat(amount=1) {
    if (this.dead) {
      console.log(`${this.name} is dead and can no longer
eat.`);
      return;
    }
    this.hunger -= amount;
    if (this.hunger < 0) {
      this.dead = true;
      console.log(`${this.name} has died from over
eating.`)
      return
    }
  },
  sleep() {
    this.hunger++;
    if (this.hunger >= 5) {
```

```
        this.dead = true;
        console.log(`${this.name} has starved.`)
    }
},
isHungry: function() {
    return this.hunger > 0;
}
}

const oscar = new Fish('oscar');
console.assert(oscar instanceof Fish);
console.assert(oscar.isHungry());
while(oscar.isHungry()) {
    oscar.eat();
}
console.assert(!oscar.isHungry());
console.assert(!oscar.dead);
oscar.eat();
console.assert(oscar.dead);
```

Lesson 28. Extending classes

After reading [lesson 28](#), you will

- Know how to extend classes to create more customized classes
- Learn how to use libraries that provide a base class
- Understand how inheritance works with classes
- Know how to use `super` to invoke functions on the superclass

The best feature classes offer over traditional constructor function declarations is their ease of extension. Many consider the syntax for extending constructor functions clunky. This has led many library authors to create their own ways to extend base objects in their libraries. With built-in extendable classes, developers can learn one simple syntax that can be used everywhere. One crucial aspect to keep in mind when extending classes in JavaScript, however, is that they still use prototypal inheritance.

Consider this

In the following code, you have a `plane` object and a `jet` object. In a sense, the `jet` object extends the `plane` object because it copies all of its properties and overrides its `fly` method. If the `jet`'s `fly` method has overlapping logic with the `plane`'s `fly` method, how would you go about making them able to share code across the two methods?

```
const plane = {
  type: 'aircraft',
  fly() {
    // make the plane fly
  }
}
const jet = Object.assign({}, plane, {
```

```
fly() {  
  // make the jet fly faster  
}  
  
});
```



28.1. EXTENDS

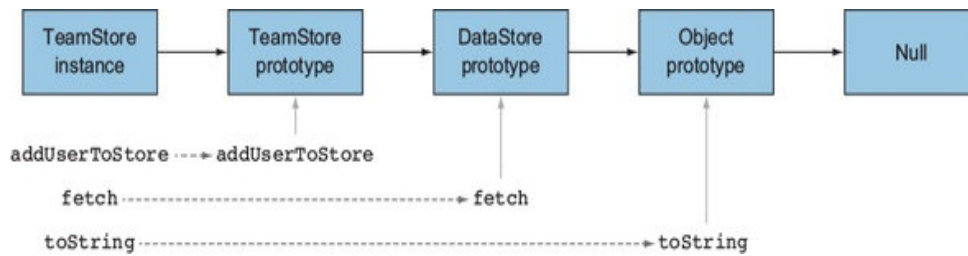
Let's continue to use the `DataStore` example from [lesson 27](#). Assume you wanted to create a `TeamStore` that was a customized version of a `DataStore`. Your `TeamStore` needs the base functionality of a `DataStore`, but also needs a couple additional methods for adding or removing users from a team. You could create a `TeamStore` that extends the `DataStore` as shown in the following listing:

Listing 28.1. Extending a class

```
class TeamStore extends DataStore {
    addUserToTeam(teamId, userId) {
        // add user to team
    }
    removeUserFromTeam(teamId, userId) {
        // remove user from team
    }
}
```

Here in [listing 28.1](#) you created a new class called `TeamStore` that extends from `DataStore`. By making `TeamStore` extend from `DataStore`, the `DataStore` prototype gets appended to the `TeamStore` prototype chain. This means that when an instance of `TeamStore` is created, the prototype chain will be as follows: the `TeamStore` instance delegates to the `TeamStore` prototype, which in turn delegates to the `DataStore` prototype that it inherited from. The `DataStore` prototype delegates back to the `Object` prototype, which delegates to `null` (see [figure 28.1](#)).

Figure 28.1. Prototypal inheritance



When you create an instance of `TeamStore` and invoke methods on the instance, if the instance doesn't have the method in question, the instance's prototype that the `DataStore` defined is then checked. If the method is still not found, it continues down the prototype chain looking for the method. This isn't new: this is how inheritance has always worked in JavaScript, and that hasn't changed with classes.

Quick check 28.1

Q1:

What's the full prototype chain of the `migaloo` object?

```

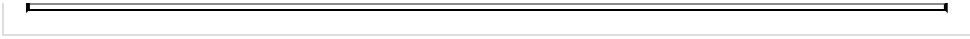
class Whale extends Animal {
  // whale stuff
}
class Humpback extends Whale {
  // humpback stuff
}

const migaloo = new Humpback();
  
```

QC 28.1 answer

A1:

instance → Humpback → Whale → Object → null



28.2. SUPER

When you defined `DataStore` in the previous example, you made the constructor function accept a parameter indicating the URL of the resource it was responsible for:

```
class DataStore {  
    constructor(resource) {  
        // setup an api connection for the specified resource  
    }  
}
```

This means that you would still need to specify the resource when creating an instance of `TeamStore`, even though `TeamStore` always uses the same URL. You could make this automatic in the constructor of `TeamStore` by using `super`, as shown in the following listing.

Listing 28.2. Invoking the superclass constructor with `super`

```
class TeamStore extends DataStore {  
    constructor() {  
        super('/teams');  
    }  
}
```

- **1 Invoking `super()` invokes the constructor of the super class.**

Inside `TeamStore`'s constructor, you invoked the constructor of its superclass (the class it extends from) using the special keyword `super`. This allows you to automatically set the `TeamStore`'s URL to `"/teams"`.

Inside a constructor function of any class that extends another, the invocation of `super` is required before any reference to `this` can be made:

```
class TeamStore extends DataStore {  
    constructor() {
```



```
    this.url = '/teams';  
    super(this.url);  
  }  
}
```

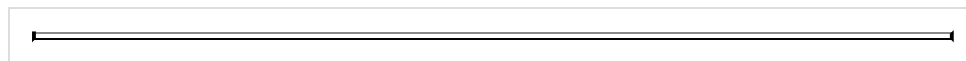
- **1 ReferenceError: this is not defined**

The keyword `super` doesn't actually reference anything. It's a special keyword that allows you to invoke the constructor of the superclass or access and invoke other methods on the superclass. Let's say, for example, that you have a products store and you want to update an analytics object about which products are accessed. You could override the `DataStore`'s `fetch` function to add your extra logic. However, you will still want the original `fetch` function (the one you overrode) to be invoked, and you can do so using `super.fetch` as shown in the next listing.

Listing 28.3. Invoking superclass methods with `super`

```
class ProductStore extends DataStore {  
  fetch(id) {  
    analytics.productWasViewed(id);  
    return super.fetch(id);  
  }  
}
```

When you reference `super[name]` you can access any property by name on the superclass's prototype. Here you used `super.fetch` to invoke the `fetch` method on the superclass.



Quick check 28.2

Q1:

What's wrong with the following code?

```
class Whale extends Animal { }  
class Humpback extends Whale {
```

```
    constructor() {  
        this.hasHump = true;  
    }  
}
```

QC 28.2 answer

A1:

this can't be referenced in the constructor of an
extending class until `super ()` is invoked.

28.3. A COMMON GOTCHA WHEN EXTENDING CLASSES

In the previous lesson we went over a way to bind methods by setting class properties that point to arrow functions:

```
class DataStore {
  fetch() {
    ajax(this.url, this.handleNewrecords)
  }
  handleNewRecords = (records) => {
    // merging new records in with existing records
  }
}
```

This is a common pattern that I've seen all over in the wild. The problem with this approach is that `handleNewRecords` doesn't get added to the `DataStore` prototype. It gets added directly onto the instance that gets created. This means that it can't be accessed with `super`.

If you don't plan on the method ever needing to be invoked via `super` in a subclass, then this approach is fine. Otherwise you'll need a new approach. So how can you bind a method but still make it extendable and invokable via `super`? One approach is to define it as an actual method, then bind the method to the instance inside the constructor, as the following listing shows.

Listing 28.4. Binding methods in a way that works with `super`

```
class DataStore {
  constructor(resource) {
    // setup an api connection for the specified resource
    this.handleNewRecords = this.handleNewRecords.bind(this);
  }
  handleNewRecords(records) {
    // merging new records in with existing records
  }
}
```

```

}

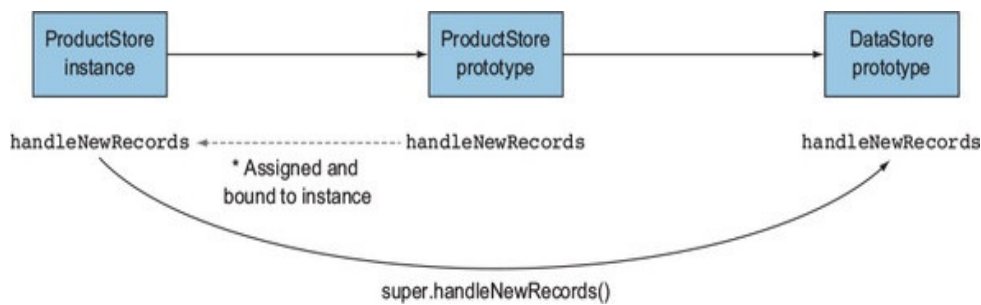
class ProductStore {
  handleNewRecords(records) {
    super.handleNewRecords(records);
    // do something else
  }
}

```

- **1 Bind the `handleNewRecords` method from the prototype onto the instance.**

This works because it still assigns the method to the instance, but the definition originates from a method on the prototype. The subclass can override the instance method on the superclass, and even use `super`, and the subclass's overridden instance method will then get bound to the instance. See [figure 28.2](#).

Figure 28.2. How the `handleNewRecords` method gets invoked on the prototype chain



Quick check 28.3

Q1:

Why won't the following code work?

```

class Whale {
  dive = () => {
    // go deep!
  }
}
class Humpback extends Whale {
  dive = () => {
    super.dive();
  }
}

```

QC 28.3 answer

A1:

Both `dive` properties are set directly on the instance, so the second one completely overrides the first. There is no `dive` method on the prototype chain, so it can't be accessed via `super`.

SUMMARY

In this lesson, you learned how to extend classes.

- You extend classes using the `extends` keyword.
- Classes use prototypal inheritance.
- You can access the superclass's constructor and methods with the `super` keyword.
- `super` must be invoked in the constructor of a subclass before `this` can be referenced.
- Class properties are set on the instance, and thus can't be accessed with `super`.

Let's see if you got this:

Q28.1

Write a class that extends the following `Car` class. Add a method called `fuel` that resets the gas back to 50. Additionally, override the `drive` method to accept a parameter of a number called `miles`, then invoke the superclass's `drive` method enough times to drive that many miles:

```
class Car {
  constructor() {
    this.gas = 50;
    this.milage = 0;
  }

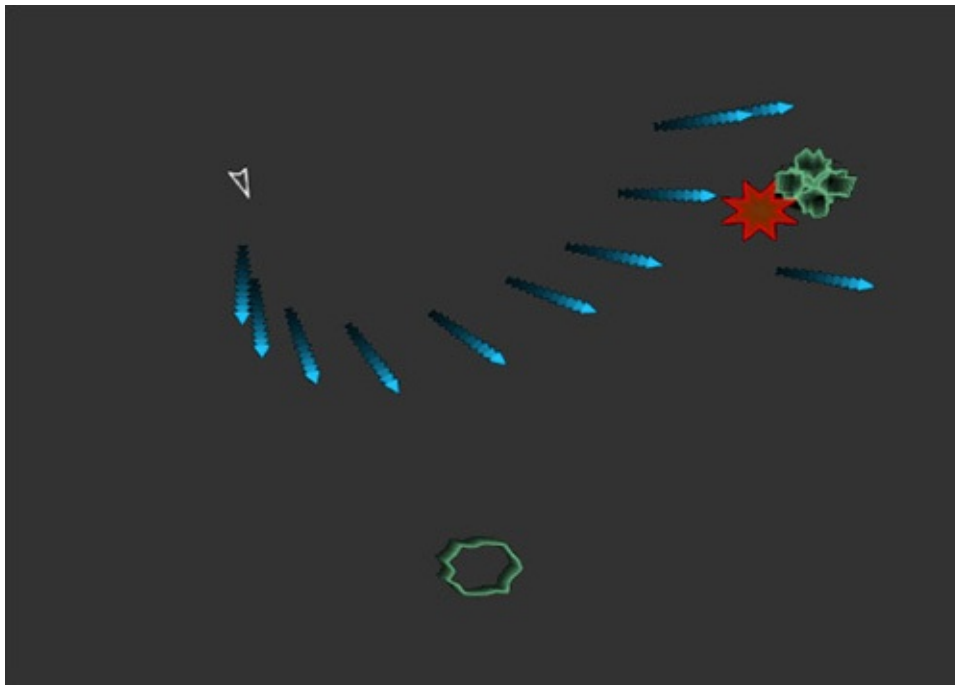
  hasGas() {
    return this.gas > 0;
  }

  drive() {
    if (this.hasGas()) {
      this.milage++;
      this.gas--;
    }
  }
}
```

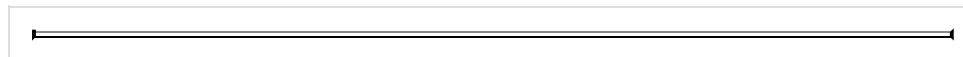
Lesson 29. Capstone: Comets

In this capstone you're going to build an Asteroids-like game called Comets, shown in [figure 29.1](#).

Figure 29.1. Comets game



There are many valid reasons why you would create a base class or a class that doesn't extend anything. But I think the majority of most developers' time spent with classes will be extending base classes provided from a framework such as React.js. For this capstone, you're going to be working with a game framework that I have put together: the framework (like most frameworks) will handle most of the moving parts while giving you some base classes you can customize to create a unique game.



Note

You'll start your project using the start folder included in the code accompanying this book. If at any time you get stuck, you can also check out the final folder with the completed game. The start folder is a project already set

up to use Babel and Browserify (see unit 0); you just need to run `npm install` to get set up. If you haven't read unit 0, you should before doing this capstone.

There's also an included `index.html` file: this is where the game will run. It already includes all the HTML and CSS it needs; you just need to open it in a browser once you bundle your JavaScript files. The `src` folder is where you'll put all your JavaScript files; there are a couple already included. The `dest` folder is where the bundled JavaScript file will go after you run `npm run build`. You'll need to remember to run `npm run build` to compile your code anytime you make a change.



29.1. CREATING A CONTROLLABLE SPRITE

You'll start out by getting something on the screen as quickly as possible. Make sure you're starting with the start folder included in the code accompanying this book. If you look in the src folder, you'll see there's already a framework folder and a shapes.js file. The framework folder contains the game framework, and the shapes.js file includes some arrays and functions for describing the shape of the game's actors. Feel free to explore the code in any of these files.

Create a file in the src directory called comet.js with the following code.

```
import ControllableSprite from
'./framework/controllable_sprite';    1
import { ship as shape } from './shapes';
2

export default class Ship extends ControllableSprite {
3
  static shape = shape;
4
  static stroke = '#fff';
4
}
```

- **1 The base class comes from the game framework you're using.**
- **2 The spaceship from your included shapes**
- **3 Your ship class extends the controllable sprite class.**
- **4 Setting some static properties**

This is a simple class, but it does a lot because you're extending from `ControllableSprite`, which itself extends from `Sprite`. The `Sprite` class is the baseline for any object that will be painted on the screen. The `ControllableSprite` class adds functionality for the

user to control the sprite using the keyboard's arrow keys. You set the static properties `shape` and `stroke`, which determine what Ship sprites will look like. Go ahead and experiment with other shapes and colors once you get this one painted on the screen.

You're almost ready to see this painted on the screen, and even control it. Create another file in the `src` folder called `index.js` and add the following code.

```
import { start } from './framework/game';      1

import Ship from './ship';                     2

new Ship();                                    3

start();                                       4
```

- **1 The start function from the framework starts the game.**
- **2 Import the class you previously created.**
- **3 Create an instance of your ship class.**
- **4 Start the game.**

This code is also pretty simple. You're just creating an instance of your `Ship` class and telling the game to start. The `index.js` file will be the root file of your source code that starts the game. If you compile your code now and open the included `index.html` file, you'll see the ship you just created. Even more, by using the arrow keys you can move it around!

29.2. ADDING COMETS

Now that you can move around, your ship looks rather lonely. You're going to add some additional actors to your game. In the src folder create a file called comet.js with the following code:

```
import { CANVAS_WIDTH, CANVAS_HEIGHT } from
'./framework/canvas';
import DriftingSprite from './framework/drifting_sprite';
import { cometShape } from './shapes'

function defaultOptions() { 1
  return {
    size: 20,
    sides: 20,
    speed: 1,
    x: Math.random() * CANVAS_WIDTH,
    y: Math.random() * CANVAS_HEIGHT,
    rotation: Math.random() * (Math.PI * 2)
  }
}

export default class Comet extends DriftingSprite { 2
  static stroke = '#73C990';

  constructor(options) { 3
    super(Object.assign({}, defaultOptions(), options)); 4
    this.shape = cometShape(this.size, this.sides); 5
  }
}
```

- **1 Default options for each comet**
- **2 The comet extends the DriftingSprite class.**
- **3 The constructor accepts options.**
- **4 Invoke super with options and default options merged together.**
- **5 Set the comet's shape.**

The `Comet` class is a bit more complex than the previous two examples, so let's dive into what's going on here. All sprites in the game framework can be given initial options (properties) as arguments to the constructor.

You didn't need any for your ship, but you do for each comet. The sprites are flexible in how to set their properties. They can be static like the `stroke` property, or set on the instance like the `shape` property.

What's the purpose of setting some properties as static and some on the instance? All of your comets will have the same green stroke, so you can set that as a static property on the class itself. But you want each comet to have a unique shape so that it looks like an individual comet, not carbon copy. Because each comet's shape is unique, it needs to be set on the instance.

The function `cometShape` returns a random shape with a given size and sides. By calling this function in the constructor, you ensure that each comet has its own unique shape.

Now that you have your `Comet` class, you need to import it and create a few instances before starting your game.

Update the `index.js` file like so:

```
import { start } from './framework/game';

import Ship from './ship';
import Comet from './comet';           1

new Ship();

new Comet();                           2
new Comet();                           2
new Comet();                           2

start();
```

- **1 Import your Comet class.**
- **2 Create a few instances.**

Now if you bundle your code again, you should see some comets floating around with your ship.

29.3. SHOOTING ROCKETS

Obviously now that you have all these floating targets, you want to do some shooting! Next you'll get some rockets on the screen. First create the rocket class. Create a new file in the src folder called `rocket.js` with the following code:

```
import DriftingSprite from './framework/drift_sprite';
import { rocket } from './shapes';

export default class Rocket extends DriftingSprite {
  static shape = rocket;
  static fill = '#1AC2FB';
  static removeOnExit = true;
  static speed = 5;
}
```

Nothing here is all that new yet. This is very similar to your comet, albeit with a different shape, of course. You also gave your rocket a fill instead of a stroke. One new property, though, is `removeOnExit`. Normally when sprites exit the screen, they reappear on the other side. This property tells your game to remove the sprite once it leaves the viewable area.

Now you need to get some rockets on the screen. You don't want to just create instances before starting the game like you did with the ship and comets. You want to *fire* rockets from the ship at your command! There are a few helper functions provided by the game framework that you're going to need to pull this off. Open the `ship.js` file and add the following imports to the top:

```
import { isPressingSpace } from './framework/interactions';
import { getSpriteProperty } from './framework/game';
import { throttle } from './framework/utils';
import Rocket from './rocket';
```

The `isPressingSpace` function tells you if the player is pressing the spacebar. You'll use this to indicate the intent to fire a rocket. When you fire a rocket, you want to set the rocket to start in the same location and direction as your ship so that it appears to be firing from your ship. Because the sprites can have their properties stored as static or instance properties or even methods, you need the special helper method `getSpriteProperty` to read the values.

During the course of the game, the screen is being re-rendered over and over. Before rendering, each sprite is notified to update itself via a hook in the form of a method on the sprite called `next()`. The `next` function will always be called on each sprite before painting the next frame, and must always return the sprite itself. You can use this hook to see if the user is pressing the spacebar and, if so, fire a rocket. The problem is that each frame renders very fast. You don't want to shoot rockets that fast; you want to slow it down. That's where the `throttle` function comes in. You give it a function and it returns a throttled function. Let's see how to make it happen. Add the following method to the ship:

```
next() {  
    super.next();  
  
    if (isPressingSpace()) {  
        this.fire();  
    }  
  
    return this;  
}
```

- **1** You're overriding `next` so you need to call `super.next`.
- **2** If the user is pressing the spacebar, you want to fire a rocket.
- **3** You still need to add this method.

- **4 The next function always needs to return this.**

Here you overrode the `next` function, so you need to invoke `super.next`. Keep in mind that when I say you need to, I don't mean it's required at the language level. If you don't want the `next` method of the superclass to also be invoked, you don't need to call `super.next()`. But in this case, you do want to.

You still need to add the `fire` method. Because you need this method to be throttled, you can't set it as a regular method. Instead, you'll set it as a class property that points to a throttled function:

```
export default class Ship extends ControllableSprite {
  static shape = shape;
  static stroke = '#fff';

  fire = throttle(() => {                                1
    const x = getSpriteProperty(this, 'x');
    const y = getSpriteProperty(this, 'y');
    const rotation = getSpriteProperty(this, 'rotation');

    new Rocket({ x, y, rotation });
  });

  ...
}
```

- **1 Setting the throttled function as a class property**

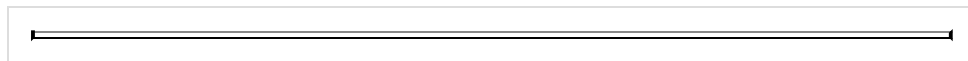
Because `fire` is set as a property, not a method, it isn't added to the prototype. It's set directly on the ship instance when created. This means that any class that extended the `Ship` class couldn't override the `fire` method and use `super.fire`.



Decorators

Alternatively there's a solution that could throttle the `fire` method and keep it as a true method that works

with `super`. That would be with *decorators*. The decorators that are in wide use today will differ from the decorators that will actually make their way into the language. At the time of this writing, there's no transpilation available for how the future decorators will work. For this reason, I've decided to exclude decorators from this book. If you want know more about them, the spec can be read here: <https://tc39.github.io/proposal-decorators>. Additionally, there's a JavaScript library called *core-decorators* that even includes a decorator for throttling a method. See it here: <https://github.com/jayphelps/core-decorators.js/tree/c4d9a654093a6c02d436e4d236f4d21e3271867d#throttle>.



Now if you bundle your code again and fire up the game, you'll be able to shoot rockets from your spaceship by pressing the spacebar. There's only one problem: when a rocket hits a comet, nothing happens!

29.4. WHEN THINGS COLLIDE

When a rocket hits a comet, you don't just want it to blow up—you want it to split into smaller comets. Add the following method to the `Comet` class:

```
multiply() {
  const x = getSpriteProperty(this, 'x');
  const y = getSpriteProperty(this, 'y');
  let size = getSpriteProperty(this, 'size');
  let speed = getSpriteProperty(this, 'speed');

  if (size < 5) {
    this.remove();
    return;
  }
  size /= 2;
  speed *= 1.5;
  const removeOnExit = size < 5;
  const r = Math.random() * Math.PI;
  const ninetyDeg = Math.PI/2;
  range(4).forEach(i => new Comet({
    x, y, size, speed, removeOnExit,
    rotation: r + ninetyDeg * i,
  }))
  this.remove();
}
```

- **1** If the comet is already smaller than 5, don't multiply anymore.
- **2** Decrease the size and increase the speed.
- **3** If the shard is smaller than 5, let it drift off into space.
- **4** You're going to multiply into 4 shards.
- **5** After adding the smaller shards, remove the original.

This method will split a comet into smaller, faster moving shards. You need to use the `range` and `getSpriteProperty` methods here, so go ahead and import those at the top of your comet file:

```
import { getSpriteProperty } from './framework/game';
import { range } from './framework/utils';
```

Now your comet can be split into multiples, but you still need a way to kick this off when a rocket hits a comet. Your game framework has built-in collision detection: you just need to specify what types of sprites you expect to collide with. Open up the `rocket.js` file and add the following property to the `Rocket` class:

```
export default class Rocket extends DriftingSprite {  
  static collidesWith = [Comet];  
  
  ...  
}
```

By setting this property, you're telling your game to invoke the `collision` method on any rocket that collides with a comet. That means you need to add the `collision` method on your `Rocket` class:

```
collision(target) {  
  target.multiply();  
  this.remove();  
}
```

- **1 collision is called with the sprite that was collided with, in this case a comet.**
- **2 Tell the comet hit by the rocket to multiply.**
- **3 Remove the rocket.**

You also need to import the `Comet` class into the `rocket` module. Add the following import to the top of the `rocket.js` file:

```
import Comet from './comet';
```

You can now build the code again and try playing. You should be able to destroy some comets. Wow, this is starting to look like a real game!

29.5. ADDING EXPLOSIONS

OK, you can destroy comets with your rockets, but what kind of rocket doesn't explode? You're going to fix that next. Create a new file called `explosion.js` and add the following code:

```
import Sprite from './framework/sprite';
import { explosionShape } from './shapes';

const START_SIZE = 10;
const END_SIZE = 50;
const SIDES = 8;

const defaultOptions = {
  size: START_SIZE
}

export default class Explosion extends Sprite {
  1
  constructor(options) {
    super(Object.assign({}, defaultOptions, options));
    this.shape = explosionShape(this.size, SIDES);
  }

  next() {
    this.size = this.size * 1.1;
    2
    this.shape = explosionShape(this.size, SIDES);
    3
    this.fill = `rgba(255, 100, 0, ${((END_SIZE -
this.size) * .005)})`
    4
    this.stroke = `rgba(255, 0, 0, ${((END_SIZE - this.size) * .1)})`
    4
    if (this.size > END_SIZE) {
    5
      this.remove();
    }
    return this;
  }
}
```

- **1 Explosions don't drift so you need to extend the base sprite class.**
- **2 You want the explosion to grow in size on every frame.**
- **3 After you increase the size, you need to recalculate the shape.**

- **4 You want the explosion to fade in as it reaches its end size.**
- **5 Once the explosion has reached its final size, remove it.**

There isn't a lot new going on here. This is similar to the `Comet` class in the sense that you're setting default options and using a dynamic shape. But you change the shape of the explosion at every frame. You're also changing the size and opacity. Once the size of the explosion increases to its final size, you remove it.

Now you just need to create an explosion every time a rocket collides. Import the explosion class, as well as the `getSpriteProperty` helper, into the rocket module:

```
import { getSpriteProperty } from './framework/game';
import Explosion from './explosion';
```

Now inside the `Rocket` class's `collision` method, create an explosion:

```
new Explosion({
  x: getSpriteProperty(this, 'x'),
  y: getSpriteProperty(this, 'y'),
});
```

Now your rockets explode when they strike a comet. (Don't forget to bundle your code again before testing!) There are several things you could still add to this game to make it better, but you need to wrap up. But there's still one glaring missing piece. When a comet hits your ship, something needs to happen.

In the `ship.js` file, import the `Comet` and `Explosion` classes as well as the `stop` function from the game framework:

```
import { getSpriteProperty, stop } from './framework/game';
1
import Comet from './comet';
```

```
import Explosion from './explosion';
```

- **1 Add the stop function to the existing import.**

Now set the Ship as colliding with the Comet and add the following collision method:

```
export default class Ship extends ControllableSprite {  
  ...  
  
  static collidesWith = [Comet];  
  
  collision() {  
  
    new Explosion({  
      x: getSpriteProperty(this, 'x'),  
      y: getSpriteProperty(this, 'y'),  
    });  
  
    this.remove();  
  
    setTimeout(stop, 500);  
  }  
  
  ...  
}
```

Now when a comet strikes the ship, there will be an explosion and the game will come to an end.

SUMMARY

In this capstone you created a game using classes. You extended base classes that were provided to you from a framework. When extending these base classes, you made use of `super` in a constructor as well as when overriding methods. You used static and instance properties, and we discussed why a function set as an instance property can't use `super` when overrode.

I hope you had as much fun building this game as I did. There's much more you can do to take this game further. You can add multiple lives, or a button to restart the game, once defeated. Don't be afraid to make changes to the framework to enhance this game either. You can even make UFOs appear and shoot at your ship; I included a `ufo` shape in the `shapes.js` module if you decide to give it a shot!

Unit 7. Working asynchronously

JavaScript has always been an asynchronous language. That makes it a perfect language for creating rich applications, because it can handle many concurrent tasks without locking the interface. But until recently, there hasn't been much first-class support for its asynchronous nature other than being able to pass around functions that can be invoked at a later time. That has all changed. JavaScript now has first-class support for promises, asynchronous functions, and soon, for observables.

Promises are objects that represent *future* values. They are more convenient to pass around than callbacks because, unlike callbacks, they don't rely on timing. A promise will give a value right away if the value has already been retrieved, or can wait until the value is retrieved. With callbacks, if you are too late and the value has already been retrieved, your callback may not get invoked.

Promises, as we'll soon see, also provide tools to make complex asynchronous calls easier and remove the need for callback hell.

It can be hard for people to wrap their heads around complex asynchronous interactions, which can lead to bugs. Blocking actions are much easier to think about, but are less efficient. Promises make complex asynchronous code easier to manage, but you still have to think asynchronously. But not with `async` functions. Async functions allow you to write your code as if it were a blocking operation.

A promise yields one value, and then it's over. An observable is like a promise that continues to yield values. Observables allow you to turn anything into an event that you can subscribe to, and they allow you to treat events like objects that you can apply higher-order functions to such as `map` and `reduce`.

Lesson 30. Promises

After reading lesson 30, you will be able to

- Use promise-based libraries to fetch asynchronous data
- Do basic error handling for promises
- Memoize asynchronous calls using `Promise.resolve`
- Combine several promises into one

A *promise* is an object that represents an eventual value. You can access this eventual or future value by calling `.then()` on the promise and supplying a callback function. The promise will eventually invoke this callback with the value. If the promise is still waiting for the value (the promise is in a *pending* state), then the promise will wait until the value is ready or has loaded (at which point the promise enters the *resolved* state) before invoking the callback with the value. If the promise has already resolved, then the callback will be invoked right away.^[1]

1

Not immediately. The callback will be added to the event loop, similar to a `setTimeout` with a delay of 0.



Consider this

Asynchronous values in JavaScript were traditionally accessed by passing around callback functions that would get invoked once the data was ready. This is limiting because only the object that has a reference to the callback gets notified when the value is ready. What if instead of passing around a callback, you could pass around a value that represents what the asynchronous value will eventually become? How would this work?

--

30.1. USING PROMISES

Let's assume you're using a library called `axios` that makes an AJAX request and returns a promise. You want to use it to load data from the GitHub API and list a specific user's organizations. You could make the request like so:

```
axios.get('https://api.github.com/users/jisaacks/orgs')      1
  .then(resp => {                                           2
    const orgs = resp.data;
    // do something with array of orgs
  });
```

- **1 The `axios.get` function makes an AJAX request and returns a promise.**
- **2 You're calling the `then` function on the promise with an arrow function. The arrow function will get invoked with the `resp` after the request completes.**

Here you call `axios.get`, which returns a promise. When that promise resolves, you get the data from the response object. You can then do something with the results.

Let's look at what this might look like if `axios` used callbacks instead of promises:

```
axios.get('https://api.github.com/users/jisaacks/orgs', resp =>
{
  const orgs = resp.data;
  // do something with array of orgs
});
```

In this simple example, it seems like using promises just adds more steps than using a callback. However, as our examples become more complex, you'll see how promises make for far more flexible and readable code.

The function has my username hard-coded. Rewrite it to

load a specified user's organizations:

```
function getOrgs(username) {  
  return  
  axios.get(`https://api.github.com/users/${username}/orgs`);  
}  
  
getOrgs('jisaacks').then(data => {  
  const orgs = resp.data;  
  // do something with array of orgs  
});
```

Here you made a function that takes a username and returns a promise. You then used that function to fetch the organizations for my user. Again let's look at what this might look like using traditional callbacks:

```
function getOrgs(username, callback) {  
  return  
  axios.get(`https://api.github.com/users/${username}/orgs`,  
  callback  
);  
}  
  
getOrgs('jisaacks', data => {  
  const orgs = resp.data;  
  // do something with array of orgs  
});
```

In this fictitious function, the `getOrgs` function has gotten more complex because it needs to accept a callback function so it can pass it to the AJAX library.

Now let's imagine you have a class that represents a user view. In this user view, you don't immediately show the user's organizations, but, to make the application faster, you want to start loading them right away, in anticipation of showing them later:

```
class UserView {  
  constructor(user) {  
    this.user = user;  
    this.orgs = getOrgs(user.username);    1  
  }  
  
  defaultView() {
```

```

    // show the default view
  }

  orgView() {
    // show loading screen
    this.orgs.then(resp => {
      const orgs = resp.orgs;
      // show the orgs
    })
  }
}

```

- **1 Start loading the user orgs right away.**
- **2 Show the orgs once loaded.**

Here you created a class called `UserView` that starts loading the user's organizations immediately. Later when the user view needs to show the user's organizations, if they're already loaded or if they're still being loaded, it won't matter. Either way, it will wait until they finish loading and then show them—or if they've already loaded, it will show them right away. This would be much harder to achieve using traditional callbacks.

Quick check 30.1

Q1:

The following example is using an AJAX function that takes a callback. Assuming the AJAX function returns a promise, rewrite it to use a promise instead:

```

function handleData(data) {
  // do something with data
}

ajax('example.com', handleData);

```

QC 30.1 answer

A1:

```
function handleData(data) {  
  // do something with data  
}  
  
ajax('example.com').then(handleData);
```

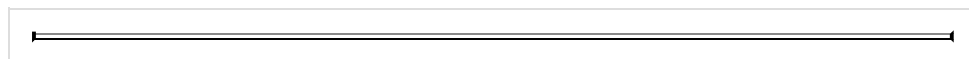
30.2. ERROR HANDLING

We don't live in a perfect world. Sometimes the data that the promise is retrieving fails, either from a network error or from some other issue. The `then` method on a promise takes an optional second callback that gets invoked with the error if an error occurs:

```
function handleResp(resp) {  
  // handle successful response  
}  
  
function handleError(err) {  
  // handle an error  
}  
  
const url = 'https://api.github.com/users/jisaacks/orgs';  
axios.get(url).then(handleResp, handleError);
```

If everything works as expected here, the `handleResp` function will get called with the response object from the request. If something goes wrong, the `handleError` function will get called with the error.

These are the basics of error handling with promises, but we'll touch on some different techniques and gotchas about error handling with promises in [lesson 31](#) on advanced promises.



Quick check 30.2

Q1:

The following example uses a function called `getJSON` that takes a function that accepts two arguments. The first is a possible error object; the second is the JSON data that was loaded. Usually only one value will be present. Rewrite this to be

promise-based instead of callback-based:

```
getJSON('example.com/data.json', (err, json) => {
  if (err) {
    // handle error
  } else {
    // handle json
  }
});
```

QC 30.2 answer

A1:

```
getJSON('example.com/data.json').then(
  (json) => {
    // handle json
  },
  (err) => {
    // handle error
  }
);
```


30.3. PROMISE HELPERS

The `Promise` object has four helper methods to make working with promises much easier:

1. `Promise.resolve()`
2. `Promise.reject()`
3. `Promise.all()`
4. `Promise.race()`

Let's start with `Promise.resolve()`. This function returns a promise that has already resolved. Any argument given to `Promise.resolve` will be returned from the promise via a parameter passed to a function supplied to `.then`. This can be helpful for all sorts of reasons, and is typically used when something is expecting a promise but you already have a value available. Consider a scenario where you're loading transactions for an accounting program. Whenever you load a transaction, you want to cache it so that next time it's requested, you don't have to load it again. You may end up with some code like this:

```
const transactions = {};  
  
getTransaction(id, cb) {  
  if (transactions[id]) {  
    cb(transaction[id]);  
  } else {  
    load(`/transactions/${id}`).then(transaction => {  
      transactions[id] = transaction;  
      cb(transaction);  
    })  
  }  
}  
  
getTransaction(405, transaction => {  
  // do something with transaction  
})
```

- **1** If you already have the data, invoke the callback right

away.

- **2 Assuming a fictitious load function that fetches data and returns it directly**
- **3 If you loaded the data, invoke the callback once you have it.**

Here you're using an imaginary load function that fetches data from a URL and returns it directly via a promise. The `getTransaction` function effectively memoizes the network's requests so that each transaction only needs to be loaded once. But you've had to resort back to callbacks, and lost many of the benefits of promises. For example, how would you add error handling to this? Rewrite `getTransaction` to always return a promise with the help of `Promise.resolve`:

```
const transactions = {};  
  
getTransaction(id) {  
  if (transactions[id]) {  
    return Promise.resolve(transactions[id]);          1  
  } else {  
    const loading = load(`/transactions/${id}`);      2  
    loading.then(transaction => {  
      transactions[id] = transaction;                 3  
    });  
    return loading;                                   4  
  }  
}  
  
getTransaction(405).then(transaction => {  
  // do something with transaction  
}, err => {                                           5  
  // handle error  
})
```

- **1 If you already have the data, use `promise.resolve` to return it.**
- **2 Create a promise to load the transaction.**
- **3 Internally add the transaction to your cache when the promise resolves.**
- **4 Return the promise.**

- **5 You can now easily add error handling.**

Your `getTransaction` function now always returns a promise. Because a promise is already being returned, you can easily add error handling outside of the `getTransaction` function without the `getTransaction` function having to concern itself with any error handling.

When invoking the `getTransaction` function, if the value is already in your cache, you return it using `Promise.resolve`, which creates an already resolved promise that gives the transaction. If you need to load the transaction, first create the promise loading the transaction. Internally you add the new transaction to your cache once loaded. You also return the promise. This shows another thing you can do with promises—did you catch it? You can call `.then` on a promise as many times as you like. Here you called `.then` on your loading promise internally to add the transaction to your cache. You also called `.then` on the same promise externally to actually do something with the transaction.

`Promise.reject()` works the same way as `Promise.resolve`, only the resulting promise is automatically in a *rejected* state instead of a *resolved* state. You can also pass a value to `Promise.reject` that will be given to the handler:

```
Promise.reject('my value').then(null, reason => {  
  console.log('Rejected with:', reason);  
});
```

1

- **1 Rejected with: my value**

Sometimes you need to fetch data from several different locations and wait for all of it to load before doing

anything. Let's consider a new website that has profile pages for authors. The profile page shows an author's information and lists the articles that they have written. You want to show a loading screen until both are loaded. You can do so using `Promise.all` like this:

```
function getAuthorAndArticles(authorId) {  
  const authorReq = load(`/authors/${authorId}/details`)  
  const articlesReq = load(`/authors/${authorId}/articles`)  
  
  return Promise.all([authorReq, articlesReq])  
1  
}  
  
getAuthorAndArticles(37).then( ([author, articles]) => {  
2  
  // author and articles have loaded  
});
```

- **1 `Promise.all` takes an array of promises and returns a single promise.**
- **2 The new promise gives an array of all the values from the previous promises.**

Here you create promises for loading both the author and the author's articles. You give an array of these promises to `Promise.all`, which returns a single promise. The new promise will wait until all the promises have resolved and will give an array of all the values given from the previous promises. If any of the promises fail, the new promise will also fail with the error given from the first promise that failed.

`Promise.race` works similarly to `Promise.all`: it takes an array of promises and returns a new promise. But this promise resolves as soon as the first promise it was given resolves, and gives its value.



Quick check 30.3

Q1:

In the following example, what number will be logged?

```
Promise.all([
  Promise.resolve(1),
  Promise.reject(2),
  Promise.resolve(3),
  Promise.resolve(4)
]).then(
  num => console.log(num),
  num => console.log(num)
)
```

QC 30.3 answer

A1:

2

SUMMARY

In this lesson, you learned the basics of how to use promises.

- A promise is an object that represents an eventual or future value.
- The value is accessed by passing a callback to `.then` on the promise.
- An optional second callback can be given to `.then` to handle if the promise is rejected.
- `.then` can be called on the promise any number of times.
- If the promise is pending, `.then` will wait until the promise resolves.
- If the promise is resolved, `.then` will still yield a value.
- `Promise.resolve` creates a pre-resolved promise.
- `Promise.reject` create a pre-rejected promise.
- `Promise.all` takes an array of promises and returns a new promise that waits for all of them to resolve.
- `Promise.race` takes an array of promises and returns a new promise that only waits for the first to resolve.

Let's see if you got this:

Q30.1

Let's say you have three endpoints:

1. `/user/4XJ/credit_availability`
2. `/transunion/credit_score?user=4XJ`
3. `/equifax/credit_score?user=4XJ`

The first endpoint checks the credit available on file for the user. The other two endpoints check the user's latest credit score from TransUnion and EquiFax. To render the page, you need to wait for the credit availability and at least one credit score. Use any promise-based AJAX library and create a new

promise that combines `Promise.all` and `Promise.race` to achieve this.

Lesson 31. Advanced promises

After reading lesson 31, you will be able to

- Create your own promises
- Wrap callback-based methods with promises
- Use and understand how to properly write nested promises
- Understand how error handling propagates in promise chains

In the previous lesson you learned the basics of promises. We'll now take a more advanced look at creating new promises and converting asynchronous code to use promises. We'll look at advanced error handling, using promises for multiple asynchronous calls, and other advanced usages.

Consider this

Sometimes you need to make multiple asynchronous calls to get the data you need. A piece of data relies on another piece of data, which relies on yet another piece of data. Yet all of these pieces of data must be fetched from different locations. Traditionally you would have to treat these as three different operations with three different error catchers. What if you could turn it into one single operation with one error catcher?

31.1. CREATING PROMISES

A promise is created by instantiating a new `Promise` object with a function argument, `new Promise(fn)`. The function given to the promise should take two arguments itself. The first one is a function to *resolve* the promise, and the second is a function to *reject* the promise:

```
const later = new Promise((resolve, reject) => {  
  // Do something asynchronously  
  resolve('alligator');  
});  
  
later.then(response => {  
  console.log(response);  
});
```

- **1 “alligator”**

Let’s say you’re loading an image via JavaScript. The classic way of doing this is to create an `Image` object and set a `src` property with callbacks assigned to `onload` and `onerror` like so:

```
const img = new Image();  
img.onload = () => // Do something when image loads  
img.onerror = () => // Do something when image fails  
img.src = 'https://www.fillmurray.com/200/300';
```

If you wanted to turn the image loading into a promise, you could wrap it in a `Promise` and assign the `onload` and `onerror` to your `resolve` and `reject` functions respectively:

```
function fetchImage(src) {  
  return new Promise((res, rej) => {  
    const img = new Image();  
    img.onload = res;  
    img.onerror = rej;  
    img.src = src;  
  });  
}
```

```
}

fetchImage('https://www.fillmurray.com/200/300').then(
  () => console.log('image loaded'),
  () => console.log('image failed')
);
```

We saw in the previous lesson that the `Promise` object has some helper methods like `Promise.resolve` and `Promise.reject`, but what if you wanted the promise to wait five seconds before resolving? This would be a handy promise-based version of `setTimeout`. You can actually wrap `setTimeout` with a promise to create such a tool:

```
function wait(milliseconds) {
  return new Promise((resolve) => {
    setTimeout(resolve, milliseconds);
  });
}

wait(5000).then(() => {
  // 5 seconds later...
});
```

You can turn any callback-based function into a promise using this technique. Let's say, for example, you're using the `navigator.geolocation.getCurrentPosition` function, which is callback-based. It takes a success callback and an error callback like so:

```
navigator.geolocation.getCurrentPosition(
  location => {
    // do something with user's geo location
  },
  error => {
    // Handle error
  }
)
```

You can wrap this in a promise like so:

```
function getGeoPosition(options) {
```

```

    return new Promise((resolve, reject) => {
      navigator.geolocation.getCurrentPosition(resolve, reject,
options);
    });
  }

  getGeoPosition().then(
    () => // do something with user's geo location
  )

```

If it takes a callback, you can wrap it in a promise!

Quick check 31.1

Q1:

How could you wrap this `registerUser` function in a promise?

```

registerUser(userData, (error, user) => {
  if (error) {
    // handle error
  } else {
    // do something with new user
  }
});

```

QC 31.1 answer

A1:

```

function registerUserAsync(userData) {
  return new Promise(resolve, reject) {
    registerUser(userData, (error, user) => {
      if (error) {
        reject(error);
      } else {
        resolve(user);
      }
    });
  }
}

```

--

31.2. NESTED PROMISES

In the previous lesson about promises, we went over how to use an AJAX library that returns a promise. But you could, instead, use the new Web Hypertext Application Technology Working Group (WHATWG) standard method `fetch` for making AJAX requests. To do so you would need to use multiple promises:

```
fetch("https://api.github.com/users/jisaacks/orgs")
  .then(resp => {
    resp.json().then(results => {
      // do something with results
    });
  });
//
```

This is starting to look like callback hell, but this isn't actually how promises are meant to be used. Every time you call `then()` on a promise, it returns a new promise. This makes promises chainable:

```
fetch("https://api.github.com/users/jisaacks/orgs")
  .then(resp => resp.json())
  .then(results => {
    // do something with results
  });
```

Notice how your `thens` are no longer nested within each other, and are instead being chained together.

When you invoke `then()` on a promise, you pass a function as a parameter. Whatever that function returns will be the value for the next `then()` in the chain:

```
Promise.resolve(1)
  .then(number => number + 1)      1
  .then(number => number + 1)      2
  .then(number => number + 1)      3
  .then(number => {
    console.log(number)           4
  });
```

- **1 1 + 1 = 2**
- **2 2 + 1 = 3**
- **3 3 + 1 = 4**
- **4 4**

There's one exception to this: if the function given to a `then()` returns a promise, the promise returned by `then()` will wait until that promise resolves or rejects a value, and will do the same:

```
const time = new Date().getTime();
const logTime = () => {
  const seconds = (new Date().getTime() - time);
  console.log(seconds/1000, 'have elapsed.');
```

}	
wait(5000)	
.then(() => {	
logTime();	1
return wait(5000);	
})	
.then(() => {	
logTime();	2
return wait(2000);	
})	
.then(() => {	
logTime();	3
});	

- **1 “have elapsed.”**
- **2 “have elapsed.”**
- **3 “have elapsed.”**

Here, the first promise specifies to wait 5 seconds. When it resolves, you log the elapsed time. You get 5.004, but that's only because the `setTimeout` you're using under the hood doesn't guarantee exact timing. That function returns another promise set to wait another 5 seconds. So the next `then()` in the chain resolves after a total of ~10 seconds. You then return yet another promise that waits 2 seconds, causing the final `then()` to resolve

after a total of ~12 seconds.

It's important to remember this only works if the function inside the `then` returns the promise. A promise inside the `then` that isn't returned won't work:

```
wait(5000)
  .then(() => {
    logTime();           1
    wait(5000);
  })
  .then(() => {
    logTime();           2
  });
```

- **1 “have elapsed.”**
- **2 “have elapsed.”**

This time around, the second `wait` wasn't returned, so the next `then()` in the chain didn't wait for it.

Quick check 31.2

Q1:

In the following made-up set of functions, each promise is nested. How could you rewrite this to use chaining instead?

```
getUserPosition().then(position => {
  getDestination(position).then(destination => {
    calculateRoute(destination => {
      // render map
    });
  });
});
```

QC 31.2 answer

A1:

```
getUserPosition()  
  .then(position => getDestination(position))  
  .then(destination => {  
    // render map  
  });
```



31.3. CATCHING ERRORS

In the previous section we saw that promises can be chained together using multiple `then()`s to create more complex promises. You also recently learned that `then()` takes a second callback to handle any errors. This raises the question: “If I chain multiple `thens`, do I need to have multiple error handlers?” The answer is, you can, but you don’t need to. Whenever a promise rejects, it will bubble up through any promises on it until it finds the first error handler:

```
Promise.resolve()
  .then(() => console.log('A'))
  .then(() => Promise.reject('X'))
  .then(() => console.log('B'))
  .then(null, err => console.log('caught:', err))
  .then(() => console.log('C'));
```

The previous example will log the following:

- A
- caught: X
- C

Notice that it didn’t log B, which occurred after the rejection but before the error handler. But it did log C, which occurred after the error handler. This is because when a promise rejects, none of the promises after it will resolve until the rejection is caught. Once caught, any promise later in the chain will be able to resolve.

Notice how you used `null` as the first argument because you only cared about catching the error at that step. There’s a convenience method `catch` for this very purpose. The previous example could have been written like so:

```
Promise.resolve()
  .then(() => console.log('A'))
  .then(() => Promise.reject('X'))
  .then(() => console.log('B'))
  .catch(err => console.log('caught:', err))
  .then(() => console.log('C'));
```

You don't have to use `Promise.reject` or explicitly call `reject()` inside of a promise to cause it to reject. Any JavaScript error will cause a rejection:

```
Promise.resolve()
  .then(() => { throw "My Error"; })
  .catch(err => console.log('caught:', err));    1
```

- **1 caught: My Error**

Let's take a look at another example:

```
ajax('/my-data.json')
  .then(
    res => {
      throw "Some Error";
    },
    err => {
      console.log('First catcher: ', err);    1
    }
  )
  .catch(
    err => {
      console.log('Second catcher:', err);    2
    }
  );
```

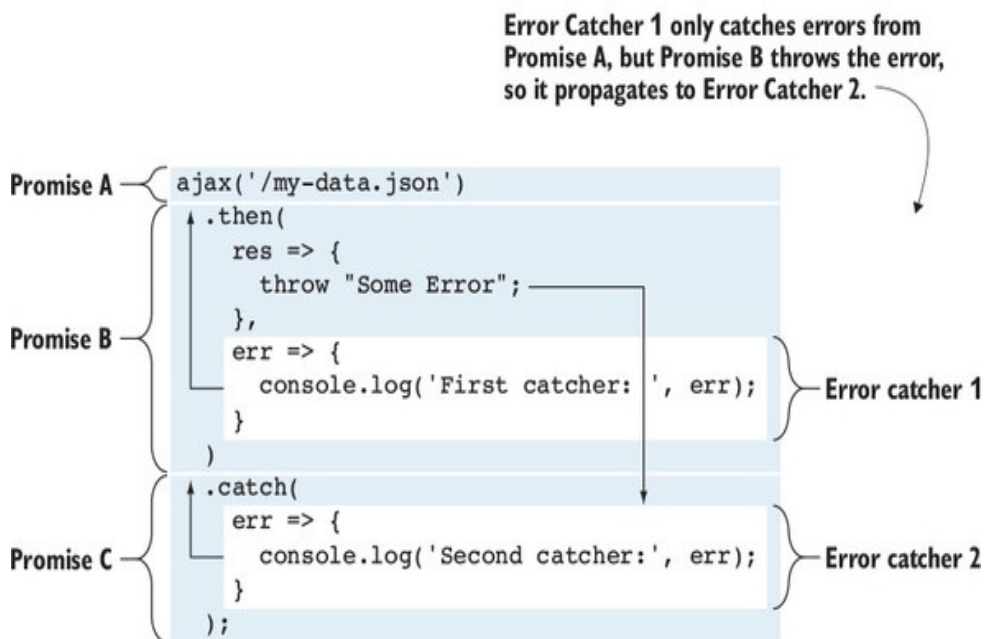
- **1 This won't catch the error.**
- **2 This will catch the error.**

When you give two functions to a `.then()`, the first function runs in the event of the promise resolving, and the second function runs in the event of the promise rejecting. But if there's an error in the first function, the second function won't catch it. It will bubble up to the next promise that's returned from the `.then()` and cause that promise to reject. So in order to catch the

error, you must catch it from that promise. See [figure 31.1](#).

Figure 31.1. Promise error handling

This example contains three promises and two error catchers.



Because every call to `.then` or `.catch` returns a new promise, it means there's always a promise at the end that could potentially go uncaught. Initially, this was a cause for concern because if you didn't catch an error from a promise, the promise *swallowed* the error and it went unreported. But most environments today raise an unhandled promise rejection error.

Quick check 31.3

Q1:

In the following example, if `handlerOne` throws an error, which catcher will catch the error? What if `handlerTwo` throws the error?

```
myPromise()
  .then(handlerOne, catcherOne)
  .then(handlerTwo, catcherTwo)
```

QC 31.3 answer

A1:

catcherTwo will catch from handlerOne.

handlerTwo would cause an unhandled promise rejection.

SUMMARY

In this lesson, you learned about the most advanced usages of promises.

- You can create a new promise with a callback that accepts `resolve` and `reject` functions.
- The `.catch(errCatcher)` function is shorthand for `.then(null, errCatcher)`.
- Every call to `.then()` or `.catch()` returns a new promise creating a promise chain.
- Responses or rejections from previous promises bubble up through the promise chain.

Let's see if you got this:

Q31.1

The following function `getArticle` is callback-based. It also uses a function called `load`, which is also callback-based. Write a function that wraps the `load` function with a promise. Then rewrite the `getArticle` to be promise-based instead of callback-based:

```
function getArticle(id, callback) {
  load(`/articles/${id}`, (error, article) => {
    if (error) {
      callback(error);
    } else {
      load(`/authors/${article.author_id}`, (error,
author) => {
        if (error) {
          callback(error);
        } else {
          load(`/articles/${id}/comments`, (error, author)
=> {
            if (error) {
              callback(error);
            } else {
              callback(null, [article, comments, author])
            }
          });
        }
      });
    }
  });
}
```

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

getArticle(57, (error, results) => {
  if (error) {
    // show error
  } else {
    // render article
  }
});
```

Lesson 32. Async functions

After reading [lesson 32](#), you will be able to

- Write asynchronous code using a synchronous syntax
- Use generators to write asynchronous code
- Use asynchronous functions to write asynchronous code

One of the reasons JavaScript can be hard to learn is because of its asynchronous nature. Asynchronous code is hard to think about, but it's this asynchronous nature that makes JavaScript such a great language for the web. Web applications are full of asynchronous actions—from loading assets to handling user input. The purpose of asynchronous functions is to make it easy to write and think about asynchronous code.

Consider this

So far in this unit we've been dealing with promises, which have proven to be much more useful than traditional asynchronous code using callbacks. But when using promises you still have to *think* asynchronously, which is much harder to do than thinking synchronously. Languages like PHP or Ruby are synchronous, so operations that take time to complete are easier to think about, but they're *blocking*. This means they halt the execution of any code and block everything else from happening until they finish. What if you could get the best of both worlds and write code that looks as if it's blocking—making it easier to think about—but still operates asynchronously?

32.1. ASYNCHRONOUS CODE WITH GENERATORS

When we discussed generator functions in [lesson 18](#), you learned how you could create pause points by yielding a value. Whenever you `yield` a value from within a generator, it pauses at that point in the code until `.next()` is called on the generator again. As a reminder, when you call `next()` on the generator, you're able to pass a value to the generator and get back a value and a `done` property indicating what the generator has completed.

What if the generator were to `yield` a promise? And what if the code that was running that generator waited for that promise to resolve and then invoked the `next()` function, passing the value resolved from the promise back to the generator? What if you did this in a loop, allowing the generator to yield several promises back to back?

Write a function that takes a generator function and does this. Don't dwell on understanding this runner function. What you do with it is more important than its inner workings:

```
const runner = gen => (...args) => {           1
  const generator = gen(...args);              2
  return new Promise((resolve, reject) => {      3
    const run = prev => {                       4
      const { value, done } = generator.next(prev); 5
      if (done) {                             6
        resolve(value);
      } else if (value instanceof Promise) {      7
        value.then(run, reject);
      } else {                                  8
        run(value);
      }
    }
  })
  run();                                       9
```



```
});  
}
```

- **1 Take a generator function with an argument and return a new function.**
- **2 Invoke the generator function with the same arguments.**
- **3 Return a new promise.**
- **4 Create a run function.**
- **5 Pass any previous value to the generator, while getting the next value from it.**
- **6 If the generator is complete, resolve the promise with the final value.**
- **7 If the value is a promise, tell it to call run again when it resolves.**
- **8 Otherwise call run again immediately.**
- **9 Start the loop by invoking run.**

Here you have a function that does exactly what we just discussed. You can give it a generator function and get back a new function that can `yield` promises, and get back their values once resolved. This runner function is pretty complex, so don't worry if you find it hard to follow. The point is how you can use this runner function.

Here's an example that uses `fetch` to load an image which requires three promises:

```
fetch('my-image.png')  
  .then(resp => resp.blob())  
  .then(blob => createImageBitmap(blob))  
  .then(image => {  
    // do something with image  
  });
```

If you were to use a generator with your runner function, you could have written it like this:

```
const fetchImageAsync = runner(function* fetchImage(url) {
```

```
const resp = yield fetch(url);
const blob = yield resp.blob();
const image = yield createImageBitmap(blob);
// do something with image
});

fetchImageAsync('my-image.png');
```

Or if you wanted to make this function more universal, you could have done something more like this:

```
const fetchImageAsync = runner(function* fetchImage(url) {
  const resp = yield fetch(url);
  const blob = yield resp.blob();
  const image = yield createImageBitmap(blob);

  return image;
});

fetchImageAsync('my-image.png').then(image => {
  // do something with image
});
```

By using the runner function to wrap your generator, every time you `yield` a promise, you get back the value from that promise. This allows you to write code that looks synchronous or blocking, but is actually still asynchronous.

At the time of this writing, the `createImageBitmap` function that you're using in these examples is only supported by the latest versions of Mozilla Firefox and Google Chrome.



Quick check 32.1

Q1:

How would you convert the following code to make use of a generator and the runner function?

```
getUserPosition()
  .then(position => getDestination(position))
  .then(destination => {
```

```
// render map  
});
```

QC 32.1 answer

A1:

```
const renderMap = runner(function* renderMap() {  
  const position = yield getUserPosition();  
  const destination = yield getDestination(position);  
  // render map  
});  
  
renderMap();
```

32.2. ASYNC FUNCTIONS

The awesome thing about learning how to write asynchronous code using generators, promises, and a runner function is that you don't have to learn anything new to make the jump to asynchronous functions. That's because async functions are just syntactic sugar for the generators + promises approach. All you really need to learn are the new keywords. To indicate that a function is an async function, you prefix it with the `async` keyword. Then, instead of using the keyword `yield`, you use the keyword `await`. That's all!

The previous `fetchImage` function could be rewritten using an async function:

```
async function fetchImage(url) {  
  const resp = await fetch(url);  
  const blob = await resp.blob();  
  const image = await createImageBitmap(blob);  
  
  return image;  
};  
  
fetchImage('my-image.png').then(image => {  
  // do something with image  
});
```

Notice how little you changed. You aren't using a generator anymore, and you don't need the runner function anymore, either. You just need to indicate that your function is async, and then you can `await` promises instead of `yielding` them.

An async function will always return a promise. If the async function itself returns a value, the promise returned will resolve to that value. If the async function returns a promise, the promise returned from the async

function will resolve that promise's value. This means you could have simplified your `fetchImage` function like so:

```
async function fetchImage(url) {
  const resp = await fetch(url);
  const blob = await resp.blob();
  return createImageBitmap(blob);
};

fetchImage('my-image.png').then(image => {
  // do something with image
});
```

Inside the `async` function, when you `await` a promise, it will wait for the promise to resolve then return the promise's value.

Quick check 32.2

Q1:

How would you convert the following code to make use of an `async` function?

```
getUserPosition()
  .then(position => getDestination(position))
  .then(destination => {
    // render map
  });
```

QC 32.2 answer

A1:

```
async function renderMap() {
  const position = await getUserPosition();
  const destination = await getDestination(position);
  // render map
};

renderMap();
```

32.3. ERROR HANDLING IN ASYNC FUNCTIONS

If any of the promises that you `await` reject, you can handle the rejection from outside async function like so:

```
async function fail() {
  const msg = await Promise.reject('I failed.');
```

return 'I Succeeded.';

```
};

fail().then(msg => console.log(msg), msg => console.log(msg))
1
```

- 1 “I Failed.”

I’ve seen many people use try-catch statements inside async functions. This will work if an actual error is thrown. Remember, however, that an error will cause a promise to reject, but you can also manually reject a promise. In the latter case, a try-catch statement won’t catch the rejection:

```
async function tryTest() {
  try {
    return Promise.reject('My Rejection Msg');
```

} catch(err) {

return Promise.resolve(`caught: \${err}`);

}

```
}
```

tryTest().then(response => {

console.log('response:', response);

});

1

- 1 Uncaught (in promise) My Rejection Msg

In this example, the try-catch statement didn’t catch the `Promise.reject` and caused your `tryTest` function to throw an `uncaught in promise` error.

You could write your own promise wrapper to make error handling in async functions easier. You just need a

function that takes a promise and returns a new promise that always resolves an array. If successful, the first value will be the result; if the promise is rejected, the second value will be the rejection:

```
function rescue(promise) {  
  return promise.then(  
    res => [res, null],  
    err => [null, err]  
  );  
}
```

With this little helper function you could now write your `tryTest` function like so:

```
async function tryTest() {  
  const [res, err] = await rescue(Promise.reject('err err'));  
  if (err) {  
    return `caught: ${err}`  
  } else {  
    return res;  
  }  
}  
  
tryTest().then(response => {  
  console.log('response:', response);  
});
```

1

- **1 response: caught: err err**

Let's imagine you're building a photo gallery and you want to fetch all the images using your `fetchImage` function. You need to wait until all of them have loaded before rendering the gallery. You could use `Promise.all` for this, but that would mean that if any of the images fail to load (highly possible), the whole thing would reject. Instead, you want any failed images to be skipped and to just get the result of all the images that loaded successfully. Use an async function to achieve this:

```
async function allResolved(promises) {  
  const resolved = [];
```

1


```

const handlers = promises.map(promise => (
  promise
    .then(resp => resolved.push(resp))           2
    .catch(() => { /* skip rejects */ })         3
));
await Promise.all(handlers);                     4
return resolved;                                5
}

```

- **1 An array to hold all of the resolved values**
- **2 When the promise resolves, add the response to the array.**
- **3 If the promise rejects, do nothing.**
- **4 Wait for all the promises to resolve (or reject).**
- **5 Return all the resolved values.**

Here you used an async function to take an array of promises, get all of their responses, and skip all of their rejections. You could use it for your photo gallery like so:

```

const sources = ['image1.png', 'image2.png', ...];
allResolved(sources.map(fetchImage)).then(images => {
  // build photo gallery with loaded images
});

```

Anytime you're working with multiple promises, consider whether using an async function could simplify things.

Quick check 32.3

Q1:

If a promise that's being awaited in an async function rejects, what happens?

QC 32.3 answer

A1:

If uncaught, the promise returned from the async function will reject.

SUMMARY

In this lesson, you learned how async functions work and how to use them.

- An async function is syntactic sugar for a generator and a runner function.
- An async function is declared by the keyword `async` before the keyword `function`.
- Inside of an async function, you can `await` a promise to get the value it resolves to.
- An async function always returns a promise.
- The value returned from the async function will be resolved from the promise that the async function returns.
- If a promise inside the async function rejects, the promise the async function returns will reject.

Let's see if you got this:

Q32.1

Here's my solution to the previous lesson's exercise. This could be greatly simplified by converting it to an async function. Make it happen:

```
function getArticle(id) {
  return Promise.all([
    loadAsync(`/articles/${id}`),
    loadAsync(`/articles/${id}/comments`)
  ]).then(([article, comments]) => Promise.all([
    article,
    comments,
    loadAsync(`/authors/${article.author_id}`)
  ]));
}
```

Lesson 33. Observables

After reading lesson 33, you will be able to

- Create your own observables
- Subscribe to observables
- Compose new observables with higher-order combinator functions
- Create your own combinator functions for composing observables

Observables are objects in which you can subscribe to streams of data. Observables are like promises, but whereas a promise only resolves or rejects once, an observable can keep emitting new values indefinitely. If you were to think of promises as an asynchronous datum that you can wrap around `setTimeout`, observables would be the asynchronous data that you can wrap around `setInterval`.

Note

In order to use observables today, at the time of this writing, you need to use one of the open source implementations. Currently zen-observable is the closest implementation of the spec:

<https://github.com/zenparsing/zen-observable>.

Consider this

Websockets allow the front end to subscribe to events from the back end. With WebSockets, the server can push new values to the client as they become available. Without them, you would need the client to pull new items from the back end by continuously polling the server, asking if new data is available yet. With

WebSockets, you could say that the client is observing from the server. Wouldn't it be nice to be able to use this same push mechanism of sending new data to any observer?



33.1. CREATING OBSERVABLES

Creating an observable is similar to creating a promise, in the sense that you invoke the constructor function with a callback. The callback that you give to the observable takes an *observer* as an argument that you send data to:

```
const myObservable = new Observable(observer => {  
  // send things to observer  
});
```

An observer is just any object with the methods `start`, `next`, `error`, and `complete`, all of which are optional:

- `start`—A one-time notification when the observer subscribes to the observable
- `next`—The method that sends the data that the observable is dispatching to the observer
- `error`—Like `next()` but used for sending errors to the observer
- `complete`—A one-time notification when the observable has finished

Let's say, for example, you're going to make a clock widget that always shows the current time. You could make an observable that continuously emits the current time every second:

```
const currentTime$ = new Observable(observer => {  
  setInterval(() => {  
    const currentTime = new Date();  
    observer.next(currentTime);  
  }, 1000);  
});
```

- **1** Notifies the observer of the current time at one second intervals

You could then *subscribe* to this observable like so:

```
const currentTimeSubscription = currentTime$.subscribe({
  next(currentTime) {
    // show current time
  }
});
```

This is great, but what if you get to a point in your application where you no longer want to show the current time? Maybe the user closed the clock widget or navigated to a new section in the app that doesn't need it. Currently, there's no way to stop the `setInterval` from running.

The callback that you pass to the `Observable` constructor can return a cleanup function that gets executed when the observable is unsubscribed.

```
const currentTime$ = new Observable(observer => {
  const interval = setInterval(() => {
    const currentTime = new Date();
    observer.next(currentTime);
  }, 1000);

  return () => clearInterval(interval); 1
});
```

- **1 This cleanup function will run when the observable is unsubscribed from.**

You can now successfully unsubscribe from the `currentTime$` observable and tell it to no longer run the interval:

```
currentTimeSubscription.unsubscribe();
```

Notice how you named the observable you created with a trailing `$`. This isn't required, but it's a common convention to signify that the variable ending with a `$` is an observable. The pronunciation of such a variable would be plural: you would refer to `currentTime$` as either "current times" or "current time observable."

The Observable object also has a couple of convenience methods for creating observables: `Observable.of()` and `Observable.from()`. The former takes any number of arguments and creates an observable of those values. The latter takes an iterable and creates an observable from it:

```
Observable.from(new Set([1, 2, 3])).subscribe({
  start() {
    console.log('--- started getting values ---');
  },
  next(value) {
    console.log('==> next value', value);
  },
  complete(a) {
    console.log('--- done getting values ---');
  }
});
```

This subscription will log the following five strings:

1. – started getting values –
2. ==> next value 1
3. ==> next value 2
4. ==> next value 3
5. – done getting values –

As you can see, when you create an observable via `Observable.from(iterable)`, it's the equivalent of `Observable.of(...iterable)`.

Quick check 33.1

Q1:

How many values will be emitted from the following observables before they complete?

```
const a = Observable.from('ABC');
const b = Observable.of('ABC', 'XYZ');
```

QC 33.1 answer

A1:

1. 3 ('A', 'B', 'C')
2. 2 ('ABC', 'XYZ')

33.2. COMPOSING OBSERVABLES

One of the coolest aspects of observables is that you can compose them using higher-order combinators to create new unique observables. This means that you can treat the stream of values that the observable emits as a list that you can run methods on such as `map` and `filter`. Think of the power of `lodash`^[1] applied to events.

¹

<https://lodash.com>

You're going to use some combinators on your `currentTime$` observable. Right now, it's emitting a new date object every second:

```
-[date object]-[date object]-[date object]-[date object]-
```

You can map this stream of dates to a stream of date strings formatted to your desire:

```
currentTime$.map(date =>
  `${date.getHours()}:${date.getMinutes()}`);
```

Now you have a stream of time strings:

```
- "15:19" - "15:19" - "15:19" - "15:20" -
```

Notice how you're getting duplicate values; we would, in fact, get 60 duplicates every minute. Your clock widget wouldn't need to be notified of the next time string until it changed, so you could apply another combinator:

```
currentTime$.map(data =>
  `${date.getHours()}:${date.getMinutes()}`).distinct();
```

Now you have a stream of unique time strings:

- "15:19" --- "15:20" --- "15:21" --

Every time you apply a combinator function to an observable, you get back a new observable. The original observable is never changed.

The `Observable` spec doesn't actually include any combinator functions, but it's intended to be composed by using them. Libraries such as `zen-observable` and `RxJS`, however, include several combinator functions.

Quick check 33.2

Q1:

When you apply the `map` combinator to the `number$` observable, how is the `number$` observable modified?

```
const number$ = Observable.from(1, 2, 3, 4, 5, 6, 7, 8, 9);  
const square$ = number$.map(num => num * num);
```

QC 33.2 answer

A1:

The `number$` observable isn't modified at all. A new observable is created.

33.3. CREATING OBSERVABLE COMBINATORS

You can probably find an open source solution for any combinator you may desire, but understanding how to create your own combinators should demystify what's actually happening when you start composing them. Create a combinator for filtering values from an observable:

```
function filter (obs$, fn) { 1  
  return new Observable(observer => obs$.subscribe({ 2  
    next(val) { 3  
      if (fn(val)) observer.next(val);  
    }  
  }));  
}
```

- **1 Accept an observable and a function to test if values should be filtered.**
- **2 Return a new observable.**
- **3 Proxy values from the previous observable to the next observer if they pass the filter test.**

Here you have a function that takes an existing observable and a function to test if the values should be filtered out or not. Then you return an observable that subscribes to the existing observable. The new observable proxies values from the existing observable, but only if they pass the filtering test. You could use this function like so:

```
filter(Observable.of(1, 2, 3, 4, 5), n => n % 2).subscribe({  
  next(val) {  
    console.log('filtered: ', val);  
  }  
});
```

Here you use the `filter` function you created to filter an observable of numbers to a new observable of only

odd numbers:

```
-1-2-3-4-5-  
  
  filtered to  
  
-1--3--5-
```

Of course a more complete combinator function would also proxy `error` and `complete`. I omitted that for brevity.

Quick check 33.3

Q1:

What type of operation would the following combinator function compose observables with?

```
function myCombinator(obs$, fn) {  
  return new Observable(observer => obs$.subscribe({  
    next(val) {  
      observer.next(fn(val));  
    }  
  }));  
}
```

QC 33.3 answer

A1:

It will perform a *map* operation.

SUMMARY

In this lesson, you learned about the basics of observables.

- You create a new observable by passing the constructor a callback that takes an observer.
- An observer is an object with `start`, `next`, `error`, and `complete` methods.
- Observables can be composed to create more specific observables by applying higher-order combinators.
- A combinator is a just a function that subscribes to an existing observable and returns a new observable

Let's see if you got this:

Q33.1

Create a combinator function called `collect` that keeps track of all values that have been emitted by another observable and emits an array of all the values so far. Then create another combinator called `sum` which takes an observable that emits arrays of values and sums them. Then combine these combinators to compose a new observable over `Observable.of(1, 2, 3, 4)`. The final observable should emit the values 1, 3, 6, and 10.

Lesson 34. Capstone: Canvas image gallery

In this capstone you're going to build a canvas-based image gallery.

Your image gallery will load random images from Unsplash^[1] and render them to an HTML canvas element using a fading transition. You'll use several promises to achieve this. You'll use a promise and an async function to fetch images. You'll also use promises to orchestrate the transitions and delay between the images, as well as other wiring that will need to be done.

¹

<https://unsplash.com/license>

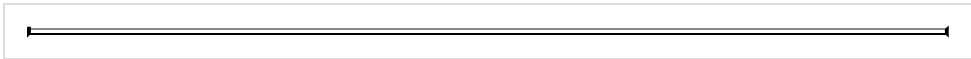


Note

You'll start your project using the start folder included in the code accompanying this book. If at any time you get stuck, you can also check out the final folder with the completed code. The start folder is a project already set up to use Babel and Browserify (see unit 0); you just need to run `npm install` to get set up. If you haven't read unit 0, you should do so before proceeding with this capstone.

There's also an included `index.html` file, which is where the app will run. It already includes all the HTML and CSS it needs, so you just need to open it in a browser once you bundle your JavaScript files. The `src` folder is where you'll put all your JavaScript files. The `dest` folder is where the bundled JavaScript file will go after you run `npm run build`. You'll need to remember to

run `npm run build` to compile your code anytime you make a change.



34.1. FETCHING IMAGES

Since you're going to be making an image gallery, start by writing a function that will fetch an image. You could use the age-old method of setting an `src` property on an `Image` object:

```
function fetchImage(src, handleLoad, handleError) {  
  const img = new Image();  
  img.onload = handleLoad;  
  img.onerror = handleError;  
  img.src = src;  
}
```

But instead use an async function and the new `fetch` API to load and create an `ImageBitmap` that you can render onto a canvas:

```
async function fetchImage(url) {  
  const resp = await fetch(url);           1  
  const blob = await resp.blob();          2  
  return createImageBitmap(blob);          3  
}
```

- **1 Wait for a promise that makes a request to the given URL.**
- **2 Wait for another promise that creates a data blob from the requested file.**
- **3 Return a final promise that creates an `ImageBitmap` from a data blob.**

Now you should be able to load an image from a URL and eventually get an `ImageBitmap` object. You don't need to worry about any callbacks for success or error states because you're using promises. Write a function that will load a random image:

```
function getRandomImage() {  
  return fetchImage('https://source.unsplash.com/random');  
}
```

This function hard-codes the URL

<https://source.unsplash.com/random> as a parameter to your previously created `fetchImage` function. The URL will be redirected to a random image from the Unsplash community of free images. That means every time you invoke this function, you'll get back a promise that will eventually yield an `ImageBitmap` for a random image.

You can test this function out now if you want to:

```
function getRandomImage().then(imgBmp => {  
  console.log('loaded:', imgBmp);  
});
```

Now that you can fetch random images, let's turn our attention to rendering them to a canvas.

34.2. PAINTING THE IMAGES ON THE CANVAS

The first thing you want to do is grab a reference to the canvas object that's provided in your HTML file:

```
const gal = document.getElementById('gallery');      1
const ctx = gal.getContext('2d');                   2
```

- **1 Get a reference to your canvas HTML element.**
- **2 Get a 2D context that you can use to start painting.**

If you happen to have peeked into the game framework code provided in the previous capstone, this code should look familiar. Whenever you're painting on an HTML canvas, you must first get either a 2D or a 3D context to paint on. Since images are two-dimensional objects, a 2D context will work just fine.

The 2D context happens to have a method called `drawImage` that takes an `ImageBitmap` and paints it to the canvas:

```
function draw(img) {                                1
  const gal = document.getElementById('gallery');
  const ctx = gal.getContext('2d');
  ctx.drawImage(img,                                2
    0, 0, img.width, img.height,
    0, 0, gal.width, gal.height
  );
}
```

- **1 Accept an `ImageBitmap` as a parameter.**
- **2 Paint the `ImageBitmap` onto the canvas.**

Notice that the `drawImage` takes an additional eight parameters after the `ImageBitmap`. These parameters are as follows:

1. Source x
2. Source y

3. Source width
4. Source height
5. Destination x
6. Destination y
7. Destination width
8. Destination height

You can use the source properties to define a section of the image to paint. And you can use the destination properties to specify a section of the canvas to paint to. You specified that you wanted to paint the entire image across the entirety of the canvas.

There's one more thing you need to add to this function before you're done. You're going to be transitioning or *fading* in new images. In order to achieve this effect, you're going to need to first paint the image using transparency or *alpha*, then over time repaint the image using slightly more alpha to create a fading-in effect. Most of the logic for figuring out how much alpha to apply will happen in another function. Your draw function just needs to be able to take an alpha property and apply it to the image you're painting:

```
function draw(img, alpha=1) {  
  const gal = document.getElementById('gallery');  
  const ctx = gal.getContext('2d');  
  ctx.globalAlpha = alpha;  
  ctx.drawImage(img,  
    0, 0, img.width, img.height,  
    0, 0, gal.width, gal.height  
  );  
}
```

The `globalAlpha` property on the canvas context will apply to anything that's getting painted, so by setting this property before drawing the image, you can effectively apply an alpha channel to the image. Cool.

Now write a function that takes an image and animates

the image fading in. You'll use your `draw` function to do the actual drawing. We just need to continue to draw a more opaque version of the image over time:

```
function fadeIn(image, alpha) {  
  draw(image, alpha);  
  if (alpha >= .99) {  
1    // we are done  
  } else {  
    alpha += (1-alpha)/24;  
2    window.requestAnimationFrame(() => fadeIn(image, alpha));  
3  }  
}
```

- **1 If the alpha is 99% or more opaque, you're finished transitioning.**
- **2 Otherwise increase the alpha.**
- **3 Then fadeIn the image with the new alpha on the next animation frame.**

This `fadeIn` function takes an image and an alpha and uses `requestAnimationFrame` to recursively increase the alpha and redraw the image. Once you're at 99% or more alpha, you're finished. But what should you do at that point? Traditionally this would be the place where you invoke a callback signifying that your animation has completed. But remember, anything that can use a callback can be made to use a promise. In fact, you can wrap this whole thing in a promise and resolve it when the animation is complete:

```
function fade(image) {  
  return new Promise(resolve => {  
    function fadeIn(alpha) {  
      draw(image, alpha);  
      if (alpha >= .99) {  
        resolve();  
      } else {  
        alpha += (1-alpha)/24;  
        window.requestAnimationFrame(() => fadeIn(alpha));  
      }  
    }  
  })  
}
```

```
    }  
  }  
  fadeIn(0.1);  
});  
}
```

6

- **1 The fade function will return a promise that resolves when the fading transition has finished.**
- **2 Define a fadeIn function that captures both the image and resolve variables in a closure.**
- **3 Draw the image at the given alpha.**
- **4 If alpha is over .99, resolve the promise.**
- **5 If alpha hasn't reached .99, increase alpha slightly and schedule another fadeIn.**
- **6 Start the fadeIn process with a low alpha.**

Here you wrapped your `fadeIn` function into another function that returns a promise. Now the `fadeIn` function resolves the promise when the animation is complete. Because the image never changes, you no longer need to pass it to the `fadeIn` function, as it will maintain the reference in a closure. You also don't need the outer `fade` function to take an `alpha` parameter, because you always want to start at `.1` and always want to end at `.99`.

You should now be able to fetch a random image and paint it to the canvas like this:

```
getRandomImage().then(fade);
```

This will fetch a random image then paint it to the canvas. The image will even fade in, not jarringly appear out of nowhere. Try it yourself!

34.3. REPEATING THE PROCESS

Now you can fetch a random image and fade it in onto the canvas, so you just need a way to orchestrate doing this over and over on a continuous loop. But you don't want to fade in an image immediately after the last; you want to wait a second to allow the user to view the image before transitioning to the next. Start by writing a function that resolves a promise after a given time period:

```
function wait(ms) {  
  return new Promise(resolve => {  
    setTimeout(resolve, ms);  
  });  
}
```

This `wait` function takes a parameter that specifies how many milliseconds to wait, and returns a promise that resolves in that amount of time. Internally you tell `setTimeout` to resolve the promise at the specified millisecond count. Now create a function called `pause` that's hard-coded to wait one and a half seconds:

```
const pause = () => wait(1500);
```

Now you can use this `pause` function like so:

```
pause().then(() => {  
  // Do something 1.5 seconds later  
});
```

More importantly, you can now create a continuous loop:

```
function nextImage() {  
  return  
  getRandomImage().then(fade).then(pause).then(nextImage);  
}
```

I love the eloquence here—the function reads exactly how it behaves. First you *get a random image*, then you *fade* that image in, then you *pause*, and then you move on to the *next image* and start the process over again.

At this point you can invoke the `nextImage` function and watch your marvelous gallery fade in random images every couple of seconds. But what happens if one of the images fails to load? In its current state, that would bring the entire thing to a halt. But you can easily fix that:

```
function start() {  
  return nextImage().catch(start);  
}
```

Do you notice what you did here? Your `start` function kicks off the whole animation by invoking `nextImage`. As long as nothing goes wrong, the `nextImage` will continue to run, loading random images and fading them in. But if something does go wrong—for example an image fails to load—you’ll catch that and restart the whole thing by calling `start` again and kicking everything off. This, of course, will be seamless to the user, as the most recently loaded image will still be displayed until the next one loads. Isn’t that neat?

Again, at this point you could simply invoke `start()` and be done. And for all intents and purposes, this would work just fine. Try it yourself. But you did create a small memory leak. Let’s take another look at that eloquent function that loops over random image animations:

```
function nextImage() {  
  return  
  getRandomImage().then(fade).then(pause).then(nextImage);  
}
```

Every time this runs, it adds to the existing chain of

promises. It has to in order to bubble back up the catch you defined in the start function. This means that your promise chain will grow longer and longer until eventually you run out of memory and crash. This would take a long time, because you're only increasing the length of the promise chain at ~2 second intervals, but you need to fix this problem nevertheless.

Instead of adding to the same promise chain over and over again, you can create a new promise chain every time. But how can this bubble back up to your catch? Simple: you'll wrap the whole thing in another promise, and if any of your internal promise chains reject, reject the outer promise:

```
function repeat(fn) {  
  return new Promise((resolve, reject) => {  
    const go = () => fn().then(() => {  
      setTimeout(go, 0);  
    }, reject);  
    go();  
  });  
}
```

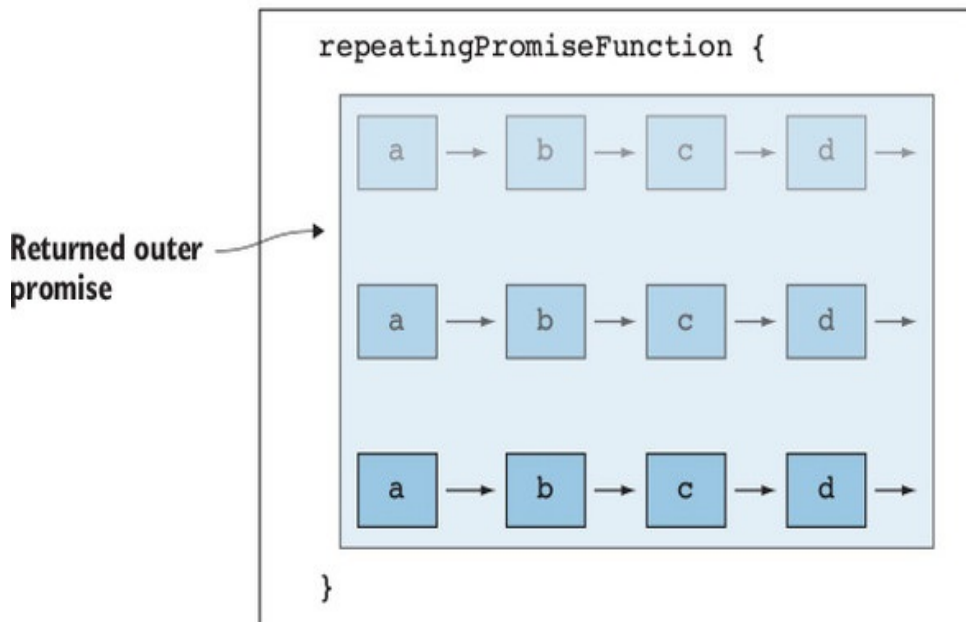
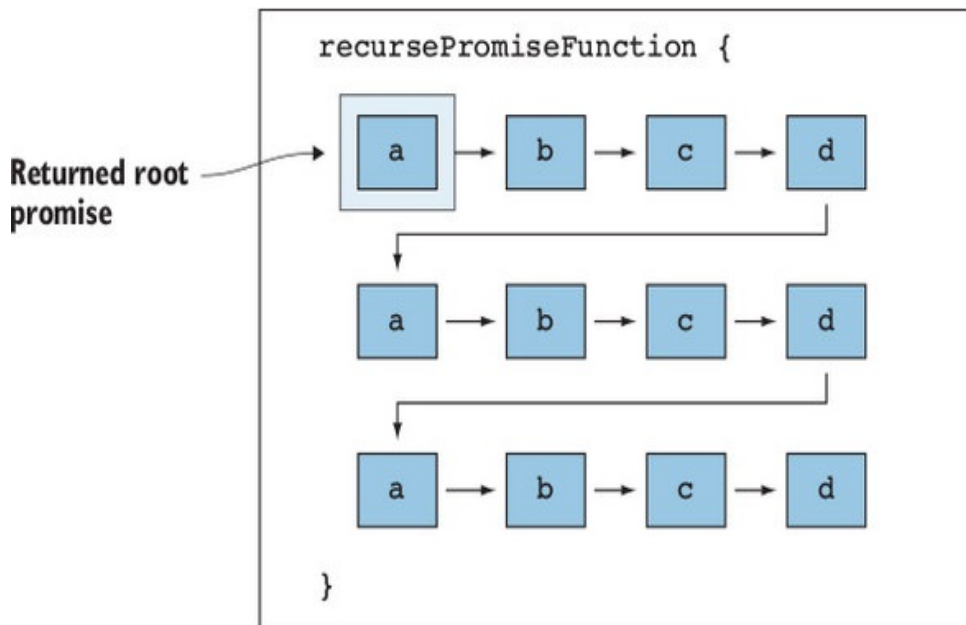
- **1 Take an initial function that will be repeating.**
- **2 Return a new promise.**
- **3 Create a new function that will invoke your passed-in function.**
- **4 When the passed-in function resolves, invoke your go function again.**
- **5 If your passed in function rejects, reject the outer promise that you returned.**
- **6 Start the process.**

This repeat function is a proxy to another function. The function it proxies is expected to return a promise, and thus the repeat function also needs to return a promise. Internally, the repeat function will continuously re-

invoke its proxied function every time it resolves. But it won't add onto a single promise chain. Instead, if any of them reject, it will manually propagate the rejection up to the promise it returned. This means no more memory leak.

Figure 34.1 shows two visualizations. The first one shows how your `nextImage` function uses recursive promises. The chain will continue to get longer and longer, because each promise is kept in memory so that the errors or resolved values will be able to bubble all the way up to the root promise that was returned. Eventually this chain will consume too much memory and crash. The second visualization shows how you can have a wrapping promise that repeats (nonrecursively) the same promise chain, but instead of linking them together, they're each linked to the outer promise (but only in the case of a failure). This means that once each inner promise resolves, there will be no need to keep them in memory because they're only connected to the outer promise on a rejection. So they won't continue to stack up and cause a memory leak.

Figure 34.1. Recursive versus repeating promises



Since the repeat function is now handling the repetition, you no longer need to make your nextImage function continuous. Modify it like this:

```
function nextImage() {  
  return getRandomImage().then(fade).then(pause);  
}
```

Now you just need to wrap it in a repeat in the start function:

```
function start() {  
  return repeat(nextImage).catch(start);  
}
```

Finally, you can call start without any caveats:

```
start();
```

You should now have an image gallery that paints random images with transitions, is self-healing, and will run forever without any memory leaks. And you achieved this without the help of any library, writing well under 100 lines of code!

SUMMARY

In this final capstone, you created an image gallery that loads random images and transitions them onto a canvas. You made use of several different promises and an async function to achieve this. By using small promises and wiring them together, you created an eloquent solution to a complex problem, using easy-to-maintain, self-documenting code.

Appendix. Exercise answers

LESSON 4

A1:

```
{
  let DEFAULT_START = 0;
  let DEFAULT_STEP = 1;

  window.mylib.range = function (start, stop, step) {
    let arr = [];

    if (!step) {
      step = DEFAULT_STEP;
    }

    if (!stop) {
      stop = start;
      start = DEFAULT_START;
    }

    if (stop < start) {
      // reverse values
      let tmp = start;
      start = stop;
      stop = tmp;
    }

    for (let i = start; i < stop; i += step) {
      arr.push(i);
    }

    return arr;
  }
}
```

LESSON 5

A1:

```
{
  const DEFAULT_START = 0;
  const DEFAULT_STEP = 1;

  window.mylib.range = function (start, stop, step) {
    const arr = [];

    if (!step) {
      step = DEFAULT_STEP;
    }

    if (!stop) {
      stop = start;
      start = DEFAULT_START;
    }

    if (stop < start) {
      // reverse values
      const tmp = start;
      start = stop;
      stop = tmp;
    }

    for (let i = start; i < stop; i += step) {
      arr.push(i);
    }

    return arr;
  }
}
```


LESSON 6

A1:

```
function
maskEmail(email, mask) {
  if (!email.includes('@')) {
    throw new Error('Invalid Email');
  }
  if (!mask) {
    mask = '*';
  }
  const atIndex = email.indexOf('@');
  const masked = mask.repeat(atIndex);
  const tld = email.substr(atIndex);

  return masked + tld;
}
```

LESSON 7

A1:

```
function
withProps() {
  let stringParts = arguments[0];
  let values = [].slice.call(arguments, 1);
  return stringParts.reduce(function(memo, nextPart) {
    let nextValue = values.shift();
    if (nextValue.constructor === Object) {
      nextValue = Object.keys(nextValue).map(function(key)
{
        return `${key}="${nextValue[key]}"`;
      }).join(' ');
    }
    return memo + String(nextValue) + nextPart;
  });
}

let props = {
  src: 'http://fillmurray.com/100/100',
  alt: 'Bill Murray'
};

let img = withProps`<img ${props}>`;

console.log(img);
// 
```

LESSON 9

A1:

```
function $(query) {  
  let nodes = document.querySelectorAll(query);  
  return {  
    css: function(prop, value) {  
      Array.from(nodes).forEach(function(node) {  
        node.style[prop] = value;  
      });  
      return this;  
    }  
  }  
}
```

LESSON 10

A1:

```
// helper function (not required but removes a lot of
// boilerplate)
function
inherit() {
  return Object.assign.apply(Object,
    [{}].concat(Array.from(arguments)))
}

const PASSENGER_VEHICLE = {
  board: function() {
    // add passengers
  },
  disembark: function() {
    // remove passengers
  }
}

const AUTOMOBILE = inherit(PASSENGER_VEHICLE, {
  drive: function() {
    // go somewhere
  }
})

const SEA_VESSEL = inherit(PASSENGER_VEHICLE, {
  float: function() {
    // go somewhere
  }
})

const AIRCRAFT = inherit(PASSENGER_VEHICLE, {
  fly: function() {
    // go somewhere
  }
})

const AMPHIBIAN_AIRCRAFT = inherit(SEA_VESSEL, AIRCRAFT, {
  // do some other stuff
})
```

LESSON 11

A1:

```
let {
  name: { first: firstName, last: lastName },
  address: { street, region, zipcode }
} = potus;

let [{
  name: firstProductName,
  price: firstProductPrice,
  images: [firstProductFirstImage]
},{
  name: secondProductName,
  price: secondProductPrice,
  images: [secondProductFirstImage]
}] = products
```

LESSON 12

A1:

```
function
createStateManager() {
  const state = {};
  const handlers = [];
  return {
    onChange(handler) {
      if (handlers.includes(handler)) {
        return;
      }
      handlers.push(handler);
      return handler;
    },
    offChange(handler) {
      if (!handlers.includes(handler)) {
        return;
      }
      handlers.splice(handlers.indexOf(handler), 1);
      return handler;
    },
    update(changes) {
      Object.assign(state, changes);
      handlers.forEach(function(cb) {
        cb(Object.assign({}, changes));
      });
    },
    getState() {
      return Object.assign({}, state);
    }
  }
}
```

LESSON 13

A1:

```
function
makeSerializableCopy(obj) {
  return Object.assign({}, obj, {
    [Symbol.toPrimitive]() {
      return Object.keys(this).map(function(key) {
        const encodedKey = encodeURIComponent(key);
        const encodedVal =
encodeURIComponent(this[key].toString());
        return `${encodedKey}=${encodedVal}`;
      }, this).join('&');
    }
  });
}

const myObj = makeSerializableCopy({
  total: 1307,
  page: 3,
  per_page: 24,
  title: 'My Stuff'
});

const url = `http://example.com?${myObj}`;
```

LESSON 15

A1:

```
function car(name, seats = 4) {
  return {
    name,
    seats,
    board(driver, ...passengers) {
      console.log(driver, 'is driving the', this.name)
      const passengersThatFit = passengers.slice(0,
this.seats - 1)
      const passengersThatDidntFit =
passengers.slice(this.seats - 1)
      if (passengersThatFit.length === 0) {
        console.log(driver, 'is alone')
      } else if (passengersThatFit.length === 1) {
        console.log(passengersThatFit[0], 'is with',
driver)
      } else {
        console.log(passengersThatFit.join(' & '), 'are
with', driver)
      }
      if (passengersThatDidntFit.length) {
        console.log(passengersThatDidntFit.join(' & '),
'got left behind')
      }
    }
  }
}
```


LESSON 16

A1:

```
function
updateMap({ zoom, center, coords = center, bounds }) {
  if (zoom) {
    _privateMapObject.setZoom(zoom);
  }
  if (coords) {
    _privateMapObject.setCenter(coords);
  }
  if (bounds) {
    _privateMapObject.setBounds(bounds);
  }
}
```

LESSON 17

A1:

```
const translator = lang => (strs, ...vals) => strs.reduce(
  (all, str) => all + TRANSLATE(vals.shift(), lang) + str
);
```

LESSON 18

A1:

```
function* dates(date = new Date()) {  
  for (;;) {  
    yield date;  
    // clone the date  
    date = new Date( date.getTime() );  
    // increment the clone  
    date.setDate( date.getDate() + 1 );  
  }  
}
```

LESSON 20

A1:

```
export let luckyNumber = Math.round(Math.random()*10)
export function guessLuckyNumber(guess) {
  return guess === luckyNumber
}
```

LESSON 21

A1:

```
import guessLuckyNumber from './luck_numbery'

let foundNumber = false

for (let i = 1; i <= 50 i++) {
  if (guessLuckyNumber(i)) {
    console.log('Guessed the correct number in ', i,
'attempts!')
    foundNumber = true
    break
  }
}

if (!foundNumber) {
  console.log('Could not guess number.')
}
```

LESSON 23

A1:

```
function
take(n, iterable) {
  i = 0
  const items = []
  for (const val of iterable) {
    items.push(val)
    i++
    if (i === n) break
  }
  return items
}

function* fibonacci() {
  let prev = 0
  let curr = 1
  while(true) {
    yield curr;
    [prev, curr] = [curr, prev + curr]
  }
}

take(10, fibonacci()) // [1, 1, 2, 3, 5, 8, 13, 21, 34,
55]

function* two() {
  yield 1
  yield 2
}

take(10, two()) // [1, 2]
```

LESSON 24

A1:

```
function
union(a, b) {
  return new Set([ ...a, ...b ]);
}

function intersection(a, b) {
  const inBoth = new Set();
  for (const item of union(a, b)) {
    if (a.has(item) && b.has(item)) {
      inBoth.add(item);
    }
  }
  return inBoth;
}

function subtract(a, b) {
  const subtracted = new Set(a);
  for (const item of b) {
    subtracted.delete(item);
  }
  return subtracted;
}

function difference(a, b) {
  return subtract(
    union(a, b),
    intersection(a, b)
  );
}
```

LESSON 25

A1:

```
function
sortMapByKey(map) {
  const sorted = new Map();
  const keys = [ ...map.keys() ].sort();
  for (const key of keys) {
    sorted.set(key, map.get(key));
  }
  return sorted;
}

function invertMap(map) {
  const inverted = new Map();
  for (const [ key, val] of map) {
    inverted.set(val, key);
  }
  return inverted;
}

function sortMapByValues(map) {
  return invertMap(sortMapByKey(invertMap(map)))
}
```


LESSON 27

A1:

```
class Fish {
  hunger = 1;
  dead = false;
  born = new Date();

  constructor(name) {
    this.name = name;
  }

  eat(amount=1) {
    if (this.dead) {
      console.log(`${this.name} is dead and can no longer
eat.`);
      return;
    }
    this.hunger -= amount;
    if (this.hunger < 0) {
      this.dead = true;
      console.log(`${this.name} has died from over
eating.`)
      return
    }
  }

  sleep() {
    this.hunger++;
    if (this.hunger >= 5) {
      this.dead = true;
      console.log(`${this.name} has starved.`)
    }
  }

  isHungry() {
    return this.hunger > 0;
  }
}
```

LESSON 28

A1:

```
class Cruiser extends Car {
  drive(miles=1) {
    const destination = this.milage + miles;
    while(this.milage < destination) {
      if (!this.hasGas()) this.fuel();
      super.drive();
    }
  }

  fuel() {
    this.gas = 50;
  }
}
```

LESSON 30

A1:

```
function loadCreditAndScore(userId) {
  return Promise.all(
    ajax(`/user/${userId}/credit_availability`),
    Promise.race(
      ajax(`/transunion/credit_score?user=${userId}`),
      ajax(`/equifax/credit_score?user=${userId}`)
    )
  )
}

loadCreditAndScore('4XJ').then(([creditAvailability,
creditScore]) => {
  // Do something with credit availability and credit
  score
})
```

LESSON 31

A1:

```
function
loadAsync(url) {
  return new Promise((resolve, reject) => {
    load(url, (error, data) => {
      if (error) {
        reject(error);
      } else {
        resolve(data);
      }
    })
  });
}

function getArticle(id) {
  return Promise.all([
    loadAsync(`/articles/${id}`),
    loadAsync(`/articles/${id}/comments`)
  ]).then(([article, comments]) => Promise.all([
    article,
    comments,
    loadAsync(`/authors/${article.author_id}`)
  ]));
}

getArticle(57).then(
  results => {
    // render article
  },
  error => {
    // show error
  }
);
```

LESSON 32

A1:

```
async function
getArticle(id) {
  const article = await loadAsync(`/articles/${id}`);
  const comments = await
loadAsync(`/articles/${id}/comments`);
  const author = await
loadAsync(`/authors/${article.author_id}`);
  return [ article, comments, author ];
}
```

LESSON 33

A1:

```
function
collect(obs$) {
  const values = [];
  return new Observable(observer => obs$.subscribe({
    next(val) {
      values.push(val);
      observer.next(values);
    }
  }));
}

function sum(obs$) {
  return new Observable(observer => obs$.subscribe({
    next(arr) {
      observer.next(arr.reduce((a, b) => a + b))
    }
  }));
}

sum(collect(Observable.of(1, 2, 3, 4))).subscribe({
  next(val) {
    console.log('sum:', val);
  }
});
```

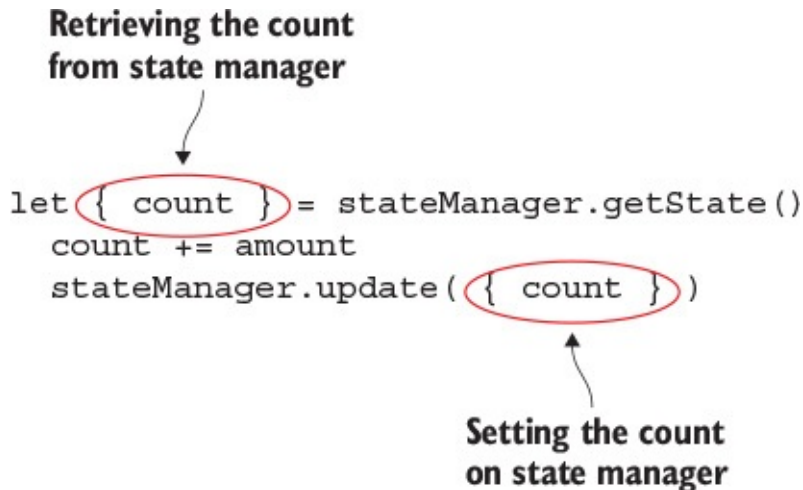
Here's a preview of some of the new syntaxes you'll learn in unit 2

The new destructuring syntax provides a symmetrical counterpart to the existing syntax for data structures.

Retrieving the count from state manager

```
let { count } = stateManager.getState()
count += amount
stateManager.update({ count })
```

Setting the count on state manager



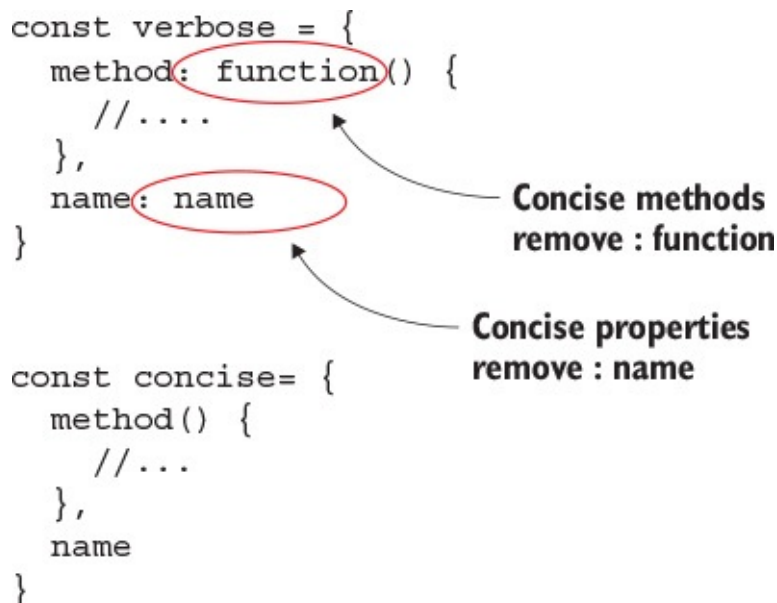
With shorthand properties and methods, you can remove a lot of redundancy from your object literals.

```
const verbose = {
  method: function() {
    //....
  },
  name: name
}

const concise = {
  method() {
    //...
  },
  name
}
```

Concise methods
remove : function

Concise properties
remove : name



Here's a preview of using promises and async functions from unit 7

```
[source,js]
----
async function fetchImage(url) {
  const resp = await fetch(url);
  const blob = await resp.blob();
  return createImageBitmap(blob);
};

fetchImage('my-image.png').then(image => {
  // do something with image
});
----
```

To achieve this without promises or async functions, you would need to write much more obtuse code.

```
[source,js]
----
function fetchImage(url, cb) {
  fetch(url, function(resp) {
    resp.blob(function(blob) {
      createImageBitmap(blob, cb);
    })
  })
};

fetchImage('my-image.png', function(image) {
  // do something with image
});
----
```


Index

[SYMBOL][A][B][C][D][E][F][G][H][I][J][K][L][M][N][O][P][R][S][T][U]
[V][W][Y][Z]

SYMBOL

** operator

+= operator

\$ character

A

action flags

ADD_ITEM value

addEventListener method

addToolTip function

ajax function

aliased parameters

alpha

anonymous functions

arguments object, 2nd

arguments, passing between functions with rest

Array.from method, 2nd

Array.of method, 2nd

Array.prototype.apply

Array.prototype.fill method, 2nd

Array.prototype.find method

Array.prototype.includes method, 2nd

arrayB variable

arrays

constructing

with Array.from

with Array.of

[with Array.prototype .fill](#)

[converting into HTML](#)

[destructuring](#)

[combining with object destructuring](#)

[parameters](#)

[searching](#)

[with Array.prototype .find](#)

[with Array.prototype .includes](#)

[arrow functions](#)

[maintaining context with](#)

[pitfalls when using](#)

[succinct code with](#)

[asterisk](#)

[async functions](#)

[asynchronous code with generators](#)

[error handling in](#)

[asynchronous code, with generators](#)

[at method](#)

[autoFormat](#)

[avg function, 2nd](#)

[await keyword](#)

[AwesomeArray](#)

[axios library](#)

B

[Babel compiler](#)

[configuring](#)

[set up as NPM script](#)

[source maps](#)

[set up of](#)

[setting up Browserify with](#)

[installing Browserify](#)

[setting up projects using Babelify](#)

[transpiling and](#)

[Babelify tool](#)

[.babelrc file](#)

[back-ticks, 2nd](#)

[bar\(\) function](#)

[behavior hooks](#)

[binaryIP function](#)

[binding imported values](#)

[block scope](#)

[blocking](#)

[bomb property](#)

[branches](#)

Browserify tool

[alternatives to](#)

[bundling modules with](#)

[modules in Node.js](#)

[modules, overview of](#)

[installing](#)

[overview of](#)

[setting up with Babel](#)

[built-in tagging function](#)

C

[Candidate stage](#)

[catch method](#)

[cd \(change directory\)](#)

[check method](#)

[choose function](#)

[class keyword](#), [2nd](#)

[classes](#)

[class methods](#)

[declarations](#)

[exporting](#)

[extending](#)

[pitfalls of](#)

[with super](#)

[instantiating](#)

[setting instance properties in class definitions](#)

[static properties of](#)

[clear method](#)

[CoffeeScript](#)

[collect\(\) function](#)

[combinators, of observables](#)

[CommonJS](#)

[compilers](#)

[compiling to JavaScript languages](#)

[complete notification](#)

[computed properties](#)

[configuring Babel](#)

[set up as NPM script](#)

[source maps](#)

[console.log statement](#)

[constants](#)

[overview of](#)

[symbols as](#)

[when to use](#)

constructor method

context, maintaining with arrow functions

ControllableSprite class

coords variable

createImageBitmap function

createStealthBomber function

createStealthSpaceShip function

createTag function, 2nd

cssClass function

curly braces, 2nd

currency function

custom processing, template literals with

D

data variable

DataStore class

DataStore() function

--debug flag

declarations of classes

default

DEFAULT_START

DEFAULT_STEP

DEL_ITEM value

delete function, 2nd

dest folder

destructuring

arrays

combining with object destructuring

parameters of

objects

combining with array destructuring

[parameters of](#)

[renaming variables](#)

[diff function](#)

[difference function](#)

[document.querySelectorAll method, 2nd](#)

[dollar sign character](#)

[domain-specific languages](#)

[See DSLs](#)

[done property, 2nd](#)

[doSomePework function](#)

[double function](#)

[Draft stage](#)

[draw function](#)

[drawImage method](#)

[DriftingSprite class](#)

[DSLs \(domain-specific languages\)](#)

[converting arrays into HTML](#)

[creating helper functions](#)

[HTML-escaping](#)

E

[ECMA \(European Computer Manufacturers Association\)](#)

[ECMAScript](#)

[ES2015](#)

[history of](#)

[specifications](#)

[eggs, Python](#)

[endsWith method](#)

[enhancedSpaceShip function](#)

[entries method](#)

[entry points](#)

[equals sign](#)

[error method](#)

errors

[catching from promises](#)

[handling for promises](#)

[handling in async functions](#)

[ES6 modules, Browserify with](#)

[ES2015](#)

[escape character](#)

[ESNext](#)

[exponent function](#)

[export default statement](#)

[export keyword](#)

[exporting classes](#)

[extending classes](#)

[pitfalls of](#)

[with super](#)

[extends keyword](#)

F

[fadeIn function](#)

[fading](#)

[fat arrow functions](#)

[fetch function, 2nd](#)

[fetchImage function, 2nd](#)

[fill method](#)

[filter function, 2nd](#)

[find and replace](#)

[find function](#)

[findDuplicates function](#)

[findFromCache function](#)

[Finished stage](#)

[flags](#)

[fold function](#)

[Font Awesome](#)

[for loop, 2nd](#)

[for statement](#)

[for..of statement](#)

[forEach method, 2nd](#)

[formattedCurrentUsername](#)

[frame set](#)

[function declaration](#)

[function scope](#)

[functions, passing arguments between with rest](#)

G

[generator functions](#)

[asynchronous code with](#)

[creating infinite lists with](#)

[defining](#)

[using](#)

[generator property](#)

[getArticle\(\) function](#)

[getInstance method](#)

[getJSON function](#)

[getOrgs function](#)

[getRatings method](#)

[getScore function](#)

[getSpriteProperty method, 2nd](#)

[getTransaction function](#)

[getValue function](#)

[getWords function](#)

[global symbols, creating behavior hooks with](#)

[global variables](#)

[globalAlpha property](#)

H

[handleError function](#)

[handleNewRecords function, 2nd](#)

[handleResp function](#)

[has method](#)

[helper functions](#)

[hoisting, with let](#)

[holes](#)

[host property](#)

[html variable](#)

[HTML, converting arrays into](#)

[htmlEscape function](#)

[htmlSafe function](#)

I

[i variable](#)

[i18n \(internationalization\)](#)

[icon property](#)

[icon variable](#)

[if statement](#)

[IIFE \(immediately invoked function expression\)](#)

[ImageBitmap](#)

[images, fetching](#)

[immutability](#)

[immutable push, spread as](#)

[import statement](#)

[importing](#)

[side effects](#)

[values from modules](#)

[in scope variables](#)

[includes method, 2nd](#)

[increment operator](#)

[index.js file](#)

[indexOf method](#)

[indices](#)

[instance properties, setting in class definitions](#)

[instantiating classes](#)

[interlace function](#)

[interlacing](#)

[interrogate function](#)

[intersection function](#)

[invertMap function](#)

[isPressingSpace function](#)

[isStillPlaying function](#)

[items array](#)

[iterables](#)

[for..of statement](#)

[overview, 2nd](#)

[spread, 2nd](#)

[iterator property, 2nd](#)

J

[JavaScript](#)

[compiling to](#)

[files becoming modules](#)

[JSON.stringify\(\) function](#)

K

[keyboard module](#)

[keys method](#)

[keyUsed parameter](#)

L

[lat property](#)

[leaks variable](#)

[length property](#)

[let](#)

[hoisting with](#)

[instead of var](#)

[scope with](#)

[listingId](#)

[lists, creating with generator functions](#)

[literal syntax](#)

[lng property](#)

[load function](#)

[loadAsync\(\) function](#)

[loaders](#)

[location of modules, specifying](#)

[location property](#)

[lock function](#)

[logStats function](#)

[lon variable](#)

M

[map method](#)

[Map.prototype.values](#)

[maps](#)

[creating](#)

[using](#)

[WeakMap](#)

[when to use](#)

[marking strings](#)

[maskEmail function](#)

[method names, shorthand](#)

[middlewares](#)

[migaloo object](#)

[missing values](#)

[mlsId](#)

[MMO \(massively multiplayer online\)](#)

[Model function](#)

[modules](#)

[binding imported values](#)

[breaking apart](#)

[bundling with Browserify](#)

[alternatives to Browserify](#)

[Browserify with ES6 modules](#)

[overview of Browserify](#)

[setting up Browserify with Babel](#)

[creating](#)

[importing side effects](#)

[importing values from](#)

[in Node.js](#)

[JavaScript files becoming](#)

[organizing](#)

[overview of](#)

[rules for](#)

specifying location of

moves() function

multiline strings, 2nd

mutability

mutations

myTag function

N

name property, 2nd

named exports

named functions

named parameters

names

for methods, shorthand

for properties

computed

shorthand

nested array destructuring

new Symbol() function

next() function, 2nd, 3rd, 4th, 5th

nextImage function

Node.js

no-op function

NPM scripts, Babel as

numbers array

O

object keys, symbols as

object literals, syntax of

computed property names

[shorthand method names](#)

[shorthand property names](#)

[Object.assign method](#)

[assigning values with](#)

[extending objects with](#)

[preventing mutations when using](#)

[setting default values with](#)

[Object.getOwnPropertyNames\(\) function](#)

[Object.getOwnPropertySymbols\(\) function](#)

[objects](#)

[destructuring](#)

[combining with array destructuring](#)

[parameters](#)

[extending with Object.assign](#)

[modifying behavior with symbols](#)

[observables](#)

[composing](#)

[creating](#)

[creating combinators](#)

[onerror method](#)

[onload method](#)

[option argument](#)

[organizing modules](#)

[output files](#)

P

[package.json file](#)

[packages, Java](#)

[pad character](#)

[padding strings](#)

[padEnd method](#)

[padStart method](#)

[pagination function](#)

[parameters](#)

[creating aliased](#)

[default](#)

[gathering with rest operators](#)

[of arrays, destructuring](#)

[of objects, destructuring](#)

[simulating named](#)

[to skip recalculating values](#)

[pending state](#)

[pluck function, 2nd](#)

[plugins](#)

[POJO \(plain old JavaScript object\), 2nd](#)

[pop method, 2nd](#)

[programming fundamentals](#)

[Promise.all\(\) function](#)

[Promise.race\(\) function](#)

[Promise.reject\(\) function](#)

[Promise.resolve\(\) function](#)

[promises](#)

[catching errors from](#)

[creating](#)

[error handling for](#)

[helpers](#)

[nested](#)

[using](#)

[property names](#)

[computed](#)

[shorthand](#)

[Proposal stage](#)

[prototypes](#)

[push method](#)

R

[range function, 2nd](#)

[records property](#)

[reduce function, 2nd](#)

[ReferenceError](#)

[RegExp object](#)

[registerUser function](#)

[removeOnExit property](#)

[renaming variables for destructuring](#)

[repeat function, 2nd](#)

[rest](#)

[gathering parameters with](#)

[passing arguments between functions with](#)

[result variable](#)

[return statement](#)

[Rollup](#)

[runner function](#)

S

[scope, with let](#)

[scoping rules](#)

[searching](#)

[arrays](#)

[with Array.prototype .find](#)

with Array.prototype .includes

strings

self documenting

sentencedTo method, 2nd

ServerJS

Set.prototype.size property

setAjaxDefaults

setColors function

setCoordinates function

setInterval

sets

creating

using

WeakSet

setter function

setTimeout function, 2nd

setTitle function

setup() function

setValue function

shape property

shift method, 2nd

short circuited

shorthand

method names

property names

side effects, importing

simulating named parameters

Singleton function

slice method

snitch method

[sortMapByKeys\(\) function](#)

[sortMapByValues function](#)

[source maps](#)

[specifications](#)

[splat](#)

[splice method](#)

[spread, 2nd](#)

[Sprite class](#)

[sprites, controllable](#)

[square brackets](#)

[src directory, 2nd](#)

[start function](#)

[start notification](#)

[startsWith method](#)

[static properties, of classes](#)

[stats.slice\(\) function](#)

[status object](#)

[stop function](#)

[Strawman stage](#)

[string interpolation](#)

[String.prototype.endsWith](#)

[String.prototype.includes method](#)

[String.prototype.padEnd, 2nd](#)

[String.prototype.padStart](#)

[String.prototype.repeat, 2nd](#)

[String.prototype.startsWith](#)

[String.raw function](#)

[strings](#)

[multiline, template literals with](#)

[padding](#)

[searching](#)

[stroke property](#)

[structuring](#)

[sub-blocks](#)

[subtract function](#)

[super\(\) function](#)

[superclass](#)

[surrogate pair](#)

[Symbol\(\) function](#)

[symbols](#)

[as constants](#)

[as object keys](#)

[global, creating behavior hooks with](#)

[modifying object behavior with](#)

[pitfalls when using](#)

T

[tagged template literal](#)

[take\(\) function](#)

[TC39 task group](#)

[TeamStore class](#)

[template literals](#)

[custom processing with](#)

[multiline strings with](#)

[not reusable templates](#)

[overview of](#)

[string interpolation with](#)

[templates module](#)

[temporal dead zones, 2nd](#)

[test function](#)

[that variable](#)

[then\(\) function, 2nd, 3rd](#)

[this variable](#)

[toString\(\) function](#)

[Traceur](#)

[transforms](#)

[translator function](#)

[transparency](#)

[transpiling](#)

[Babel and](#)

[compiling to JavaScript languages](#)

[overview](#)

[triple equals characters](#)

[tryTest function](#)

[type error](#)

U

[undefined values](#)

[Underscore](#)

[Unicode character](#)

[union\(\) function](#)

[unlock function](#)

[Unsplash](#)

[update function](#)

[updateLabel function](#)

[updateMap\(\) function, 2nd](#)

[url property](#)

[UIView class](#)

V

[value property](#)

[values](#)

[assigning with Object.assign](#)

[imported, binding](#)

[importing from modules](#)

[recalculating, skipping with parameters](#)

[setting default with Object.assign](#)

[values method](#)

[var, let instead of](#)

[variables](#)

[declaring with let](#)

[hoisting with let](#)

[let instead of var](#)

[scope with let](#)

[renaming for destructuring](#)

[shadowing](#)

[visualize function](#)

W

[wait function](#)

[WeakMap](#)

[WeakSet](#)

[WebAssembly](#)

[Webpack](#)

[WHATWG \(Web Hypertext Application Technology Working Group\), 2nd](#)

[white space](#)

[withAnonymousFunction object](#)

[withNamedFunction object](#)

[withProps\(\) function](#)

[withShorthandFunction object](#)

Y

[-y flag](#)

[yield keyword](#)

Z

zen-observable

List of Figures

Lesson 5. Declaring constants with const

Figure 5.1. Mutating the value

Figure 5.2. Demonstrating that primitives are immutable

Lesson 6. New string methods

Figure 6.1. Dissecting the padStart function

Lesson 7. Template literals

Figure 7.1. The value of the icon variable is interpolated into the string

Lesson 10. Object.assign

Figure 10.1. Assigning values with Object.assign

Lesson 11. Destructuring

Figure 11.1. Structuring and destructuring syntax

Figure 11.2. Destructuring a product array

Lesson 12. New object literal syntax

Figure 12.1. Using shorthand property names

Figure 12.2. Shorthand method names remove redundant syntax.

Lesson 14. Capstone: Simulating a lock and key

Figure 14.1. Basic multiple choice question

Figure 14.2. Choose the Door game in action

Lesson 16. Destructuring parameters

Figure 16.1. Figuring out which text has been added

(green), changed (yellow), and deleted (red)

Figure 16.2. Simulating named parameters

Lesson 17. Arrow functions

Figure 17.1. Arrow function parentheses rules

Lesson 18. Generator functions

Figure 18.1. A two-way communication channel

Figure 18.2. Generator function execution order

Lesson 19. Capstone: The prisoner's dilemma

Figure 19.1. Prisoner lifecycle

Lesson 22. Capstone: Hangman game

Figure 22.1. A hangman game

Lesson 23. Iterables

Figure 23.1. The rest-spread relationship

Lesson 25. Maps

Figure 25.1. Displaying keys and values of search facets

Lesson 26. Capstone: Blackjack

Figure 26.1. A Blackjack game

Figure 26.2. The dealer's initial hand

Lesson 28. Extending classes

Figure 28.1. Prototypal inheritance

Figure 28.2. How the handleNewRecords method gets
invoked on the prototype chain

Lesson 29. Capstone: Comets

Figure 29.1. Comets game

Lesson 31. Advanced promises

Figure 31.1. Promise error handling

Lesson 34. Capstone: Canvas image gallery

Figure 34.1. Recursive versus repeating promises

List of Tables

Lesson 25. Maps

Table 25.1. Common methods shared by Map and Set

List of Listings

Lesson 2. Transpiling with Babel

[Listing 2.1. Compiling from src folder to a dist folder](#)

[Listing 2.2. Babel stage-0 presets](#)

Lesson 4. Declaring variables with let

[Listing 4.1. Using a free-standing block to keep a variable private](#)

[Listing 4.2. There is a scope problem here, but what is it?](#)

Lesson 8. Capstone: Building a domain-specific language

[Listing 8.1. Abstracting the process of collecting interpolated values](#)

[Listing 8.2. Abstracting of interlacing strings and interpolated values](#)

[Listing 8.3. Processing a template literal](#)

[Listing 8.4. A simple HTML-escaping function](#)

[Listing 8.5. Your tagging function, htmlSafe](#)

[Listing 8.6. Basic HTML-escaping DSL in action](#)

[Listing 8.7. DSL interpolating an array of values, not just one](#)

Lesson 9. New array methods

[Listing 9.1. avg version 1: producing an error, as arguments is not an array](#)

[Listing 9.2. avg version 2: using slice to convert arguments into an array](#)

[Listing 9.3. Updating the avg function to use Array.from](#)

Listing 9.4. Using includes to check whether an array contains a value

Listing 9.5. filter method returns all matches even if you only want one

Lesson 10. Object.assign

Listing 10.1. Basic pizza-tracking map

Listing 10.2. Adding a bomb property to the spaceship

Listing 10.3. Helper function for enhancing spaceships

Listing 10.4. Copying the spaceship

Lesson 12. New object literal syntax

Listing 12.1. Mixing shorthand and longhand object literals

Listing 12.2. Shorthand methods are anonymous functions

Listing 12.3. Recursive function with this.at

Lesson 13. Symbol—a new primitive

Listing 13.1. Using constants as flags

Listing 13.2. The createBomberSpaceShip function from lesson 10

Listing 13.3. Working with well-known symbols

Listing 13.4. Every line here throws an error

Lesson 14. Capstone: Simulating a lock and key

Listing 14.1. Detecting incorrect keys

Listing 14.2. Main interface for the game

Listing 14.3. Building a multiple choice question

Listing 14.4. Combined Choose the Door function

Lesson 15. Default parameters and rest

[Listing 15.1. Accessing this in default parameters](#)

[Listing 15.2. Mandatory parameters must precede optional ones](#)

[Listing 15.3. Fudging default parameters \(not recommended!\)](#)

[Listing 15.4. Calling getRatings twice](#)

[Listing 15.5. Calling getRatings only once](#)

[Listing 15.6. Error occurs if you put params after rest in JavaScript](#)

[Listing 15.7. Params after rest in CoffeeScript](#)

[Listing 15.8. Using rest to forward parameters when monkey-patching](#)

Lesson 18. Generator functions

[Listing 18.1. A generator function in action](#)

[Listing 18.2. Generating an infinite list with a generator function](#)

[Listing 18.3. Generating a Fibonacci sequence](#)

Lesson 19. Capstone: The prisoner's dilemma

[Listing 19.1. A Prisoner Factory with name and generator](#)

[Listing 19.2. interrogate function pits two prisoners against each other](#)

[Listing 19.3. Getting stats and prisoner name from the generator](#)

[Listing 19.4. Putting it all together](#)

[Listing 19.5. Defining more prisoners](#)

Listing 19.6. Interrogating the prisoners 50 times

Lesson 21. Using modules

Listing 21.1. src/format.js

Listing 21.2. src/product.js

Listing 21.3. src/format/date.js

Listing 21.4. src/format/number.js

Listing 21.5. src/format/index.js

Listing 21.6. src/format/index.js

Listing 21.7. src/format/index.js

Lesson 22. Capstone: Hangman game

Listing 22.1. src/words.js

Listing 22.2. src/words.js

Listing 22.3. src/status.js

Listing 22.4. /src/status_display.js

Listing 22.5. /src/letter_slots.js

Listing 22.6. /src/keyboard.js

Listing 22.7. /src/index.js

Listing 22.8. /src/index.js

Listing 22.9. /src/index.js

Lesson 23. Iterables

Listing 23.1. Comparing for..in and for..of

Listing 23.2. Grouping parameters into an array using rest

Listing 23.3. Ungrouping parameters using spread

Listing 23.4. Ungrouping parameters using spread

Listing 23.5. Original functions from unit 1 capstone

Listing 23.6. Updated functions from unit 1 capstone

Lesson 24. Sets

Listing 24.1. Drawing the next frame

Lesson 25. Maps

Listing 25.1. A module that guarantees a singleton instance of any object type

Lesson 26. Capstone: Blackjack

Listing 26.1. src/cards.js

Listing 26.2. src/cards.js

Listing 26.3. src/cards.js

Listing 26.4. src/cards.js

Listing 26.5. src/cards.js

Listing 26.6. src/cards.js

Listing 26.7. src/cards.js

Listing 26.8. src/cards.js

Listing 26.9. src/cards.js

Listing 26.10. src/utils.js

Listing 26.11. Using the wait function

Listing 26.12. src/index.js

Listing 26.13. src/index.js

Listing 26.14. src/index.js

Listing 26.15. src/index.js

Listing 26.16. src/index.js

[Listing 26.17. src/index.js](#)

[Listing 26.18. src/index.js](#)

[Listing 26.19. src/index.js](#)

[Listing 26.20. src/index.js](#)

Lesson 27. Classes

[Listing 27.1. Incorrect syntax for adding methods to classes](#)

[Listing 27.2. Using unbound methods as callbacks](#)

[Listing 27.3. Setting properties on classes](#)

[Listing 27.4. Desugaring static properties](#)

Lesson 28. Extending classes

[Listing 28.1. Extending a class](#)

[Listing 28.2. Invoking the superclass constructor with super](#)

[Listing 28.3. Invoking superclass methods with super](#)

[Listing 28.4. Binding methods in a way that works with super](#)